Recurrent neural networks for radar target identication

Article *in* Proceedings of SPIE - The International Society for Optical Engineering · September 1993
DOI: 10.1117/12.152541

4 authors, including:

Steven K. Rogers
Air Force Research Laboratory
230 PUBLICATIONS **3,076** CITATIONS

SEE PROFILE

AFIT/GSO/ENG/92D-02

RECURRENT NEURAL NETWORKS FOR RADAR TARGET IDENTIFICATION

THESIS

Eric T. Kouba
Captain, USAF

AFIT/GSO/ENG/92D-02

AFIT/GSO/ENG/92D-02

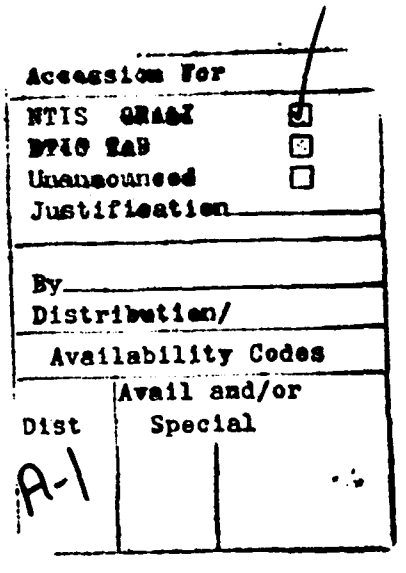
RECURRENT NEURAL NETWORKS FOR RADAR TARGET IDENTIFICATION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Space Operations

Eric T. Kouba, B.S.N.E.
Captain, USAF

December, 1992



Acknowledgments

I would like to thank the three people who made this thesis possible. Maj Steve Rogers taught me how to use neural networks. LtC David Meer taught me about radar. Most of all, I would like to thank LtC Ned Libby (USA) for suggesting this thesis topic in the first place. LtC Libby spent many hours explaining to me the finer details of target recognition.

Eric T. Kouba

Table of Contents

		Page
Acknowledgments		ii
Table of Contents		iii
List of Figures		vii
Abstract		viii
I. Introduction		1-1
1.1 Background		1-1
1.1.1 Identification Approaches		1-1
1.1.2 Radar Signature Parameters		1-2
1.1.3 Polarization Diverse Radar Signature		1-2
1.1.4 Narrowband Radar		1-3
1.1.5 Wideband Radar		1-3
1.1.6 Aspect Angle Dependence		1-3
1.1.7 Aspect Angle Sequences		1-4
1.2 Problem		1-5
1.3 Summary of Current Knowledge		1-5
1.4 Assumptions		1-5
1.5 Scope		1-6
1.6 Approach		1-6
1.7 Organization		1-6
II. Literature Review		2-1
2.1 Introduction		2-1
2.2 Radar Concepts		2-1
2.2.1 Scattering Center		2-1
2.2.2 Scintillation		2-2

2.2.3 Narrowband Radar 2-2
2.2.4 Wideband Radar Range Resolution 2-2
2.2.5 Wideband Radar Range Extent 2-3
2.2.6 Thermal Noise 2-3
2.2.7 Aspect Angle Estimates 2-4
2.3 Decision Processing Algorithms 2-4
2.3.1 Recurrent Neural Network 2-5
2.3.2 Farhat: Optical Neural Network 2-5
2.3.2.1 Farhat: Data 2-6
2.3.2.2 Farhat: Results 2-6
2.3.2.3 Farhat: Limitations 2-6
2.3.3 Brown: Quadratic Classifier, Backpropagation Neural Network, and Counterpropagation Neural Network 2-7
2.3.2.1 Brown: Data 2-7
2.3.2.2 Brown: Results 2-8
2.3.2.3 Brown: Limitations 2-8
2.3.4 Hidden Markov Models 2-8
2.4 Summary 2-9

III. Methodology 3-1
3.1 Introduction 3-1
3.2 Radar Signature Generation 3-1
3.2.1 Aircraft Models 3-1
3.2.2 Radar and Geometry Parameters 3-3
3.2.3 Radar Signature Estimation 3-3
3.3 Feature Extraction 3-5
3.3.1 Aspect Angle Sequences 3-5
3.3.2 Azimuth Estimates 3-6
3.3.3 Scintillation 3-6
3.3.4 Amplitude Thresholding 3-7

3.3.5 Range Bin Windowing 3-7
3.3.6 Amplitude Downsampling 3-9
3.3.7 Data Normalization 3-9
3.4 Decision Processing: Recurrent Neural Networks . . . 3-10
3.4.1 Modifications 3-10
3.4.2 Output Classification 3-10
3.4.3 Epoch Accuracy 3-12
3.4.4 Sequence Length Analysis 3-12
3.5 Test Cases 3-13
3.5.1 Case One: Two Aircraft, No Noise 3-13
3.5.2 Case Two: Five Aircraft, No Noise 3-13
3.5.3 Case Three: Five Aircraft, With Noise . . . 3-14
3.6 Summary 3-14

IV. Results 4-1
4.1 Introduction 4-1
4.2 Case One: Two Aircraft, No Noise 4-2
4.3 Case Two: Five Aircraft, No Noise 4-3
4.3.1 Case Two: Sectors 1, 2, 4, 5, and 6 4-4
4.3.2 Case Two: Sector 3 4-6
4.3.3 Case Two: Summary 4-6
4.4 Case Three: Five Aircraft, With Noise 4-7
4.4.1 Case Three: One Peak Range Bin 4-8
4.4.2 Case Three: Two Peak Range Bins . . . 4-8
4.5 Summary 4-9

V. Conclusions and Recommendations 5-1
5.1 Introduction 5-1
5.2 Conclusions 5-1
5.3 Recommendations for Future Research 5-3

Appendix A. Software Development A-1
A.1 File Structure A-1
A.2 FORMAT_RCST A-1
A.3 RECURRENT A-3

Appendix B. FORMAT_RCST Source Code B-1

Appendix C. RECURRENT Source Code C-1

Appendix D. UTILITIES Source Code D-1

Bibliography BIB-1

Vita VITA-1

List Of Figures

Figure		Page
1-1	Range Bins for Wideband Radar	1-3
1-2	Aspect Angle Sequence	1-4
1-3	Sequence of Wideband Radar Signatures.	1-4
3-1	SCAMP Models of a MiG-21, F-4, F-15, F-16, and F-18	3-2
3-2	Example portion of an RCS_TOOLBOX data file	3-4
3-3	Example of a polarization diverse radar signature	3-5
3-4	Example of the FORMAT_RCST feature extraction process . . .	3-8
3-5	Recurrent Neural Network Diagram	3-11
4-1	Case One: azimuth versus target width	4-3
4-2	Case Two, Sector 1: sequence length versus recognition accuracy . .	4-5
4-3	Case Two, Sector 3: sequence length versus recognition accuracy . .	4-6
4-4	Case Three, effect of additional input features on test set accuracy . .	4-9

Abstract

A real-time recurrent learning algorithm was applied to a five class radar target identification problem. The wideband radar was assumed to measure both kinematic (tracking information expressed as estimated aspect angles) and high range resolution data from a single, isolated aircraft. The aspect angles (azimuth and elevation) of the aircraft relative to the radar were assumed to be constantly changing. This created temporal sequences of high range resolution radar signatures that changed as the aspect angles changed. These sequences were used as input features to a recurrent neural network for three radar target identification test cases. The first test case demonstrated the feasibility of using recurrent neural networks for radar target identification. The second test case demonstrated the relationship between sequence length and target recognition accuracy. For the third test case, the recurrent net achieved 96% test set accuracy under the following conditions: 5 aircraft classes, azimuth range between 60° and 90°, elevation range between +5° and -5°, 1° signature granularity, and signatures corrupted by 5 dBsm scintillation noise.

RECURRENT NEURAL NETWORKS FOR RADAR TARGET IDENTIFICATION

I. Introduction

1.1 Background

Aircraft surveillance is an important Air Force mission. This mission can be divided into three functions: detection, tracking, and identification. These functions must often be performed at long ranges and under adverse weather conditions, so radar is an obvious sensor choice. Current radar systems can detect and track aircraft under a wide range of operational conditions. This thesis focuses on the identification function.

1.1.1 Identification Approaches. Radar target identification is an active research area, with many different available approaches [Cohen, 1991:233-242]. In each approach, identification generally proceeds in four steps: sensor measurement, signal processing, feature extraction, and decision processing [Cohen, 1991:241]. For sensor measurement, a surveillance radar emits a signal towards a target and measures the reflected radar signature. For signal processing, a radar receiver converts antenna measurements into a large set of parameters to represent the radar signature. After signal processing, the radar signature may be represented by hundreds or even thousands of parameters. For feature extraction, an algorithm (in hardware or software) performs data reduction or data compression. After feature extraction, the radar signature is typically represented by tens of parameters or less. For decision processing, an algorithm (usually in software) identifies the target based on information contained within a single radar signature or within multiple radar signatures.

This thesis focuses on developing a decision processing algorithm for radar target identification. Stealth technology, impulse radar, bistatic radar, synthetic

aperture mode radar, and man-in-the-loop systems are not discussed. Following sections assume that the sensor measurement and signal processing steps have already been performed.

1.1.2 Radar Signature Parameters. A radar signature can be described by three types of parameters: frequency, polarization, and amplitude [Trebits, 1984:27-31]. Amplitude values are determined by interactions between a radar signal and a target. These interactions are dependent on the frequency and polarization parameters as set by the radar. Amplitudes are often presented as a radar cross section, measured in decibels relative to a hypothetical one square meter target (dBsm) viewed under the same operational conditions (transmit power, antenna gain, range, etc.) [Stimson, 1983:115, 170-174]. A radar can measure amplitudes at a single frequency (for narrowband radar) or over a range of frequencies (for wideband radar). Antenna polarizations can be vertical, horizontal, circular left, circular right, or elliptical. Different polarizations may be used during the transmit and receive cycles. If a radar transmits two signals (using orthogonal polarizations) and receives each signal twice (using the same orthogonal polarizations), then the resulting radar signature is fully polarized. A fully polarized radar signature contains all available radar information about a target [Trebits, 1984:30].

1.1.3 Polarization Diverse Radar Signature. For most radar target identification cases, it is not necessary to measure a fully polarized radar signature. If a radar transmits a circularly polarized signal and receives using orthogonal linear polarizations, then the resulting radar signature is polarization diverse [Sacchini, 1992a:127]. A polarization diverse radar signature is a compressed form of a fully polarized signature, with some loss of information. This information loss is negligible as long as the target does not respond preferentially to one circular polarization and not the other [Sacchini, 1992:127]. This thesis will assume all radar signatures are polarization diverse (using left circular transmit polarization) unless specifically noted otherwise.

1.1.4 Narrowband Radar. For narrowband radar, the measured amplitudes are vector-sum values for the entire target. A polarization diverse, narrowband radar signature contains two parameters: the LV amplitude for the left circular transmit - vertical receive polarization, and the LH amplitude for the left circular transmit - horizontal receive polarization.

1.1.5 Wideband Radar. For wideband radar, amplitude values are measured at very precise, discrete times to gain additional information about a target. Since radar waves travel at the speed of light, measurements over discrete time intervals correspond to discrete range intervals (or range bins) across the target (Figure 1-1). Amplitude values are measured for each range bin as if it were an individual radar target. A polarization diverse, wideband (or high range resolution) radar signature is a matrix containing LV and LH amplitudes for each range bin across the target.

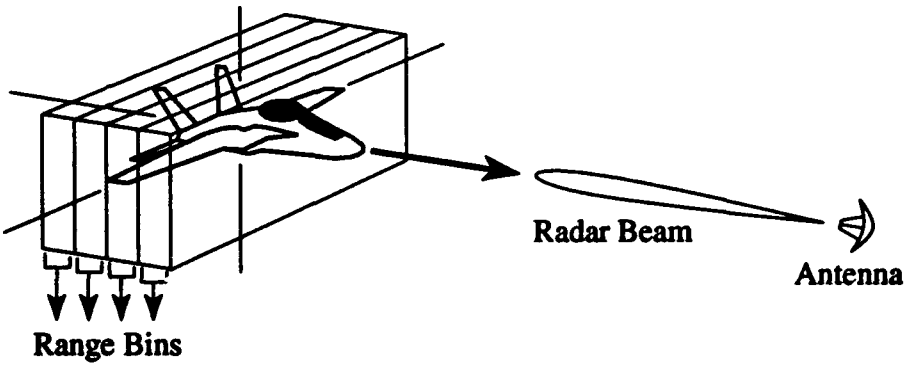


Figure 1-1: Range Bins for Wideband Radar [Mensa, 1991:6]

1.1.6 Aspect Angle Dependence. A radar signature measures the radar reflective properties of an aircraft. However, these reflective properties change as the aircraft is viewed from different aspect angles (azimuth and elevation) [Trebits, 1984:44]. For this thesis, azimuth and elevation are measured in the aircraft frame of reference from the aircraft towards a radar (Figure 1-2). A given aircraft viewed at given aspect angles will produce a particular radar signature (Figures 1-2 and 1-3). If a given signature is unique to a particular aircraft class, then a decision processing

algorithm can identify that aircraft based on the information contained within a single radar signature. However, this "unique signatures" condition is not often encountered with real radar targets. Radar signatures from different aircraft are sometimes essentially identical at the same or different aspect angles [Libby, 1992a:Ch 1, 6-8]. When this happens, additional information must be measured before an aircraft can be identified.

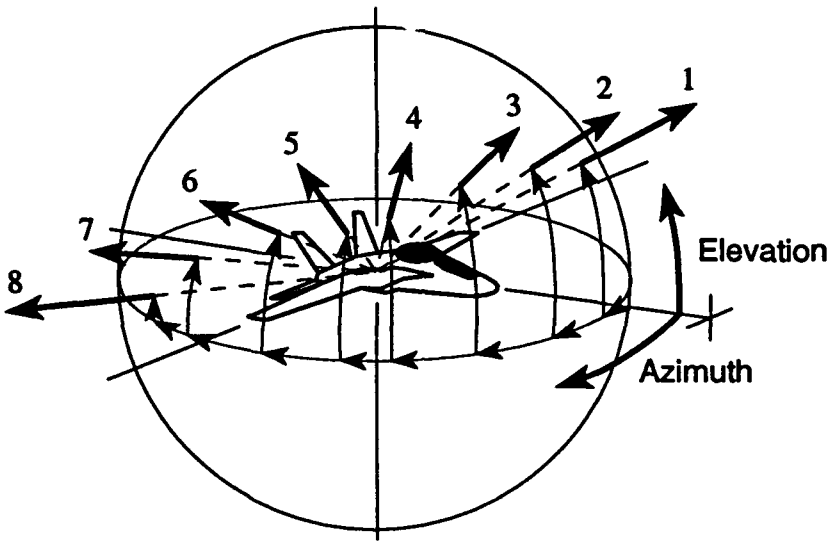


Figure 1-2: Aspect Angle Sequence [Libby, 1992a:Ch 1, 9]

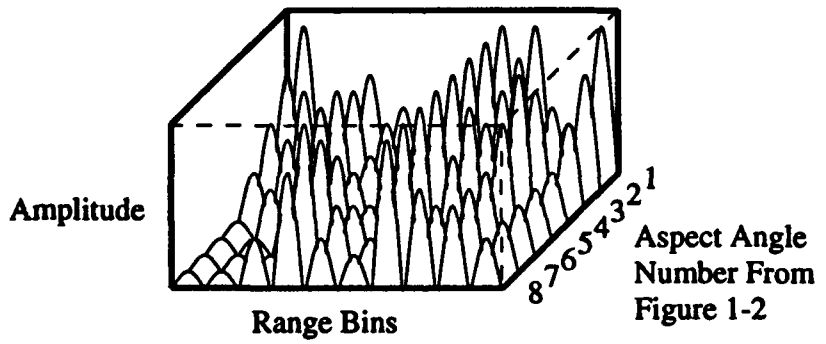


Figure 1-3: Sequence of Wideband Radar Signatures

1.1.7 Aspect Angle Sequences. Existing radars can provide this additional information [Libby, 1992a:Ch 1, 6-8]. Since an aircraft in flight is constantly in motion, a radar can measure sequences of radar signatures as the aspect angles change

(Figures 1-2 and 1-3). A *sequence* of radar signatures is much more likely to be unique than its component signatures. There are an infinite number of possible aspect angle sequences, so *a priori* aspect angle estimates are needed to limit the number of candidate sequences. A radar can generate these aspect angle estimates by using kinematic information from aircraft position and velocity measurements. The aspect angle estimates and temporal sequences of radar signatures are additional features that can be used to discriminate between aircraft classes.

1.2 Problem

The goal of this thesis is to identify various types of aircraft using sequences of high range resolution radar signatures.

1.3 Summary of Current Knowledge

Only a few radar target identification algorithms have been developed that use sequences of radar signatures (temporal sequences based on target aspect angle changes). These algorithms include an optical neural network [Farhat, 1989:670-680], a quadratic classifier, a backpropagation neural network, and a counterpropagation neural network (the last three by [Brown and others, 1990:217-224]). However, there are other algorithms from classical sequence analysis disciplines that have not yet been used for radar target identification [Libby, 1992b]. These algorithms include hidden Markov models and recurrent neural networks. Since hidden Markov models [DeWitt, 1992:Ch 1, 1] and other stochastic estimation techniques [Libby, 1992a:Ch 1, 11] for radar target identification are currently under research here at AFIT, this thesis focuses on recurrent neural networks.

1.4 Assumptions

- The radar operates at 10 GHz frequency with a 1 GHz bandwidth.
- Radar signatures are measured in the time domain from a single, isolated aircraft.
- The aircraft aspect angles are changing at a constant rate.

- Radar signatures are strong enough to be clearly seen above thermal noise.
- Radar signatures are corrupted only by scintillation effects.

1.5 Scope

This thesis focuses on solving a five class radar target identification problem by using recurrent neural networks [Lindsey, 1991:1-3] [Williams and Zipser, 1989:270-280].

1.6 Approach

Radar target identification using recurrent neural networks was performed in three steps. First, the RCS_TOOLBOX computer program generated a database of high range resolution radar signatures (provided by MIT Lincoln Laboratories Group 93 [Bramley and others, 1991]). Second, these signatures were processed into sequences of feature values for use during recurrent neural network training and testing. Third, a recurrent neural network computer program (implementing a real time recurrent learning algorithm) was trained to recognize sequences of radar signatures from different aircraft classes [Lindsey, 1991:58-74]. Radar target identification accuracy was verified against an independent set of RCS_TOOLBOX test sequences.

1.7 Organization

The remaining sections of this thesis are organized into four chapters. Chapter II discusses several important radar concepts and summarizes a few decision processing algorithms that can use sequences of radar signatures for radar target identification. Chapter III presents the procedures for generating radar signatures, adding noise, extracting feature values, training a recurrent neural network, and testing for identification accuracy. Chapter IV gives the experimental results for three test cases. Chapter V gives conclusions and recommendations for future research.

II. Literature Review

2.1 Introduction

This chapter discusses several important radar concepts that apply to this thesis: scattering centers, scintillation, narrowband radar, wideband range resolution, wideband range extent, thermal noise, and aspect angle estimates. This chapter also summarizes a few decision processing algorithms that can use sequences of radar signatures for radar target identification. If an algorithm has been used previously for radar target identification, the associated data files, algorithm results, and limitations are discussed.

2.2 Radar Concepts

A radar signature is created by interactions between a radar signal and a target. These interactions are a function of geometry between the radar and target, and reflectivity properties of the target at the radar signal frequency and polarization [Barton, 1988:108]. For simple targets (spheres, plates, cylinders, cones, and corner reflectors), radar signatures can be analytically or approximately computed [Trebits, 1984:31-43]. For complex targets such as aircraft, radar signatures must be either physically measured or numerically estimated [Barton, 1988:109]. This thesis uses the RCS_TOOLBOX computer program to numerically estimate radar signatures [Bramley and others, 1991].

2.2.1 Scattering Centers. A complex target can be viewed as a set of small, simple targets (scattering centers) [Trebits, 1984:44]. The radar signature for each scattering center can be computed, so the radar signature for a complex target can be numerically estimated by coherently summing the individual radar signatures from the component scattering centers. This coherent sum is based on the addition and cancellation of scattering center amplitudes as adjusted by each scattering center's phase. Each scattering center can have different reflectivity properties, depending on materials used for that part of the target. Particular scattering centers may be partially or completely obscured when the target is viewed from certain aspect angles: these

effects must be accounted for using three-dimensional geometry. Interference effects (caused by multiple radar signal bounces between scattering centers) may be included for accuracy or ignored for computational speed: RCS_TOOLBOX ignores interference effects.

2.2.2 Scintillation. Scintillation is a problem for radar target identification. It causes a radar signature to vary rapidly with time and with target aspect angle. It is caused by scattering center amplitude and phase variations and by interactions between returns from individual scattering centers [Stimson, 1983:192-193]. Amplitude and phase variations can happen for a variety of reasons: control surface motion, rotating propellers, structural flexing, vibrations, atmospheric effects, etc. If a scattering center is rotated by a small amount, then its amplitude will change. If a scattering center moves along the line of sight between the aircraft and radar, then its phase will change. Slight motions are sufficient to cause large phase changes: a 10 GHz radar operates at a 3 cm wavelength, and a quarter wavelength shift will change a scattering center's phase by 180° [Stimson, 1983:86,192-193]. As the amplitude and phase values change, the total amplitude for the target will change. For this thesis, scintillation effects were included as the primary source of radar system noise. RCS_TOOLBOX computed noise-free radar signatures, so scintillation effects had to be added using a separate computer program (FORMAT_RCST, as described in Section 3.3.3).

2.2.3 Narrowband Radar. For narrowband radar, all scattering center signatures from a complex target are coherently summed into a single "range bin". For this reason, scintillation makes narrowband signatures fluctuate rapidly with time and aspect angle. This also makes narrowband signatures a poor feature set for aircraft identification.

2.2.4 Wideband Radar Range Resolution. For wideband radar, range resolution is a measure of how closely two targets may be placed (along the line of sight between the radar and the targets) before the two radar signatures begin to

overlap. It is determined by the radar signal bandwidth [Mensa, 1991:9]. For complex targets, the range resolution parameter effectively divides the object into range "bins". The radar signatures from scattering centers within the same range bin will be coherently summed into a single signature for that range bin. This makes wideband radar resistant to scintillation effects [Skolnik, 1980:181-182]. Scintillation still causes random fluctuations in the scattering center radar signatures, but the effects are contained within individual ranges bins.

2.2.5 Wideband Radar Range Extent. A polarization diverse, wideband radar signature is a matrix containing LV and LH amplitudes for each range bin within the range extent. Range extent is the window of ranges over which a wideband radar will measure target information. A complete discussion of the factors affecting range extent is beyond the scope of this thesis. In any case, range extent must be large enough to completely cover a target. For most aircraft, a range extent on the order of tens of meters is sufficient.

2.2.6 Thermal Noise. Thermal noise is present in the output of every radar receiver. The total noise power level can be expressed as the finite sum of noise contributions from the atmospheric background, antenna, transmission lines, and receiver [Stimson, 1983:169]. For wideband radar, the finite noise power level causes each range bin to contain a finite amount of noise energy. This noise energy (in joules) can be expressed as an equivalent noise amplitude (in dBsm) by figuring out how much energy would be measured from a hypothetical one square meter target (for a given radar under specific operating conditions) [Barton, 1988:16]. For a target scattering center to be detected, the energy in that range bin due to the scattering center must be greater than the energy due to noise. Noise effects can be filtered out of a radar signature by setting an amplitude threshold value: anything below the threshold is considered to be noise and anything above it is part of a target [Stimson, 1983:175]. For this thesis, the amplitude threshold was assumed to be high enough to filter out all thermal noise effects.

2.2.7 Aspect Angle Estimates. Tracking information (position and velocity) can be used to estimate aspect angles by assuming an aircraft is constrained to aerodynamic flight. Basic geometry can provide these estimates by assuming the aircraft is in straight, level, upright, zero angle-of-attack flight. Advanced techniques (Kalman filters and coordinate transformations) are required to handle the more general cases [Bogler, 1990:147-199] [Libby, 1992a:Ch 3, 13-21].

Two types of aspect angle uncertainty are required to model a radar target tracking algorithm [Libby, 1992b]. First, there is an overall "bias" uncertainty that causes an offset between the target actual and estimated aspect angles. It is caused mainly by uncertainty in the aircraft's aspect angle relative to its velocity vector (sideslip angle, angle of attack, etc.). The offset error affects all aspect angle estimates for a given target and will persist over a sequence of radar measurements. The offset error can have a standard deviation of as much as a few degrees: this accounts for most of the uncertainty within a tracking algorithm [Libby, 1992b]. The second type of uncertainty causes small, random deviations from the target's apparent aspect trajectory. It is caused mainly by high frequency aerodynamic effects (aircraft control response, wind buffeting, etc.). For a typical sequence of wideband radar signatures, these deviations are uncorrelated in time [Libby, 1992b]. In stable flight, these random fluctuations are expected to have a standard deviation of less than one degree [Libby, 1992b].

For current radar systems, aircraft azimuth and elevation angles can be estimated within $\pm 5°$ (including both types of uncertainty) [Libby, 1992b]. A decision processing algorithm should take advantage of every possible source of information, therefore this thesis uses the aspect angle estimates as feature values.

2.3 Decision Processing Algorithms

This section describes several decision processing algorithms that can use sequences of radar signatures for radar target identification. If the algorithm has been used previously for radar target identification, the associated data files, algorithm results, and limitations are discussed.

2.3.1 Recurrent Neural Network. Williams and Zipser developed a recurrent neural network algorithm that can learn supervised temporal tasks [Williams and Zipser, 1989:270-280]. This algorithm has been used previously for function prediction [Lindsey, 1991:2] and word recognition [Robinson, 1992], but not for radar target identification. Similar to other neural nets, a recurrent net multiplies an input vector by a weight matrix to get an output vector (a computationally simple operation). For radar target identification, the input vector is a set of numbers representing a radar signature (the amplitudes from a polarization diverse signature). Aspect angle estimates are included as separate feature values in the input vector. The weight matrix is trained using labeled input vectors and gradient descent: this process minimizes the mean-squared error between the desired and actual output vectors. The output vector includes both output nodes and hidden nodes. The index of the maximum value within the output nodes identifies the target.

To process temporal sequences, the output vector for a given time step is fed back as additional inputs for the next time step. The input vector includes the aspect angles and radar signature values for the current iteration and the output vector as calculated during the previous iteration. Output vectors are computed for each time step.

Recurrent nets appear to be ideally suited for processing sequences of radar signatures for two reasons. First, sequences may contain an arbitrary number of signatures. Second, a recurrent net will retain temporal information for fixed or indefinite periods of time as required. The biggest limitation for recurrent nets is computational speed during training (but not testing) [Lindsey, 1991:9-10].

2.3.2 Farhat: Optical Neural Network. Farhat developed an optical neural network algorithm (based on a feed-forward net, not a recurrent net) that uses sequences of wideband radar signatures to identify airborne targets [Farhat, 1989:670-680]. Similar to other neural nets, it multiplies an input vector by a weight matrix to get an output vector. However, this algorithm performs all of its operations optically. The input vector is a sinogram, which is a Cartesian plot of the radar intensities of

scattering centers on a target versus target aspect angle (similar to Figure 1-3). Each sinogram is converted into a 2-dimensional binary image representing an entire radar sequence. The weight matrix is computed (not trained) by converting a complete set of 2-dimensional training images into a 4-dimensional connectivity matrix. The connectivity matrix is partitioned back into a 2-dimensional representation for optical storage on a hologram. The output vector is an optical text label that identifies the target.

2.3.2.1 Farhat: Data. Training and test sequences were measured for three target classes: a B-52, an AWACS, and a Space Shuttle. Real data sets were obtained by placing three physical models on a turntable in a radar test range (16 GHz to 17 GHz, wideband). Fully polarized signatures were measured for each target. Aspect angle changes were limited to azimuth only. Training sequences were measured for each azimuth quadrant: these were converted into sinograms and digitized into 32 by 32 pixel binary images. Test sequences were generated by windowing the azimuth axis of the binary images. Zero padding filled out the rest of the test images.

2.3.2.2 Farhat: Results. Target identification was accurate for test sequences with 18° or more of azimuth angle changes. Accuracy losses were noticeable for test sequences with 9° azimuth angle changes. These results apply to a three class identification problem with azimuth angles known exactly. Note: Farhat did not explain how the azimuth angle measurements would be obtained.

2.3.2.3 Farhat: Limitations. Data storage is the biggest limitation for the Farhat algorithm. The optical storage algorithm can retain approximately 37 images (each 32 by 32 pixels) without degradation [Farhat, 1989:675]. Each image stores information for a 90° azimuth angle change for a single target. Therefore, the Farhat algorithm can be "saturated" by measuring the radar signatures from three targets for a full 360° azimuth rotation at three discrete elevation angles (36 images total). The Farhat algorithm can be modified to store more images, but there will always be an upper limit for a given system: this is less than optimal because a radar target

identification system should be able to handle many targets and a potentially infinite number of aspect angle sequences [Cohen, 1991:238].

2.3.3 Brown: Quadratic Classifier, Backpropagation Neural Network, and Counterpropagation Neural Network. Brown developed three algorithms that use fixed-length sequences of narrowband radar amplitudes to identify three classes of ground targets [Brown and others, 1990:217-224].

The first algorithm is a quadratic (Bayesian-like) classifier. It computes the probability that a particular target is present given that a particular radar sequence has been received. This algorithm measures the Mahalanobis distance between an unknown sequence (a vector of radar amplitudes) and all known sequences. The minimum distance identifies the target.

The second algorithm is a multi-layer perceptron neural network. It multiplies an input vector by two weight matrices (two layers of hidden nodes) to get an output vector. The input vector is a set of numbers that represent the amplitudes of a radar signature. The weight matrices are trained using labeled input vectors and gradient descent: this process minimizes the mean-squared error between the desired and actual output vectors. The index of the maximum value within the output vector identifies the target.

The third algorithm is a counterpropagation neural network that combines a Kohonen map and a Grossberg outstar net. This algorithm is trained with methods similar to those used for a backpropagation net. After training, the Kohonen map takes an input vector and maps it to an intermediate feature space of discrete nodes. These nodes represent the discrete probability density function of all input vectors. The Grossberg outstar net approximates a look-up table and labels each Kohonen node: these labels identify the target.

2.3.3.1 Brown: Data. Training and test sequences were measured for three classes: a tank, a truck, and an armored personnel carrier. Real data sets were obtained by placing three physical models on a turntable in a radar test range (35 GHz,

narrowband). Fully polarized signatures were measured for each target. Aspect angle changes were limited to azimuth only. Radar sequences were represented by a vector of 8 feature values (for the quadratic classifier) or 21 feature values (for the neural networks): Brown did not specifically identify what these values represented. A total of 2932 sequences were measured: half for training and half for testing.

2.3.3.2 Brown: Results. The average quadratic classifier accuracy was 73% for a three class, rotation-invariant problem. The average backpropagation net accuracy was 74%. The average counterpropagation net accuracy was 70%. The neural network results apply to a four class (three targets plus "unidentified"), rotation-invariant problem. Brown identified the backpropagation net as having the best potential.

2.3.3.3 Brown: Limitations. The fixed sequence length is the biggest limitation for the Brown backpropagation algorithm. The sequence length is defined by the input vector size, so the training and testing vectors for a given net architecture must be the same length. This means the appropriate sequence length (to maximize radar target identification accuracy) for a backpropagation net can be found only by trial and error during training. Recurrent neural networks are more efficient for sequence length analysis: a recurrent net outputs the estimated target identity at each time step during a sequence [Williams and Zipser, 1989:271].

2.3.4 Hidden Markov Models. Hidden Markov models have been used in the past for isolated word recognition [Parsons, 1987:307-317] and have only recently been proposed for radar target identification [Libby, 1992a:Ch 3, 49-50]. For aircraft identification, a hidden Markov model assumes a system (aircraft) must be in one of a finite number of states (aspect angles) and that each state has a finite number of possible outputs (radar signatures) [DeWitt, 1992:Ch 3, 9-11]. The system transitions through a sequence of states and each state emits an output. The transitions and outputs are assumed to be random with defined probabilities. Therefore, a sequence of

measured outputs can be analyzed to see if they match the possible outputs from a given system. The best possible match identifies the system.

2.4 Summary

A recurrent neural network has two key advantages over the radar target identification algorithms used in the past. First, the memory storage capacity (for the number of targets and training sequences) is not arbitrarily limited, unlike an optical neural network. Second, data sequences of arbitrary length can be handled by the same recurrent neural network architecture, unlike a backpropagation neural network. Since hidden Markov models [DeWitt, 1992:Ch 1, 1] and other stochastic estimation techniques [Libby, 1992a:Ch 1, 11] for radar target identification are under research here at AFIT, this thesis focuses on recurrent neural networks.

III. Methodology

3.1 Introduction

Radar target identification using recurrent neural networks was performed in three steps. First, the RCS_TOOLBOX computer program generated a database of high range resolution radar signatures [Bramley and others, 1991]. Second, these signatures were processed into sequences of feature values for use during recurrent neural network training and testing. Third, a recurrent neural network computer program (implementing a real time recurrent learning algorithm) was trained to recognize sequences of radar signatures from five different aircraft classes [Lindsey, 1991:58-74]. Radar target identification accuracy was verified against an independent set of RCS_TOOLBOX test sequences.

3.2 Radar Signature Generation

The RCS_TOOLBOX computer program is actually a collection of over a dozen separate programs. Three of these programs are required to generate high range resolution radar signatures of aircraft: SCAMP, SETUPPTD, and PTDTABLE. These programs, in software, put an aircraft model on a turntable in a radar test range [Bramley, 1991:Ch 2, 1-8]. This is the same experimental setup as for a real radar test range.

3.2.1 Aircraft Models. SCAMP creates a computer file to represent the three dimensional geometry of an aircraft (Figure 3-1) [Bramley, 1991:Ch 3, 1-8]. Aircraft models of a MiG-21 and F-4 were created and verified here at AFIT [Libby, 1992b]. Aircraft models of an F-15, F-16, and F-18 came with the RCS_TOOLBOX package. These last three models have not been verified, but they can be used to produce deterministic radar signatures. None of the five SCAMP models were validated against real radar signatures.

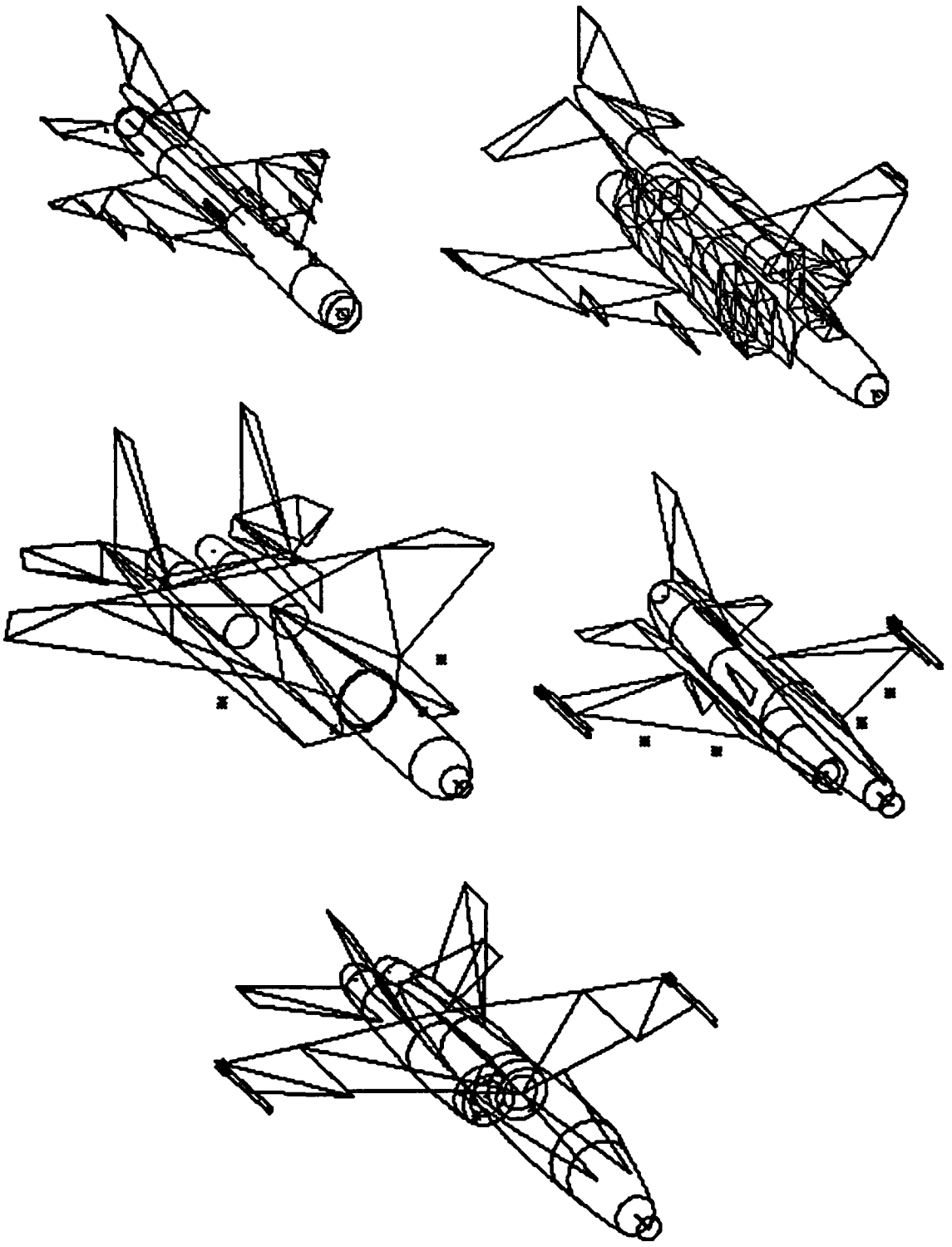


Figure 3-1: SCAMP Models of a MiG-21, F-4, F-15, F-16, and F-18

3.2.2 Radar and Geometry Parameters. SETUPPTD defines the operating conditions under which wideband radar signatures are generated [Bramley, 1991:Ch 8, 1-7]. The radar sensor is defined by setting polarization, frequency, bandwidth, range resolution, and range extent parameters. The aircraft orientation relative to the radar is controlled by defining various turntable motion parameters. The aircraft is placed on the turntable at defined roll and pitch angles. The turntable itself may be rotated and tilted as required. In effect, these orientation parameters define the aircraft aspect angles. The coordinate transformation between the orientation parameters and aspect angles can be simplified by setting the roll and pitch angles to zero. This allows the tilt angle to control elevation and turntable rotation to control azimuth. Once the initial aircraft orientation is fixed, the turntable may be rotated through a series of discrete angular steps. This allows PTDTABLE to generate multiple radar signatures as the aspect angles change.

3.2.3 Radar Signature Estimation. PTDTABLE uses information contained in the SCAMP and SETUPPTD files to generate high range resolution radar signatures [Bramley, 1991:Ch 9, 1-3]. This program uses the physical theory of diffraction to estimate signatures and assumes that the aircraft dimensions are large compared to the radar wavelength. PTDTABLE estimates radar signatures (for a specific SCAMP model) at a number of discrete aspect angles (as defined in the SETUPPTD file). For this thesis, wideband radar signatures were generated in the time domain for the following conditions:

- 10 GHz radar center frequency
- 1 GHz bandwidth
- 0.15 meter range resolution (defined by the bandwidth [Mensa, 1991:9])
- 30 meter range extent (producing 200 range bins, each 0.15 meters across)
- 5 aircraft classes: MiG-21, F-4, F-15, F-16, and F-18
- Primary aspect angles: -5° to +5° elevation in 1° steps, with 1° azimuth granularity from 0° to 180° (for training data)

- Offset aspect angles: -4.5° to +4.5° elevation in 1° steps and also at 0° elevation, with 1° azimuth granularity from 0.5° to 179.5° (for independent test data)

- Polarization diverse radar signatures: amplitudes measured at LV and LH polarizations (left circular transmit, vertical and horizontal receive)

Each radar signature took about 5 seconds to compute on a Silicon Graphics 4D workstation. Figure 3-2 shows part of an RCS_TOOLBOX file. Figure 3-3 plots the resulting radar signature.

```
wbtable
Recurrent Neural Networks for Radar Target Identification
LV polarization, elevation 0.0, azimuth 0.0 start
jets/mig21.scamp
  1                                  :number of frequencies
  10000.0000
  4  1                               :transmitter and receiver polarizations
    0.0000     0.0000     0.0000     :roll,pitch and tilt
  0                                  :no. of pts to skip between shadowing
    0.0000     1.0000                :thet0 and delta-theta
    0.0000     0.0000                :psit0 and delta-psit
    0.0000     0.0000                :psir0 and delta-psir
  181                                :number of pulses
    0.1500                           :range bin size
 1000.0000                           :band width
  200                                :range wdth/srsc
    3                                :imode
     0.000
 14335 33969  14324 15758  14383 33667  14370 15508
 14390 33379  14413 15083  14425 33212  14449 14777
 14461 32731  14479 14603  14452 32449  14637 14118
 14483 32153  14564 13963  14586 31802  14604 13520
 14632 31620  14646 13254  14678 31157  14690 12989
 14735 30868  14739 12701  14781 30426  14795 12460
 14829 30137  14864 12132  14874 29901  14914 11706
 14937 29563  14994 11467  14999 29208  15043 11109
 15072 28900  15112 10813  15151 28571  15182 10475
 15224 28262  15265 10126  15314 27966  15340  9838
 15401 27648  15444  9471  15493 27344  15543  9161
 15600 26991  15656  8845  15714 26679  15776  8513
 ...
(first radar signature continues, followed by other signatures)
```

Figure 3-2: Example portion of an RCS_TOOLBOX data file, showing the LV polarization data for a MiG-21 at 0° elevation, 0° azimuth. The array of numbers contains the amplitude and phase values for each range bin [Bramley, 1991:Ch 9, 1-3]. The amplitude and phase values are transformed for integer representation [Bramley, 1991:Ch A, 13].

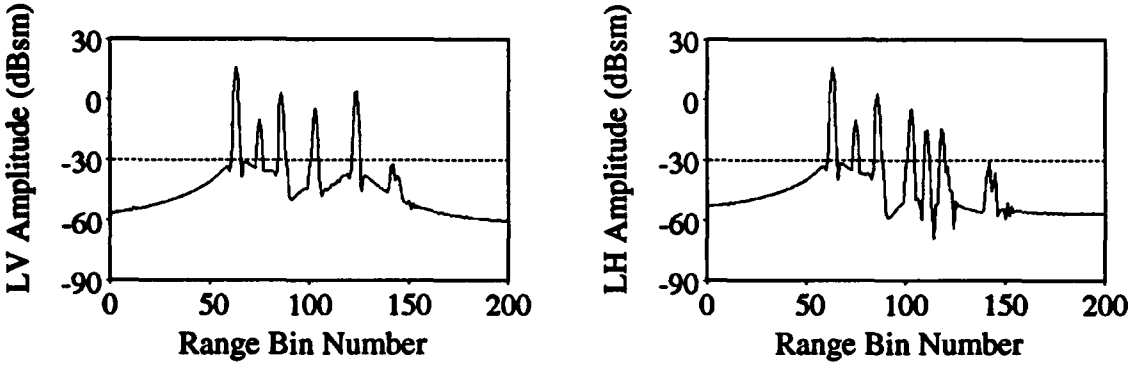


Figure 3-3: Example of a polarization diverse radar signature, showing the LV and LH polarization responses of a MiG-21 at 0° elevation, 0° azimuth [Bramley and others, 1991:Ch A, 11-14]. The line at -30 dBsm shows the thermal noise threshold.

3.3 Feature Extraction

This section describes the FORMAT_RCST data processing computer program. For the program source code, see Appendix B. FORMAT_RCST reads in the raw RCS_TOOLBOX signature data files and performs a number of operations. Each of these operations is described in turn:

- Select a starting aspect angle for a sequence of radar signatures.
- Produce an azimuth estimate by adding noise to the true azimuth value.
- Add scintillation noise to each of the amplitude values in a signature.
- Apply a lower threshold limit to all amplitude values.
- Locate the target within the range extent and estimate target width.
- Divide the target into range bin intervals and downsample to a fixed number of peak range bins.
- Statistically normalize feature values to improve neural network training.

3.3.1 Aspect Angle Sequences. For training data files, the sequence starting aspect angles were randomly generated (selected one at a time without replacement). This improved the chances that the recurrent neural network algorithm would converge to a global minimum mean squared error solution instead of a local minimum. Multiple

data files were used for training and the resulting test accuracies were compared. Training data files were generated only from the primary aspect angles. Test data files were generated only from the offset aspect angles (ref: Section 3.2.3).

For all data files, sequences were generated by varying only the aircraft azimuth. Elevation was constant for any one sequence, but could vary between sequences. Within each sequence, the azimuth (truth) angle increased by 1° per time step for a total of 10 time steps (10°). Reverse direction (decreasing azimuth) sequences were not included. The 10° sequence length was chosen to be the upper limit for data analysis in this thesis. A recurrent net outputs the estimated target identity at each time step in a sequence [Williams and Zipser, 1989:271], so the 10° limit allowed all sequences shorter than 10° to be analyzed automatically.

3.3.2 Azimuth Estimates. The aspect angle offset (bias) was generated using a Gaussian distribution with a standard deviation of 2 degrees. Aspect angle random deviations were generated using a Gaussian distribution with a standard deviation of 0.2 degrees. Random numbers from both distributions were limited to ± 3 standard deviations. These values caused an azimuth uncertainty slightly larger than the ± 5° accuracy quoted for typical radar tracking algorithms [Libby, 1992b]. Azimuth estimates were computed each time a signature was placed in a sequence.

3.3.3 Scintillation. RCS_TOOLBOX computed noise-free amplitude values for all radar signatures. For this thesis, scintillation was the only source of radar system noise. Noise was added by taking an RCS_TOOLBOX amplitude value, adding to it a random number (Gaussian distribution with a given standard deviation), and using the result as a representation for a range bin amplitude measurement in a noisy radar signature. Noise contributions were limited to ±3 standard deviations: this prevented the very low probability extremes of the Gaussian distribution from affecting the data sets [Libby, 1992b]. Noise was added to each amplitude in the raw RCS_TOOLBOX signatures before thresholding (ref: Section 3.3.4). Note: this procedure does not exactly model scintillation noise as it affects real radar systems. However, it is sufficient to produce random amplitude variations that are similar to actual scintillation

effects [Sacchini, 1992b]. This procedure was intended to simulate *only* scintillation noise effects *above* the threshold limit (Figure 3-4).

The standard deviation for amplitude noise was 5 dBsm. This value was chosen to simulate the effects of typical radar system noise [Libby, 1992b]. Noise effects were computed each time a signature was placed into a sequence: a given (truth) aspect angle was represented by multiple radar signatures, each with its own noise realizations.

3.3.4 Amplitude Thresholding. Each RCS_TOOLBOX signature contained 200 range bins (with 200 LV and 200 LH amplitudes making up the polarization diverse radar signature). This was too much information for a recurrent neural network to process efficiently (using the computer resources available at AFIT), so some sort of amplitude feature extraction was required. This thesis used a combination of amplitude thresholding, windowing, and downsampling. Thresholding was chosen because information in a radar signature is contained in the high amplitude peaks, not the low amplitude nulls [Sacchini, 1992:125]. Therefore, all amplitude values were compared against a -30 dBsm lower threshold value (Figure 3-4). For this thesis, it was assumed that all thermal noise effects were eliminated by the same amplitude threshold. The windowing and downsampling routines are described in the following sections.

3.3.5 Range Bin Windowing. The amplitude threshold was also used to locate the aircraft within the 200 range bins (the range extent). Starting from the front of the range extent, the range bin amplitude values were searched for the first non-threshold value: this point marked the front of the aircraft's radar signature. The same process was used to find the back of the signature. The range extent was then windowed to include only those range bins between the front and back of the aircraft signature (Figure 3-4). The target window size varied from about 10 to 200 range bins, depending on the aircraft and aspect angles. Since each range bin measured 0.15 meters across the aircraft, the target window size corresponded to the aircraft width as seen by the radar. The target window size (in range bins) was used as a feature value.

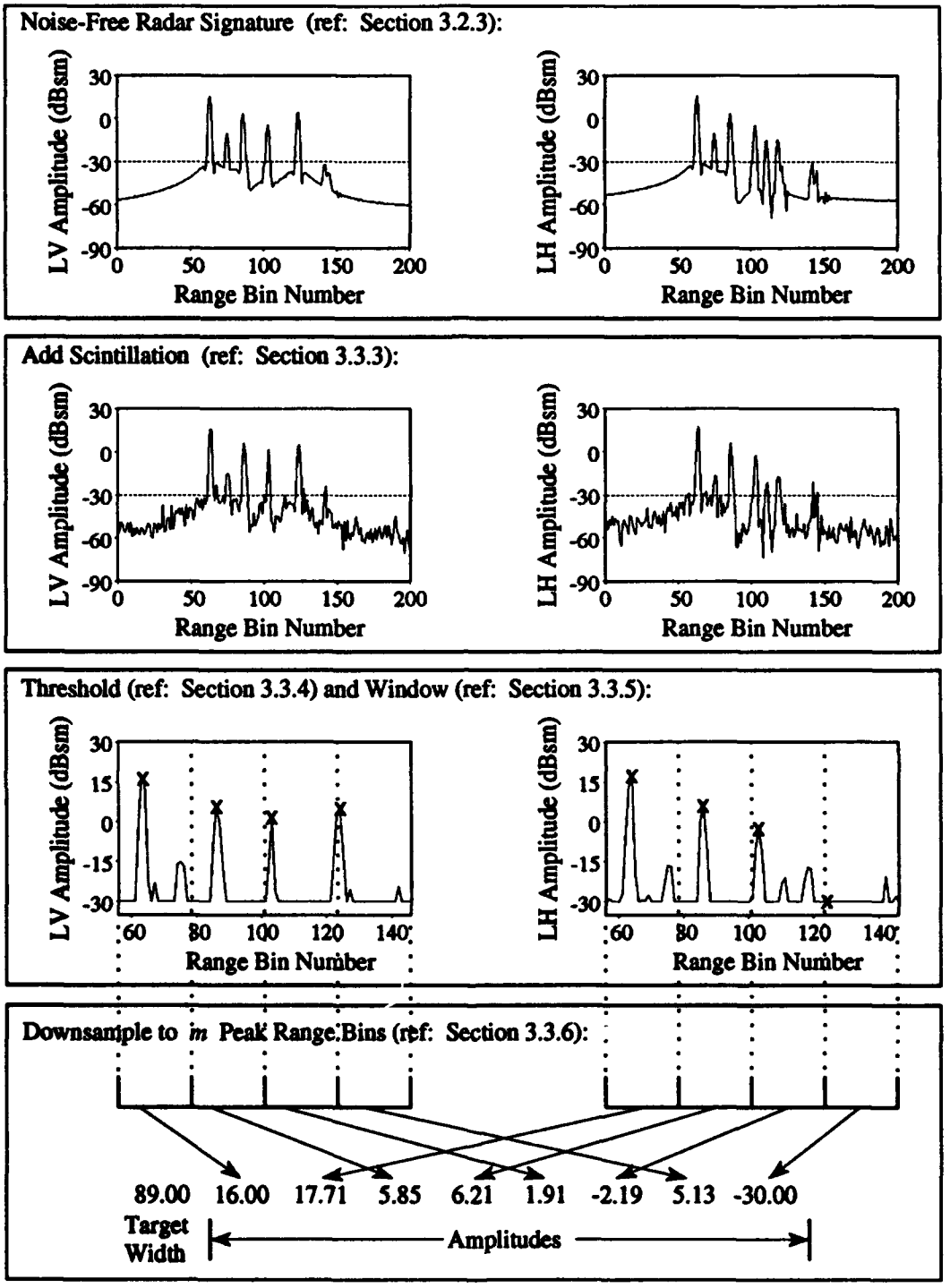


Figure 3-4: Example of the FORMAT_RCST feature extraction process for a single signature. This signature is from a MiG-21 at 0° elevation, 0° azimuth. Note the small peak in range bin 142: scintillation noise makes it visible above the thermal noise threshold.

3.3.6 Amplitude Downsampling. A recurrent neural network must have a fixed number of input feature values to describe each signature. Since the number of range bins in the target window was variable, a feature extraction routine was needed to select a fixed number of range bins for all signatures. This thesis used a peak extraction routine. Each range window was divided into a fixed number m intervals, with each interval being n range bins wide: n was constant for all intervals within the *same* signature, but varied *between* signatures. Within each interval, a max out of n downsampling routine selected the range bin with the highest LV and LH amplitude sum. These peak LV and LH amplitude values (from the same range bin) became feature values for that interval and were used as inputs to the recurrent neural network (Figure 3-4). For this thesis, the phrase "peak range bin" refers to the paired LV and LH amplitude values selected by this downsampling routine.

Amplitude downsampling was performed independently on each radar signature as it was placed into a sequence. The *location* of the peak range bins varied between signatures, but peak location was not used as an input feature value. For some radar signatures, m was larger than the number of range bins in the target window. When this happened, all of the available range bins were used directly as peak range bins. The extra feature values were then padded with noise threshold amplitude values.

3.3.7 Data Normalization. All of the feature values were statistically normalized before being written to an output data file. This allowed the recurrent neural network to properly learn the input data sets. Statistical data were calculated from the set of raw RCS_TOOLBOX signatures, before any noise effects were added. A mean and standard deviation were calculated separately for the azimuth, the target width (based on amplitude thresholding), and the amplitude values. For the amplitude statistics, all of the LV and LH amplitude values were considered together. Any amplitude less than or equal to the amplitude threshold limit was ignored: FORMAT_RCST computed the amplitude mean and standard deviation based only on the data points above the amplitude threshold.

3.4 *Decision Processing: Recurrent Neural Networks*

This section describes the RECURRENT computer program, which uses a real time recurrent learning (gradient descent) algorithm for radar target identification. For a diagram of a recurrent neural network, see Figure 3-5. The real time recurrent learning algorithm itself is described completely in the literature [Lindsey, 1991:12-17][Williams and Zipser, 1989:270-280]. Previous work at AFIT had already implemented the algorithm as the computer program, RECNET [Lindsey, 1991:12-17]. However, certain modifications were required to properly handle the radar target identification problem. For the RECURRENT source code listing, see Appendix C.

3.4.1 Modifications. The most important change called for breaking the training loop over all input vectors (in RECNET) into two loops: one for sequences and the other for signatures within a sequence. At the beginning of each sequence, the output values, the output derivatives, and the partial derivatives matrix (P) were all set to zero. The program was also changed to allow for batch mode learning after the presentation of each signature, sequence, or epoch. For this thesis, batch mode learning was done after each sequence. Finally, the hyperbolic tangent was used as the non-linearity squashing function instead of the standard sigmoid.

3.4.2 Output Classification. A recurrent neural network can only process sequences of data [Williams and Zipser, 1989:270-280]. For any given signature, there will always be a time delay between presenting a signature to the net and recording the output node classification values *affected by that signature.* For this thesis, the time delay was one time step: the minimum sequence length was two signatures.

In RECURRENT, additional subroutines were written to compute classification accuracies at each time step during a sequence. These subroutines checked to see if the output node values were "right" or "good". If all of the actual output node values were within 20% of the desired values, then the neural network classified the sequence (at that time step) as "right". For the hyperbolic tangent, desired outputs were +1.0 for the correct class and -1.0 for all other classes. If the output node with the highest actual

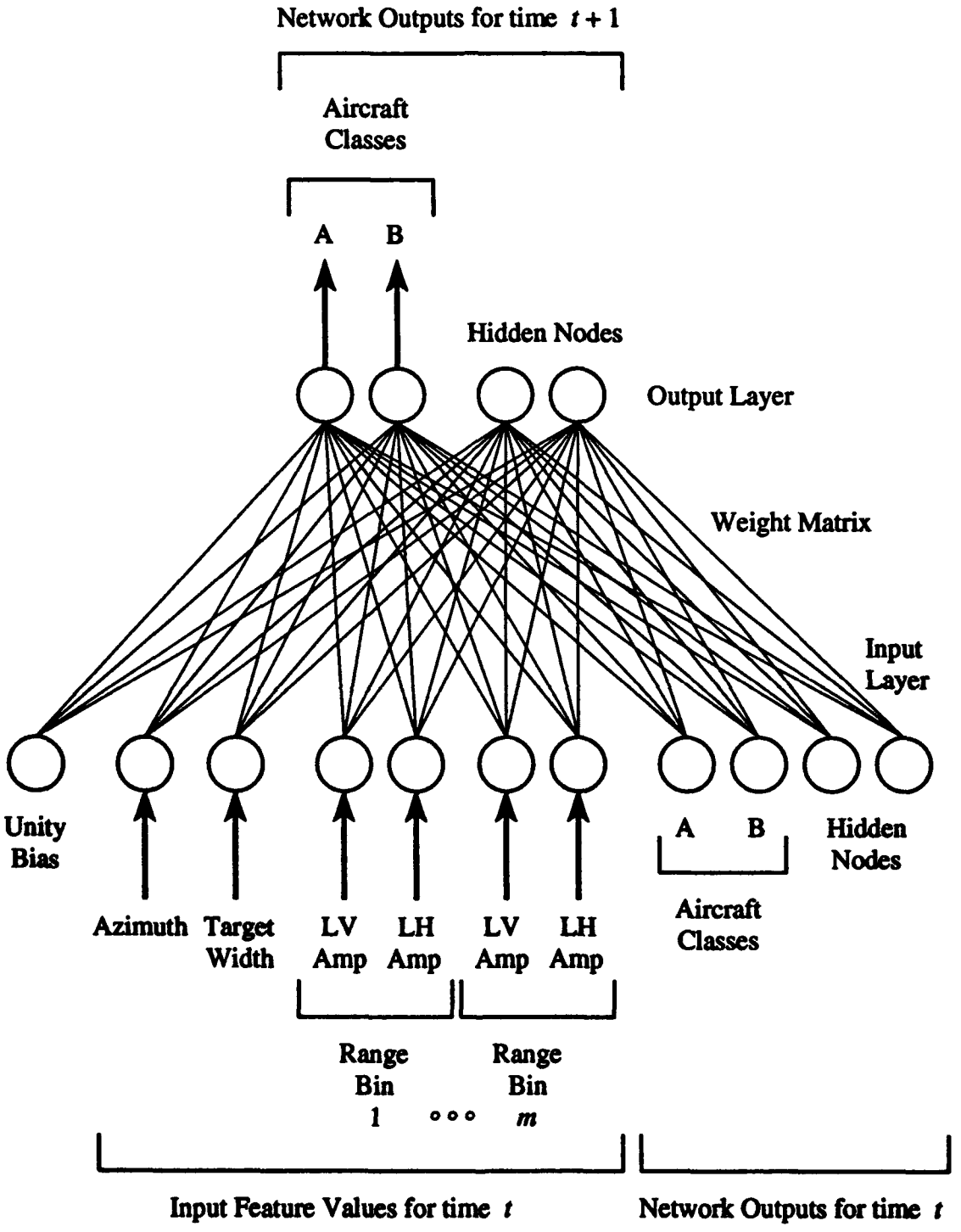


Figure 3-5: Recurrent Neural Network Diagram

output (through a max pick routine) corresponded to the desired output node class, then the neural network classified the sequence (at that time step) as "good". Note: all outputs that are "right" are also "good", but not vice versa. Output node values and their classification categories were recorded for each time step: these formed the basis for the epoch accuracy and sequence length statistics.

3.4.3 Epoch Accuracy. Errors measured over an entire epoch (training or testing) were expressed using the mean squared error per output node. This was done by measuring the cumulative difference between the desired and actual output values over all output nodes and signatures in an epoch, then dividing by the total number of output nodes and signatures. The real time recurrent learning algorithm seeks to minimize the mean squared error between the desired and actual output node values by using gradient descent. The gradient descent process does this by adjusting the interconnection weight values between the input and output layer nodes [Williams and Zipser, 1989:270-280].

The mean squared error was also used by RECURRENT to modify the learning rate (alpha) during training. After the first five training epochs, the learning rate was cut in half if error increased by more than 1%. This is based on the RECNET variable learning rate algorithm [Lindsey, 1991:16-17].

3.4.4 Sequence Length Analysis. For this thesis, the recurrent neural network classified a target at each time step in a sequence except the first. This capability was used to help find the optimum sequence length for radar target identification under specific test conditions. If the classification accuracy at a given time step $t + 1$ (which is computed after t input signatures have been processed) is summed over all test sequences, then the resulting statistic is the accuracy of the network if all of the test sequences contained exactly t signatures. This assumes that each *sequence* is classified only by the output node values at the end of the sequence. If this summation is computed for each time step starting at $t = 2$ signatures, then the number of signatures required to produce unique sequences can be found (ref: Section 1.1.7).

3.5 Test Cases

The recurrent neural network algorithm was tested for radar target identification performance in three steps. All of the topics discussed in Sections 3.2 and 3.3 were applied, with specific modifications as listed under each test case.

3.5.1 Case One: Two Aircraft, No Noise. This test case was investigated to prove the feasibility of using recurrent neural networks for radar target identification. It was chosen for conceptual simplicity and computational speed. The following conditions were used:

- 2 aircraft classes: MiG-21 (class 1), F-4 (class 2)
- 0° elevation
- 0° to 180° azimuth
- Perfect knowledge of aircraft azimuth
- No scintillation noise

3.5.2 Case Two: Five Aircraft, No Noise. This test case was investigated to find a suitably confused region in the radar *signature* feature space. It was chosen so the advantages of *sequence* analysis could be clearly demonstrated. The following conditions were used:

- 5 aircraft classes: MiG-21 (class 1), F-4 (class 2), F-15 (class 3), F-16 (class 4), F-18 (class 5)
- 0° elevation
- 0° to 180° azimuth in 30° sectors: train six separate nets, one for each sector.
- Perfect knowledge of aircraft azimuth
- No scintillation noise
- A minimum number of input features to describe each radar signature.

3.5.3 Case Three: Five Aircraft, With Noise. This test case was investigated to demonstrate that recurrent neural networks can be used for radar target identification under realistic operational conditions. The following conditions were used:

- 5 aircraft classes: MiG-21 (class 1), F-4 (class 2), F-15 (class 3), F-16 (class 4), F-18 (class 5)
- +5° to -5° elevation
- 60° to 90° azimuth (a sector from Case Two)
- Azimuth uncertainty included
- Scintillation noise included

3.6 Summary

This chapter described a methodology to: (1) generate high range resolution radar signatures, (2) process these signatures into sequences for recurrent neural network training, and (3) train and test a recurrent neural network for radar target identification. The test case results are given in Chapter IV.

IV. Results

4.1 Introduction

This chapter contains results from the three radar target identification test cases described in Section 3.5. For all test cases, training and test data files were generated from different aspect angles (within the same region). The recurrent neural net was able to generalize from discrete training data at 1° aspect angle granularity, However, recall that within any given sequence, elevation was constant and azimuth increased by 1° per time step.

Case One considered two aircraft classes, a single elevation, and no noise. This test case was investigated to show that a recurrent neural network can process sequences of radar signatures and identify aircraft classes. Each radar signature was described by 22 external input features: the azimuth, target width, and 20 amplitude values from 10 peak range bins. Four consecutive radar signatures were sufficient to reach a test set accuracy of 99% "right" and 99% "good" (ref: Section 3.4.2).

Case Two considered five aircraft classes, a single elevation, and no noise. In this test case, the recurrent net was deliberately given minimal information about each radar signature. Over most of the aircraft aspect angles, radar signatures were described by 4 external input features: the azimuth, target width, and 2 amplitude values from a single peak range bin. Nine consecutive radar signatures were sufficient to reach a test set accuracy of 84% to 99% right and 99% "good". One aspect angle region was more difficult for target recognition (Sector 3, from 60° to 90° azimuth), where each radar signature had to be described by 6 input features: the azimuth, target width, and 4 amplitude values from 2 peak range bins. For these aspect angles, 5 consecutive radar signatures were sufficient to reach a test set accuracy of 99% "right" and 99% "good".

Case Three considered five aircraft classes, a ± 5° elevation range, and noise. Each radar signature was described by 6 external input features: the estimated target azimuth, estimated target width, and 4 noisy amplitude values from 2 peak range bins. Elevation was not used as an input feature. Nine consecutive radar signatures were sufficient to achieve a test set accuracy of 91% "right" and 96% "good".

4.2 *Case One: Two Aircraft, No Noise.*

This test case proves the feasibility of using recurrent neural networks for radar target identification. It was chosen for conceptual simplicity and computational speed. It is also similar to the experiment conducted by Farhat [Farhat, 1989:670-680]. The following recurrent neural network configuration was used:

- 1 bias input
- 1 external input (at time *t*) for azimuth
- 1 external input (at time *t*) for target width (ref: Section 3.3.5)
- 20 external inputs (at time *t*) for radar amplitude values, containing the LV and LH amplitudes for 10 peak range bins from each signature (ref: Section 3.3.6)
- 2 output nodes (at time $t + 1$): MiG-21 (class 1), F-4 (class 2)
- no hidden nodes

There were a total of 50 adjustable weights in the weight matrix. Three sets of data files were used to train and test the network. Training data files contained signatures from 0° elevation at the integer azimuths from 0° to 180° (ref: the primary aspect angles from Section 3.2.3). Each training file had a total of 344 sequences (representing all possible 10° sequences from 0° to 180° azimuth at a single elevation). Test data files contained signatures from 0° elevation at the half angle azimuths from 0.5° to 179.5° (ref: the offset aspect angles from Section 3.2.3). Each test file had a total of 342 sequences.

The network was trained for 100 epochs using an initial learning rate of 0.01. During training, the learning rate generally decreased to 0.005 (ref: Section 3.4.3). After training, the mean squared error per output node (versus the independent test sets) was approximately 0.02. For test set accuracy, the net achieved 92% "right" and 98% "good" after 2 signatures and 99%+ "right" and "good" after about 4 signatures. This means the MiG-21 and F-4 can be identified under the Case One conditions, where each radar signature is represented by 22 external input features. Radar signatures must be measured over 4° of azimuth to identify the aircraft classes.

These results are easy to explain: target regions within the *signature* feature space were easily distinguished. Figure 4-1 shows a plot of target width as a function of azimuth angle. Figure 4-1 clearly shows distinct regions for each target class, even without extra information from the radar signature amplitudes. The azimuth and target width features alone should be sufficient to identify the target classes. In this feature space, almost any decision processing algorithm will achieve excellent results. The recurrent neural network was able to process temporal sequences of radar signatures and identify targets for the Case One conditions. This demonstrated the feasibility of using recurrent neural networks for radar target identification.

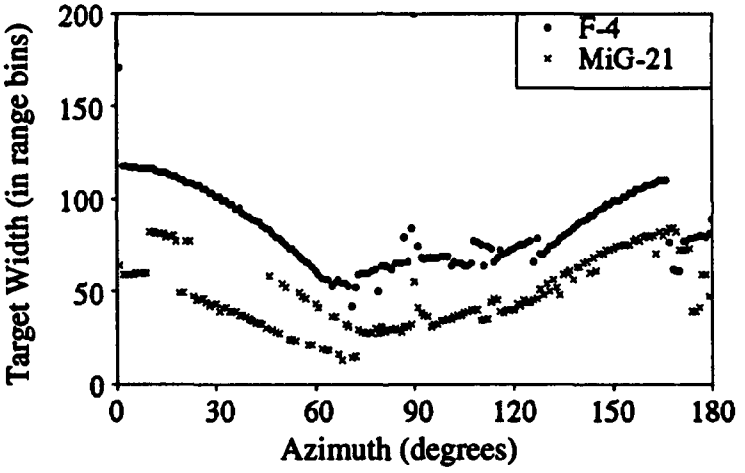


Figure 4-1: Azimuth versus target width for the Case One feature space. As the aircraft azimuth varies, the radar measures different target widths. These two features separate the MiG-21 and F-4 classes, except for a small region near 170° azimuth. The overall (sinusoidal) shapes are due to physical geometry: both aircraft are longer than they are wide. The discontinuities are due to low amplitude peaks alternately appearing and disappearing near the -30 dBsm thermal noise threshold.

4.3 Case Two: Five Aircraft, No Noise.

This test case demonstrates the advantages of using a *sequence* feature space for decision processing. For this problem, the number of external input features describing each radar signature was deliberately reduced. This was done to force target regions in

the *signature* feature space to overlap. The recurrent net was able to process this reduced amount of data as sequences and still identify the aircraft classes.

The 0° to 180° azimuth range was arbitrarily divided into six sectors. This was done to speed up network training (with smaller data files) and to find the azimuth region that was the most difficult for radar target identification. The sectors were numbered from 1 to 6 with sector 1 ranging from 0° to 30° azimuth. Each sector was used to train separate networks. Case Two results are presented in two parts. The first part deals with sectors 1, 2, 4, 5, and 6: these sectors all had similar results. The second part deals with sector 3 (between 60° and 90° azimuth): sequences in this region were more difficult to classify. Sector 3 was the most difficult to solve under Case Two conditions, so it was analyzed further in Case Three.

4.3.1 Case Two: Sectors 1, 2, 4, 5, and 6

The following recurrent neural network configurations were used:

- 1 bias input
- 1 external input (at time t) for azimuth
- 1 external input (at time t) for target width (ref: Section 3.3.5)
- 2 external inputs (at time t) for radar amplitude values, containing the LV and LH amplitudes for a single peak range bin from each signature (ref: Section 3.3.6)
- 5 output nodes (at time $t + 1$): MiG-21 (class 1), F-4 (class 2), F-15 (class 3), F-16 (class 4), F-18 (class 5)
- no hidden nodes for sectors 1, 2, 5, and 6: 50 adjustable weights in these weight matrices
- 3 hidden nodes for sector 4: 104 adjustable weights in this weight matrix

For each sector, two sets of data files were used to train and test the network. Training data files contained signatures from 0° elevation at the integer azimuths (ref: the primary aspect angles from Section 3.2.3). Each training file had a total of 110

sequences (representing all possible 10° sequences from a 30° azimuth region, single elevation, five targets). Test data files contained signatures from 0° elevation at the half angle azimuths (ref: the offset aspect angles from Section 3.2.3). Each test file had a total of 105 sequences.

Sectors 1, 2, 5, and 6 were trained for 100 epochs using an initial learning rate of 0.001. During training, the learning rate generally decreased to 0.0025. After training, the mean squared error per output node (versus the independent test sets) varied from about 0.1 to 0.35. Sector 4 was a bit more difficult to train, requiring 500 epochs. During training, the sector 4 learning rate generally decreased to 0.00125. After training, the sector 4 means squared error per output node was about 0.35. Figure 4-2 shows the recognition accuracy plots against test data sets for sector one. Sectors 2, 4, 5, and 6 had similar accuracy plots.

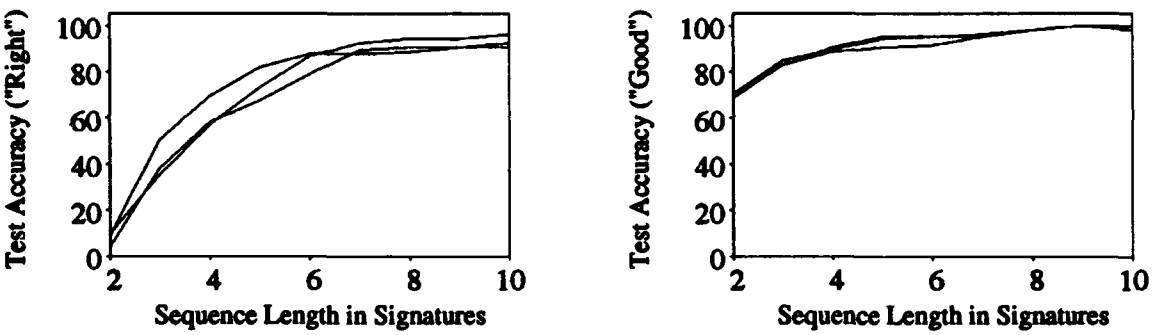


Figure 4-2: Plots of sequence length versus test set recognition accuracy for Case Two, sector one. Results are given from three separate weight matrices (hence the three curves on each plot) created by training the recurrent net three times. Each weight matrix was tested for accuracy using both the "right" and "good" criteria. As the number of signatures in a sequence increases, the sequences becomes more and more unique.

The recognition accuracies shown in Figure 4-2 are based on classifying a sequence using only the outputs from the last signature in the sequence. Since each successive output value is dependent on all previous values, these plots measure the information content in a given sequence length. For a two signature sequence, the test set classification accuracies are about 10% right and 70% good. As the sequence length

increases, recognition accuracy also increases. For sectors 1, 2, 4, 5, and 6, the recurrent neural network achieves a test set accuracy of 84% to 99% "right" and 99%+ "good" accuracy after about 9 signatures. This means the five aircraft classes can be identified (for most aspect angles) under the Case Two conditions, where each radar signature is represented by 4 external input features. Radar signatures must be measured over 9° of azimuth to identify the aircraft classes.

4.3.2 Case Two: Sector 3

The recurrent neural network configuration was the same as for sectors 1, 2, 5, and 6, except that an additional 2 external inputs were used for radar amplitude values (using two peak range bins rather than just one). Recall that only the peak amplitude values were used as input features.. The locations of these peaks within the radar signatures were not fed into the net (ref: Section 3.3.6). There were no hidden nodes, so the weight matrix contained a total of 60 adjustable weights. Training and test data files were generated using the same procedures as for the other sectors.

The network was trained for 500 epochs using an initial learning rate of 0.001. During training, the learning rate generally stayed constant. After training, the mean squared error per output node (versus the independent test sets) was approximately 0.055. Figure 4-3 shows the recognition accuracy plots against test data sets for sector

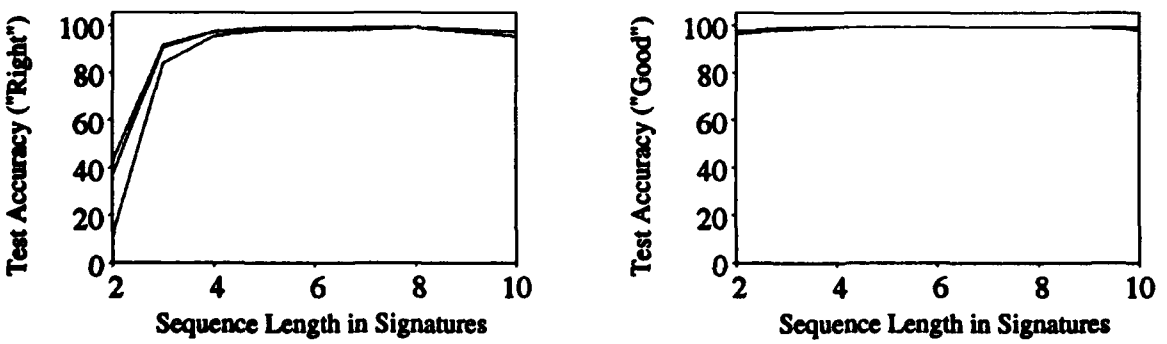


Figure 4-3: Plots of sequence length versus test set recognition accuracy for Case Two, sector three. These plots are similar to those in Figure 4-2, except the results are based on 6 external input features to describe each radar signature. Six feature values were required because 4 features could provide only 90% "good" accuracy.

three. For a two signature sequence, the test set classification accuracies are about 33% right and 97% good. The recurrent neural network achieves 99%+ "right" and "good" accuracy (versus the independent test sets) after about 5 signatures. This means the five aircraft classes can be identified under Case Two conditions, where each radar signature is represented by 6 external input features. Radar signatures must be measured over 5° of azimuth to identify the aircraft classes.

4.3.3 Case Two: Summary. This test case demonstrated the advantages of using a *sequence* feature space for decision processing. As the sequence length increased, recognition accuracy also increased. Since sector 3 was the most difficult to solve under Case Two conditions, it was analyzed further in Case Three.

4.4 Case Three: Five Aircraft, With Noise.

This test case was investigated to demonstrate that recurrent neural networks can be used for radar target identification under realistic operational conditions. The following conditions were used:

- 1 bias input
- 1 external input (at time *t*) for the estimated azimuth value (bias standard deviation of 2° and random fluctuations standard deviation of 0.2°)
- 1 external input (at time *t*) for the estimated target width (ref: Section 3.3.5)
- A variable number of radar amplitude values, starting off with the LV and LH amplitudes from a single peak range bin (ref: Section 3.3.6). Additional amplitude features were added one range bin at a time.
- scintillation noise (standard deviation of 5 dBsm) added to the amplitude values
- 5 output nodes: MiG-21 (class 1), F-4 (class 2), F-15 (class 3), F-16 (class 4), F-18 (class 5)
- no hidden nodes

The network started out with a total of 50 adjustable weights in the weight matrix. For each additional peak range bin, the number of weights increased by 10. Data sets were created by extending sector three over a ±5° elevation range. To adequately represent the distribution of noisy input values, each sequence was placed in the data files twice. Noise effects were computed each time a signature was placed into a sequence, so aspect angles between 69° and 81° azimuth were represented by a total of 20 independent noise realizations. This applied to training and test data files. Training data files contained signatures from the primary aspect angles (ref: Section 3.2.3). Each training file had a total of 2420 sequences (representing all possible 10° sequences out of a 30° azimuth region, 11 elevations, five targets, each sequence used twice). Test data files contained signatures from the half-angle elevations (ref: the offset aspect angles from Section 3.2.3). Each test file had a total of 2100 sequences.

4.4.1 Case Three: One Peak Range Bin. The network was trained for 100 epochs using an initial learning rate of 0.0001. During training, the learning rate generally decreased to 0.00005. After training, the mean squared error per output node (versus the independent test sets) was approximately 0.65. Plots of test set recognition accuracy versus sequence length are shown in Figure 4-4. Accuracy reaches a maximum of about 55% right (with 8 signatures) and about 76% good (with 7 signatures). These results mean that a single peak range bin cannot provide enough information to produce unique signatures.

4.4.2 Case Three: Two Peak Range Bins. The network was trained for 100 epochs using an initial learning rate of 0.0001. During training, the learning rate generally decreased to 0.000025. After training, the mean squared error per output node (versus the independent test sets) was approximately 0.28. Plots of recognition accuracy versus sequence length are shown in Figure 4-4. Test set accuracy reaches a maximum of about 90% right (with 9 signatures) and about 96% good (with 8 signatures). These results are remarkable, considering that target classifications are made using only six input features and simple vector*matrix multiplications. Two peak range bins provided almost enough information to produce unique signatures.

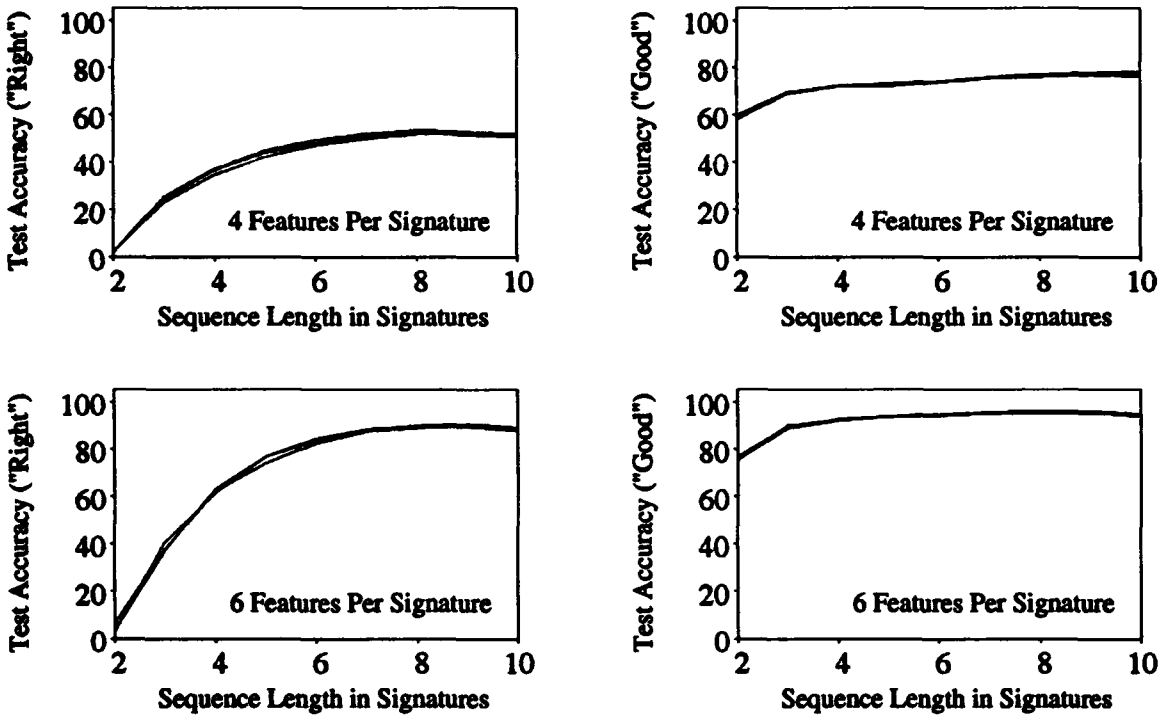


Figure 4-4: These plots show the effect of additional input feature values on test set accuracy for Case Three. The top pair of curves use 4 input features: the azimuth estimate, the target width estimate, and two noisy amplitude values from a single peak range bin. The bottom pair of curves use amplitudes from an additional peak range bin. For both sets of curves, test set accuracy increases as longer sequences are processed.

4.5 Summary

This chapter presented results from three separate test cases. Case One demonstrated the feasibility of using recurrent neural networks for radar target identification. Case Two showed that there is a relationship between the number of signatures in a sequence and the test set accuracy. If enough signatures are included in a sequence, the sequence will become unique. Case Three showed that a recurrent neural network can process sequences of noisy data and still achieve very high test set accuracies.

V. Conclusions

5.1 Introduction

The goal of this thesis was to identify five aircraft classes using sequences of high range resolution radar signatures. Radar target identification is an important part of the aircraft surveillance mission. Identification was performed using a recurrent neural network, which implemented a real time recurrent learning algorithm. Recurrent neural networks were trained and tested against three successively more difficult radar target identification problems. Chapter IV contains the results for each of these test cases.

5.2 Conclusions

The first test case (two aircraft classes, 0° elevation, no noise) demonstrated the feasibility of using recurrent neural networks for radar target identification. Each radar signature was described by 22 external input features: the azimuth, target width, and 20 amplitude values from 10 peak range bins. Four consecutive radar signatures were sufficient to reach a test set accuracy of 99% "right" and 99% "good". For Case One, target regions within the signature feature space were easily distinguished.

The second test case (five aircraft classes, 0° elevation, no noise) demonstrated the advantages of using sequence analysis to identify targets. The amount of information in each radar signature was arbitrarily reduced in an attempt to force the different target classes into the same regions of the signature feature space. For most aspect angles, each radar signature was described by 4 external input features: the azimuth, target width, and 2 amplitude values from a single peak range bin. Nine consecutive radar signatures were sufficient to reach a test set accuracy of 84% to 99% "right " and 99% "good". Sector 3 (60° to 90° azimuth) was more difficult for radar target identification. Each radar signature in sector 3 had to be described by 6 input features: the azimuth, target width, and 4 amplitude values from 2 peak range bins. Five consecutive radar signatures were sufficient to reach a test set accuracy of 99% "right" and 99% "good". Case Two showed that recognition accuracy increases as the

sequence length increases. For cases when the radar signatures are not unique, sequences can be used to help increase recognition accuracy.

The third test case (five aircraft classes, +5° to -5° elevation range, with noise) demonstrated that a recurrent neural network can identify targets under realistic operational conditions. Each radar signature was described by 6 external input features: the estimated target azimuth, the estimated target width, and 4 noisy amplitude values from 2 peak range bins. Nine consecutive radar signatures were sufficient to achieve a test set accuracy of 91% "right" and 96% "good".

These results bring up several interesting conclusions:

- There is a significant amount of information contained in the temporal sequences of radar signatures. Sequence processing increases target recognition accuracy.

- Aspect angle and target width estimates make good features for radar target identification. For simple problems, these two features may be sufficient to identify the target classes. When noisy radar data is used, these features can be processed as sequences: noisy features or not, they contribute to recognition accuracy.

- A wideband radar with a sufficient range resolution is required to measure the target width. Once the target width is measured, most of the information (amplitudes) in a high range resolution radar signature may be downselected and thrown out. Only a few of the dominant range bins need to be retained as additional feature inputs.

- For this thesis, a simple peak extraction algorithm was sufficient to identify the dominant range bins.

- As the radar signatures are described by more and more features, recognition accuracy increases. This means there is a tradeoff between using sequence processing with fewer features and "single-look" processing with more features. For a given radar target identification problem, these two approaches can be compared to find the optimal solution.

5.3 Recommendations for Future Research

Are there any other features that may be used to help identify radar targets? For example, temporal phase variations are used by synthetic and inverse synthetic aperture mode radar to produce a two-dimensional image of a target. These phase variations may be a feature value that can be used for aircraft target identification as well. Alternatively, a recurrent neural network might be able to identify ground targets by processing sequences of synthetic aperture mode radar signatures.

Can a recurrent neural network recognize the radar sequences from arbitrary aspect angle trajectories? This thesis assumed that targets followed a "turntable motion" type of trajectory. For real world radar applications, this type of motion is not often seen: there is an infinite number of potential aspect angle sequences. Can a recurrent network be trained (with a finite data set) to recognize an infinite number of sequences?

What kind of performance will a recurrent neural network achieve when the targets are just barely detectable above the thermal noise limit? This thesis did not address thermal noise effects: for long range aircraft surveillance systems, thermal noise will be a dominant factor. Can a recurrent neural network be used to locate and/or identify targets that are right at the detection threshold?

Appendix A. Software Development

Appendix B contains the source listing for FORMAT_RCST. This program reads in wideband radar signatures (RCS_TOOLBOX [Bramley and others, 1991: A, 11-13]) and formats them into sequences for input to RECURRENT. Appendix C contains the source listing for RECURRENT, which is based on the RECNET code [Lindsey, 1991:58-74]. RECURRENT implements a real time recurrent learning algorithm. Appendix D contains the source listing for UTILITIES: these subroutines are accessed by function calls from FORMAT_RCST and RECURRENT. All of these programs are written in "ANSI C" for use on a NeXTstation computer.

A.1 File Structure

FORMAT_RCST and RECURRENT are written to use the following file structure. All executable files are stored in a single (base) directory. Within the base directory, all of the RCS_TOOLBOX wideband radar signatures are stored in a single (signatures) subdirectory. This directory name must be written as the "path" variable in FORMAT_RCST. The signatures directory may be further divided as desired (sorted by aircraft, for example). These subdivisions must be written as filename extensions for the "jet" variables in FORMAT_RCST. Also within the base directory, a new subdirectory should be created for each test case. FORMAT_RCST writes data files to the case directory (overwriting existing data files that have the same filenames). RECURRENT uses the case directory to read input data and to write output data (again, overwriting existing output files with the same filenames).

A.2 FORMAT_RCST

Different versions of FORMAT_RCST must be compiled for each different test case. Test cases are described using the "#define" statements in FORMAT_RCST (ref: B-1, B-2). FORMAT_RCST may be run multiple times to create training data from one set of aspect angles and test data from a different set: this is done by commenting out sections of MAIN within FORMAT_RCST (ref: B-5). FORMAT_RCST also

produces "setup" files that control the RECURRENT neural network architecture (ref: B-12). FORMAT_RCST is invoked by typing:

```
format_rcst case
```

where "case" is an existing directory. The following data and setup files are written to the "case" directory (caution: existing files with the same names are overwritten):

feature.statistics	computed values from training data
test_feature.statistics	echoed values from FORMAT_RCST, created only if the "alternate_statistics_flag" is set (ref: B-2).
undetected.aircraft	aspect angles where an aircraft has fewer range bins after windowing than the desired number of peak range bins. Bug: the minimum target width is 1 range bin, not 0.
mig21.rcst_data	diagnostics file, typically commented out
clean_list.unnormalized	training feature values with no noise
train.setup	RECURRENT setup file
train	RECURRENT training data, with sequences presented in random order
train.unnormalized	typically commented out (ref: B-13)
check.setup	RECURRENT setup file
check	RECURRENT training data, with sequences listed in a defined order
check.unnormalized	typically commented out (ref: B-13)
test.setup	RECURRENT setup file
test	RECURRENT test data, with sequences listed in a defined order
test.unnormalized	typically commented out (ref: B-13)

A.3 RECURRENT

The recurrent neural network architecture is defined by the "train.setup" file. This defines the size of the weight matrix and most of the training parameters (epochs, alpha, etc). Setup files are used for all three data input files: this allows each file to have a different number of sequences and a different sequence length. Bug: the sequence length analysis subroutine assumes equal sequence lengths. RECURRENT is invoked (with a random initial weight matrix) by typing:

```
recurrent case
```

where "case" is an existing directory containing the FORMAT_RCST data files. RECURRENT may also be invoked (with a user-defined initial weight matrix) by typing:

```
recurrent case weightsfile
```

where "case" is an existing directory containing the FORMAT_RCST data files. and "weightsfile" is an existing file with a properly sized weight matrix. For both methods, the following data and setup files are written to the "case" directory (caution: existing files with the same names are overwritten):

weights.initial	randomly generated weight matrix
weights.final	weight matrix after training
weights.restart	echoes a user-defined initial weight matrix
epochs	mean squared error and accuracy results for each training epoch and for one test epoch. This file is formatted for NXYPLOT, a plotting program for NeXT computers.
length.check	accuracy versus sequence length (ref: Section 3.4.4) for "check" data, computed during the

	test epoch. This is formatted for NXYPLOT, a plotting program for NeXT computers.
`length.test`	accuracy versus sequence length (ref: Section 3.4.4) for "`test`" data, computed during the test epoch. This is formatted for NXYPLOT, a plotting program for NeXT computers.
`signatures.check`	actual and desired output node values for the "`check`" data (signature by signature), computed during the test epoch
`signatures.test`	actual and desired output node values for the "`test`" data (signature by signature), computed during the test epoch
`sequences.check`	diagnostics on sequence classification (not very useful), computed during the test epoch
`sequences.test`	diagnostics on sequence classification (not very useful), computed during the test epoch

Appendix B. FORMAT_RCST Source Code

This appendix contains the FORMAT_RCST source code. The various functions of FORMAT_RCST are described in Section 3.3.

```
/*  format_rcst.c  ************************************************************

This program reads *.wbtable* files created by the program PTDTABLE and
formats the data for input to a recurrent neural network.  The *.wbtable*
data format is described in the RCS_TOOLBOX User's Guide [Bramley and others,
1991:A, 11-13].  PTDTABLE operations are also described in the RCS_TOOLBOX
User's Guide [Bramley and others, 1991:Ch 9, 1-3].  The program RECURRENT
(described in Appendix C) implements the recurrent neural network.

Each line in the output files have the following format:
{ [elevation tag], [azimuth tag], [azimuth feature value], [target width feature value],
[LV amplitude 1], [LH amplitude 1], [LV amplitude 2], [LH amplitude 2]... (etc, for all
desired range bins describing each signature), [class 1 output node desired value],
[class 2 output node desired value]... (etc, for all target classes) }

Multiple lines are listed to complete each sequence.  Multiple sequences are
listed to complete the data file.

Written Fall 1992 by Eric T. Kouba

*****************************************************************************/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>


char *path = { "rcst_signatures/" };     /* Directory containing all input files. */
char *directory;                         /* Target directory to place output files.
                                            The target directory must already exist.
                                            Any existing files with the same names as
                                            the output files will be overwritten. */

#define recursion              2         /* The number of times to pass through the
                                            data creation loops.  Useful for getting
                                            multiple examples of noisy data.  */

#define aircraft               5         /* Number of target classes, with the
                                            subdirectory and file name prefixes
                                            within the *path directory, above.  */
char *jet[aircraft] = { "mig21/mig21_", "f4/f4_", "f15/f15_", "f16/f16_", "f18/f18_" };


#define rcst_azimuth_begin     60        /* .wbtable signature index to begin and */
#define rcst_azimuth_end       90        /* end reading.  This index corresponds to
                                            the target azimuth. */

int rcst_azimuth = rcst_azimuth_end - rcst_azimuth_begin + 1;
                                         /* Signatures per target per elevation. */

#define rcst_elevation         11        /* Number of .wbtable files per target class,
```

```
                                        with each file containing data from
                                        different elevations. */

char *rcst_file[rcst_elevation] = { "el50_", "el40_", "el30_", "el20_", "el10_", "el00_",
                                   "el_10_", "el_20_", "el_30_", "el_40_", "el_50_" };

                                    /* Example .wbtable path and filename:
                                       "rcst_signatures/mig21/mig21_el25_lv.wbtable"
                                       containing the LV polarized data for a MiG-21
                                       at 2.5 degrees elevation. */

#define azimuth_offset_sigma   2.0      /* Bias standard deviation in degrees between
                                           truth and estimated azimuth */
#define azimuth_sigma          0.2      /* Standard deviation for random azimuth error */
#define amplitude_sigma        5.0      /* Scintillation noisestandard deviation */
#define amplitude_threshold  -30.0      /* Thermal noise threshold */
#define range_bins           200        /* .wbtable range bins per radar signature */
#define feature_bins           1        /* Desired number of output range bins
                                           per target (feature range bins) */
#define sigs_per_seq          10        /* Signatures per sequence */
#define node_on                1.0      /* tanh values for the recurrent net */
#define node_off              -1.0

#define alternate_statistics_flag     0     /* 0 to use calculated values */
#define input_az_mean         75.000000     /* FORMAT_RCST computes data statistics */
#define input_az_sigma         8.973265     /* for normalization.  External values  */
#define input_width_mean      42.854546     /* can be used instead by setting the   */
#define input_width_sigma     18.506121     /* flag = 1.  Useful for creating test  */
#define input_amp_mean       -14.737031     /* data. */
#define input_amp_sigma       12.598812


FILE *Open_File2();          /* Subroutines from UTILITIES.C. */
FILE *Open_File4();
int Get_Integer();
float Uniform_Random();      /* RAN1.C from Numerical Recipies in C. */
int integer_seed;
float Gaussian_Random();     /* GASRAN.C from Numerical Recipies in C. */
float *Vector();
int **Integer_Matrix();
float ***Matrix_3D();


FILE *LV;                /* LV polarization, input data */
FILE *LH;                /* LH polarization, input data */
FILE *RCST_DATA;         /* Diagnostic listing of the raw .wbtable data. */
FILE *STATISTICS;        /* Output listing of data statistics. */
FILE *UNDETECTED;        /* Diagnostic listing of aircraft and aspect angles that
                            are too narrow for the desired number of range bins. */
FILE *FEATURE_DATA;      /* Unnormalized output data */
FILE *RECURRENT_DATA;    /* Normalized output data */
FILE *RECURRENT_SETUP;   /* Setup file for output data */


int seqs_per_el;         /* Sequences per elevation */
int total_signatures;
int total_sequences;

struct {
   float ***aspect;      /* elevation and azimuth */
   float ***input;       /* combined LV and LH data */
   int **flag;           /* sequence marker for random ordering */
} data[aircraft];
```

```
float az_sum;            /* for statistics */
float az_square;
float az_mean;
float az_sigma;
float az_min;
float az_max;
float az_n;

float width_sum;
float width_square;
float width_mean;
float width_sigma;
float width_min;
float width_max;
float width_n;

float amp_sum;
float amp_square;
float amp_mean;
float amp_sigma;
float amp_min;
float amp_max;
float amp_n;

int window_start;        /* First and last range bins with a  */
int window_end;          /* non-threshold amplitude value.    */
int target_region;       /* Apparent target width. */

int interval_start;      /* These are used to divide the target region */
float interval_width;    /* of interest into "n" intervals, where "n"  */
int interval_end;        /* is the desired number of range bins to use */
float max_total_amp;     /* as feature values. */
int tagged_range_bin;

float *signature_input;  /* Pointers for line by line data processing. */
float *noise_input;
float *signature_output;
float signature_lv;      /* Input data value. */
float signature_lh;
float bin_lv;            /* Output data value. */
float bin_lh;

float azimuth_offset;
float azimuth_noise;
float azimuth_estimate;
float amplitude_noise;


char ignore[80];
float trash;
int a;                    /* aircraft counter */
int b;                    /* elevation file counter */
int c;                    /* signatures per elevation counter */
int d;                    /* sequences per elevation counter */
int e;                    /* signatures per sequence counter */
int f;                    /* total sequences counter */
int g;                    /* file recursion counter */
int i, j;                 /* generic counters */


void Initialize();        /* Subroutines */
void Read_Signatures();
void Print_RCST_Data();
void Set_Statistics();
```

```
void Compute_Statistics();
void Extract_Width_Feature();
void List_Clean_Sequences();
void Extract_Clean_Features();
void Downsample();
void Print_Signature();
void Print_Recurrent_Setup();
void Random_Sequences();
void List_Noisy_Sequences();
void Extract_Noisy_Features();
void Add_Noise();
void Print_Noisy_Signature();



/* main program for FORMAT_RCST ****************************************/

void main( int argc, char *argv[] )
{
   if( argc != 2 )
      Exit_Message( "Usage:  format_rcst <directory>" );
   directory = argv[1];

   srand( time( NULL ));         /* Use the cheap Unix random number generator  */
   integer_seed = -rand();       /* to seed the high class "Numerical Recipies" */
   Uniform_Random( &integer_seed );                              /* generator. */

   Initialize();


   for( a = 0; a < aircraft; a++ )                  /* Read all .wbtable files. */
      for( b = 0; b < rcst_elevation; b++ )
      {
         LV = Open_File4( path, jet[a], rcst_file[b], "lv.wbtable", "r" );
         LH = Open_File4( path, jet[a], rcst_file[b], "lh.wbtable", "r" );
         Read_Signatures();
         fclose( LV );
         fclose( LH );
      }
                        /* Diagnostics print of raw .wbtable data for one aircraft. */
/*    RCST_DATA = Open_File2( directory, "/mig21.rcst_data", "w" );
      Print_RCST_Data( a = 0 );
      fclose( RCST_DATA );
*/

   if( alternate_statistics_flag == 1 )
   {                                                    /* Use defined statistics. */
      STATISTICS = Open_File2( directory, "/test_feature.statistics", "w" );
      Set_Statistics();
      fclose( STATISTICS );
   }
   else
   {                                                     /* Or computed values. */
      STATISTICS = Open_File2( directory, "/feature.statistics", "w" );
      Compute_Statistics();
      fclose( STATISTICS );
   }


   UNDETECTED = Open_File2( directory, "/undetected.aircraft", "w" );

   fprintf( UNDETECTED, "clean_list:\n" );
   FEATURE_DATA = Open_File2( directory, "/clean_list.unnormalized", "w" );
```

```
   List_Clean_Sequences();                                    /* No noise added. */
   fclose( FEATURE_DATA );


   RECURRENT_SETUP = Open_File2( directory, "/train.setup", "w" );
   Print_Recurrent_Setup();
   fclose( RECURRENT_SETUP );

   fprintf( UNDETECTED, "\ntrain:\n" );
   FEATURE_DATA = Open_File2( directory, "/train.unnormalized", "w" );
   RECURRENT_DATA = Open_File2( directory, "/train", "w" );
   for( g = 0; g < recursion; g++ )
   {
      integer_seed = -rand();                 /* Re-initialize the random seed. */
      Uniform_Random( &integer_seed );
      Random_Sequences();                     /* Random sequence order, with noise. */
   }
   fclose( FEATURE_DATA );
   fclose( RECURRENT_DATA );


   RECURRENT_SETUP = Open_File2( directory, "/check.setup", "w" );
   Print_Recurrent_Setup();
   fclose( RECURRENT_SETUP );

   fprintf( UNDETECTED, "\ncheck:\n" );
   FEATURE_DATA = Open_File2( directory, "/check.unnormalized", "w" );
   RECURRENT_DATA = Open_File2( directory, "/check", "w" );
   for( g = 0; g < recursion; g++ )
   {
      integer_seed = -rand();
      Uniform_Random( &integer_seed );
      List_Noisy_Sequences();
   }
   fclose( FEATURE_DATA );
   fclose( RECURRENT_DATA );


   RECURRENT_SETUP = Open_File2( directory, "/test.setup", "w" );
   Print_Recurrent_Setup();
   fclose( RECURRENT_SETUP );

   integer_seed = -rand();
   Uniform_Random( &integer_seed );

   fprintf( UNDETECTED, "\ntest:\n" );
   FEATURE_DATA = Open_File2( directory, "/test.unnormalized", "w" );
   RECURRENT_DATA = Open_File2( directory, "/test", "w" );
   for( g = 0; g < recursion; g++ )
   {
      integer_seed = -rand();
      Uniform_Random( &integer_seed );
      List_Noisy_Sequences();
   }
   fclose( FEATURE_DATA );
   fclose( RECURRENT_DATA );

   fclose( UNDETECTED );

}
/* end MAIN */
```

```
/* Subroutine to dynamically allocate memory ****************************/

void Initialize()
{
   seqs_per_el = rcst_azimuth - sigs_per_seq + 1;
   total_signatures = aircraft * rcst_elevation * rcst_azimuth;
   total_sequences = aircraft * rcst_elevation * seqs_per_el;

   for( a = 0; a < aircraft; a++ )
   {
      data[a].aspect = Matrix_3D( 0, rcst_elevation - 1, 0, rcst_azimuth - 1, 0, 1 );
      data[a].input = Matrix_3D( 0, rcst_elevation - 1, 0, rcst_azimuth -1,
                                                    0, ( range_bins * 2 ) - 1 );
      data[a].flag = Integer_Matrix( 0, rcst_elevation-1, 0, seqs_per_el-1 );

      noise_input = Vector( 0, (range_bins * 2)-1 );
      signature_output = Vector( 0, 2 + (feature_bins * 2)-1 );
   }

   return;
}
/* end INITIALIZE */



/* Subroutine to read one .wbtable file ****************************/

void Read_Signatures()
{
   for( i = 0; i < 10; i++ )
      fgets( ignore, 80, LV );
   fscanf( LV, "%f %f %f", &trash, &trash, &data[a].aspect[b][0][0] );
   for( i = 0; i < 10; i++ )                                    /* get the elevation */
      fgets( ignore, 80, LV );
   for( i = 0; i < 20; i++ )
      fgets( ignore, 80, LH );

   for( c = 0; c < rcst_azimuth_begin; c++ )
      for( i = 0; i < 1 + (range_bins * 2); i++ )
      {
         fscanf( LV, "%f", &trash );
         fscanf( LH, "%f", &trash );
      }

   for( c = 0; c < rcst_azimuth; c++ )
   {
      data[a].aspect[b][c][0] = data[a].aspect[b][0][0];          /* elevation */
      fscanf( LV, "%f", &data[a].aspect[b][c][1] );               /* azimuth */
      fscanf( LH, "%f", &trash );

      for( i = 0; i < range_bins; i++ )
      {
         fscanf( LV, "%f", &data[a].input[b][c][i * 2] );         /* LV amplitude */
         fscanf( LV, "%f", &trash );
         data[a].input[b][c][i * 2] = ( data[a].input[b][c][i * 2] / 100.0 ) - 200.0;

         fscanf( LH, "%f", &data[a].input[b][c][1 + (i * 2)] );   /* LH amplitude */
         fscanf( LH, "%f", &trash );
         data[a].input[b][c][1 + (i * 2)] = ( data[a].input[b][c][1 + (i * 2)] / 100.0 )
                                                                    - 200.0;
      }
   }
```

```
            /* Diagnostics print of the first few range bins from the first signature */
/* printf( "%6.1f  %6.1f  ", data[a].aspect[b][0][0], data[a].aspect[b][0][1] );
   for( i = 0; i < 4; i++ )
      printf( "%7.2f  ", data[a].input[b][0][i] );
   printf( "\n" );
*/
   return;
}
/* end READ_SIGNATURES */



/* Subroutine to print out the raw .wbtable data for one aircraft ******************/

void Print_RCST_Data( int a )
{
   for( b = 0; b < rcst_elevation; b++ )
      for( c = 0; c < rcst_azimuth; c++ )
      {
         fprintf( RCST_DATA, "%6.1f  ", data[a].aspect[b][c][0] );
         fprintf( RCST_DATA, "%6.1f  ", data[a].aspect[b][c][1] );
         for( i = 0; i < (range_bins * 2); i++ )
            fprintf( RCST_DATA, "%7.2f  ", data[a].input[b][c][i] );
         fprintf( RCST_DATA, "\n" );
      }

   return;
}
/* end PRINT_RCST_DATA */



/* Subroutine for using defined statistics to normalize the data ******************/

void Set_Statistics()
{
   az_mean = input_az_mean;
   az_sigma = input_az_sigma;
   width_mean = input_width_mean;
   width_sigma = input_width_sigma;
   amp_mean = input_amp_mean;
   amp_sigma = input_amp_sigma;

   fprintf( STATISTICS, "\n\tmean\t\tsigma\n" );
   fprintf( STATISTICS, "az\t%f\t%f\n", az_mean, az_sigma );
   fprintf( STATISTICS, "width\t%f\t%f\n", width_mean, width_sigma );
   fprintf( STATISTICS, "amp\t%f\t%f\n", amp_mean, amp_sigma );

   return;
}
/* end SET_STATISTICS */



/* Subroutine to compute statistical values from the input data ******************/

void Compute_Statistics()
{
   az_sum = 0.0;                        /* Initialize */
   az_square = 0.0;
   az_mean = 0.0;
```

```
az_sigma = 0.0;
az_min = 999.0;
az_max = -999.0;
az_n = (float)total_signatures;

width_sum = 0.0;
width_square = 0.0;
width_mean = 0.0;
width_sigma = 0.0;
width_min = 999.0;
width_max = -999.0;
width_n = (float)total_signatures;

amp_sum = 0.0;
amp_square = 0.0;
amp_mean = 0.0;
amp_sigma = 0.0;
amp_min = 999.0;
amp_max = -999.0;
amp_n = 0.0;            /* Amplitude statistics are computed only for values above
                           the thermal noise threshold */

for( a = 0; a < aircraft; a++ )
   for( b = 0; b < rcst_elevation; b++ )
      for( c = 0; c < rcst_azimuth; c++ )
      {
         az_sum += data[a].aspect[b][c][1];
         az_square += data[a].aspect[b][c][1] * data[a].aspect[b][c][1];
         if( data[a].aspect[b][c][1] < az_min )
            az_min = data[a].aspect[b][c][1];
         if(data[a].aspect[b][c][1] > az_max )
            az_max = data[a].aspect[b][c][1];

         Extract_Width_Feature();

         width_sum += (float)target_region;
         width_square += (float)( target_region * target_region );
         if( (float)target_region < width_min )
            width_min = (float)target_region;
         if( (float)target_region > width_max )
            width_max = (float)target_region;

         for( i = 2; i < (range_bins * 2); i++ )
            if( data[a].input[b][c][i] > amplitude_threshold )
            {
               amp_sum += data[a].input[b][c][i];
               amp_square += data[a].input[b][c][i] * data[a].input[b][c][i];
               if( data[a].input[b][c][i] < amp_min )
                  amp_min = data[a].input[b][c][i];
               if( data[a].input[b][c][i] > amp_max )
                  amp_max = data[a].input[b][c][i];
               amp_n += 1.0;
            }
      }

az_mean = az_sum / az_n;
az_sigma = sqrt(( az_n * az_square - az_sum * az_sum ) / ( az_n * ( az_n - 1 )));

width_mean = width_sum / width_n;
width_sigma = sqrt((width_n*width_square-width_sum*width_sum) / (width_n*(width_n-1)));

amp_mean = amp_sum / amp_n;
amp_sigma = sqrt(( amp_n*amp_square - amp_sum*amp_sum ) / ( amp_n * ( amp_n - 1 )));
```

```
   fprintf( STATISTICS, "\n\tsum\t\tsquare\t\tmean\t\tsigma\t\tmin\t\tmax\n" );
   fprintf( STATISTICS, "az\t%f\t%f\t%f\t", az_sum, az_square, az_mean );
   fprintf( STATISTICS, "%f\t%f\t%f\n", az_sigma, az_min, az_max );
   fprintf( STATISTICS, "width\t%f\t%f\t%f\t", width_sum, width_square, width_mean );
   fprintf( STATISTICS, "%f\t%f\t%f\n", width_sigma, width_min, width_max );
   fprintf( STATISTICS, "amp\t%f\t%f\t%f\t", amp_sum, amp_square, amp_mean );
   fprintf( STATISTICS, "%f\t%f\t%f\n", amp_sigma, amp_min, amp_max );

   return;
}
/* end COMPUTE_STATISTICS */



/* Subroutine to compute target width based on the clean input data ***************
   This subroutine called only to get statistical data. ***************************/

void Extract_Width_Feature()
{
   window_start = 0;
   window_end = range_bins - 1;

      /* Search for the first range bin that has a non-threshold LV or LH value */
   while( data[a].input[b][c][window_start * 2]       <= amplitude_threshold &&
          data[a].input[b][c][1 + (window_start * 2)] <= amplitude_threshold &&
          window_start < window_end )
      window_start++;

                                                  /* Also for the last range bin */
   while( data[a].input[b][c][window_end * 2]       <= amplitude_threshold &&
          data[a].input[b][c][1 + (window_end * 2)] <= amplitude_threshold &&
          window_start < window_end )
      window_end--;

   target_region = window_end - window_start + 1;

   return;
}
/* end EXTRACT_WIDTH_FEATURE */



/* Subroutine to print formatted, unnormalized output data, no noise ****************/

void List_Clean_Sequences()
{
   for( b = 0; b < rcst_elevation; b++ )
      for( d = 0; d < seqs_per_el; d++ )
         for( a = 0; a < aircraft; a++ )
            for( e = 0; e < sigs_per_seq; e++ )
            {
               c = d + e;
               Extract_Clean_Features();
               Print_Signature();
            }

   return;
}
/* end LIST_CLEAN_SEQUENCES */
```

```
/* Subroutine to extract formatted, unnormalized output data, no noise ****************/

void Extract_Clean_Features()
{
   signature_output[0] = data[a].aspect[b][c][1];

   window_start = 0;
   window_end = range_bins - 1;

   while( data[a].input[b][c][window_start * 2]       <= amplitude_threshold &&
          data[a].input[b][c][1 + (window_start * 2)] <= amplitude_threshold &&
          window_start < window_end )
      window_start++;

   while( data[a].input[b][c][window_end * 2]       <= amplitude_threshold &&
          data[a].input[b][c][1 + (window_end * 2)] <= amplitude_threshold &&
          window_start < window_end )
      window_end--;

   target_region = window_end - window_start + 1;      /* Get the target width */
   signature_output[1] = (float)target_region;         /* And downsample the range bins
                                                          to get the desired number    */
   Downsample( data[a].input[b][c] );                  /* of amplitude features. */

   return;
}
/* end EXTRACT_CLEAN_FEATURES */



/* Subroutine to perform a max out of n downsample.  Divides the target region of
   interest into m intervals of range bins where m is the desired number of range bins
   to pass as feature values.  The LV and LH amplitudes for each selected range bin
   are passed as feature values. ***************************************************/

void Downsample( float *signature_input )
{
   if( target_region <= feature_bins )    /* If the target has fewer range bins than  */
   {                                      /* the desired number of feature range bins */
      fprintf( UNDETECTED, "aircraft  %d,  elevation  %d,  azimuth  %d:   ", a, b, c );
      fprintf( UNDETECTED, "%d  range bins\n", target_region );

      for( i = 0; i < feature_bins; i++ )
      {
         if( i < target_region )          /* Use all range bins in the target window */
         {
            if( signature_input[(window_start + i) * 2] < amplitude_threshold )  /* LV */
               signature_output[2 + (i * 2)] = amplitude_threshold;
            else
               signature_output[2 + (i * 2)] = signature_input[(window_start + i) * 2];

            if( signature_input[1 + ((window_start + i) * 2)] < amplitude_threshold )
               signature_output[3 + (i * 2)] = amplitude_threshold;              /* LH */
            else
               signature_output[3 + (i * 2)] = signature_input[1 +
                                                   ((window_start + i) * 2)];
         }
         else                              /* and then pad with thermal noise values */
         {
            signature_output[2 + (i * 2)] = amplitude_threshold;
            signature_output[3 + (i * 2)] = amplitude_threshold;
         }
```

```
      }
   }

   else                              /* There are enough range bins in the target window */
   {
      interval_start = window_start;
      interval_width = (float)target_region / (float)feature_bins;
                                       /* Divide the target into intervals of
                                          range bins.  Select one range bin from
                                          each interval based on max pick. */
      for( i = 0; i < feature_bins; i++ )
      {
         interval_end = window_start + (int)( interval_width * (float)( i + 1 ));
         max_total_amp = 2.0 * amplitude_threshold;
         signature_lv = amplitude_threshold;
         signature_lh = amplitude_threshold;
         tagged_range_bin = interval_start;

         for( j = interval_start; j < interval_end; j++ )
         {
            if( signature_input[j * 2] < amplitude_threshold )            /* LV */
               bin_lv = amplitude_threshold;
            else
               bin_lv = signature_input[j * 2];

            if( signature_input[1 + (j * 2)] < amplitude_threshold )      /* LH */
               bin_lh = amplitude_threshold;
            else
               bin_lh = signature_input[1 + (j * 2)];

            if( bin_lv + bin_lh > max_total_amp )
            {
               max_total_amp = bin_lv + bin_lh;
               signature_lv = bin_lv;
               signature_lh = bin_lh;
               tagged_range_bin = j;
            }
         }

         signature_output[2 + (i * 2)] = signature_lv;
         signature_output[3 + (i * 2)] = signature_lh;

         interval_start = interval_end;
      }
   }

   return;
}
/* end DOWNSAMPLE */



/* Subroutine to print one line of formatted, unnormalized output data, no noise *****/

void Print_Signature()
{
   fprintf( FEATURE_DATA, "%6.1f  ", data[a].aspect[b][c][0] );     /* elevation */
   fprintf( FEATURE_DATA, "%6.1f  ", data[a].aspect[b][c][1] );     /* azimuth */

   for( i = 0; i < 2 + (feature_bins * 2); i++ )                    /* azimuth, width, */
      fprintf( FEATURE_DATA, "%7.2f  ", signature_output[i] );      /* and amplitudes */

   for( i = 0; i < aircraft; i++ )
```

```
   {
      if( i == a )
         fprintf( FEATURE_DATA, "%4.1f  ", node_on );
      else
         fprintf( FEATURE_DATA, "%4.1f  ", node_off );
   }

   fprintf( FEATURE_DATA, "\n" );

   return;
}
/* end PRINT_SIGNATURE */



/* Subroutine to print a setup file for the program RECURRENT ********************/

void Print_Recurrent_Setup( char *extension )
{
   fprintf( RECURRENT_SETUP, "%d      :input features >= 1\n", 2 + (feature_bins * 2) );
   fprintf( RECURRENT_SETUP, "%d      :output classes >= 1\n", aircraft );
   fprintf( RECURRENT_SETUP, "0       :hidden nodes >= 0\n" );
   fprintf( RECURRENT_SETUP, "%d      :total sequences >= 1\n", total_sequences *
                                                                recursion );
   fprintf( RECURRENT_SETUP, "%d      :signatures per sequence >= 2\n", sigs_per_seq );
   fprintf( RECURRENT_SETUP, "100      :training epochs >= 0\n" );
   fprintf( RECURRENT_SETUP, "0.0001      :learning step size alpha > 0.0\n" );
   fprintf( RECURRENT_SETUP, "3      :sigmoid flag, 1 linear, 2 exp sigmoid, " );
   fprintf( RECURRENT_SETUP, "3 tanh sigmoid\n" );
   fprintf( RECURRENT_SETUP, "2      :batch learn flag, 1 frame, 2 sequence, 3 epoch\n" );

   return;
}
/* end PRINT_RECURRENT_SETUP */



/* Subroutine to print noisy sequences in a random order (training data) ***********/

void Random_Sequences()
{
   for( a = 0; a < aircraft; a++ )
      for( b = 0; b < rcst_elevation; b++ )
         for( d = 0; d < seqs_per_el; d++ )
            data[a].flag[b][d] = 0;           /* mark each sequence with a flag */

   for( f = 0; f < total_sequences; f++ )
   {
      a = (int)( Uniform_Random( &integer_seed ) * (float)aircraft );
      if( a == aircraft )
         a = aircraft - 1;                    /* generate a random target class */

      b = (int)( Uniform_Random( &integer_seed ) * (float)rcst_elevation );
      if( b == rcst_elevation )
         b = rcst_elevation - 1;              /* random elevation */

      d = (int)( Uniform_Random( &integer_seed ) * (float)seqs_per_el );
      if( d == seqs_per_el )
         d = seqs_per_el - 1;                 /* random azimuth to begin the sequence */

      while( data[a].flag[b][d] == 1 )        /* search from that point through the */
      {                                       /* sequences until you find a sequence */
```

```
         d++;                                  /* that has not yet gone into the file */
         if( d == seqs_per_el )
         {
            d = 0;
            b++;
            if( b == rcst_elevation )
            {
               b = 0;
               a++;
               if( a == aircraft )
                  a = 0;
            }
         }
      }
      data[a].flag[b][d] = 1;                  /* Mark the sequence flag */
                                               /* Generate the azimuth bias that applies
                                                  to all signatures in the sequence. */
      azimuth_offset = azimuth_offset_sigma * Gaussian_Random( &integer_seed );
      if( azimuth_offset > 3.0 * azimuth_offset_sigma )
          azimuth_offset = 3.0 * azimuth_offset_sigma;
      if( azimuth_offset < -3.0 * azimuth_offset_sigma )
          azimuth_offset = -3.0 * azimuth_offset_sigma;

      for( e = 0; e < sigs_per_seq; e++ )      /* Loop over all signatures per sequence */
      {
         c = d + e;
         Extract_Noisy_Features();
         Print_Noisy_Signature();              /* noisy, normalized data */
/*         Print_Signature();  */              /* noisy, unnormalized data */
      }
   }

   return;
}
/* end RANDOM_SEQUENCES */



/* Subroutine to print noisy sequences in aspect angle order (test data) ***********/

void List_Noisy_Sequences()
{
   for( b = 0; b < rcst_elevation; b++ )
      for( d = 0; d < seqs_per_el; d++ )
         for( a = 0; a < aircraft; a++ )
         {
            azimuth_offset = azimuth_offset_sigma * Gaussian_Random( &integer_seed );
            if( azimuth_offset > 3.0 * azimuth_offset_sigma )
               azimuth_offset = 3.0 * azimuth_offset_sigma;
            if( azimuth_offset < -3.0 * azimuth_offset_sigma )
               azimuth_offset = -3.0 * azimuth_offset_sigma;

            for( e = 0; e < sigs_per_seq; e++ )
            {
               c = d + e;
               Extract_Noisy_Features();
               Print_Noisy_Signature();
/*               Print_Signature();    */
            }
         }

   return;
}
```

```
/* end LIST_NOISY_SEQUENCES */



/* Subroutine to add noise to the noise-free RCS TOOLBOX signatures, and extract
   features based on the noisy data values ***************************************/

void Extract_Noisy_Features()
{
   Add_Noise();

   signature_output[0] = azimuth_estimate;

   window_start = 0;
   window_end = range_bins - 1;

   while( noise_input[window_start * 2]       <= amplitude_threshold &&
          noise_input[1 + (window_start * 2)] <= amplitude_threshold &&
          window_start < window_end )
      window_start++;

   while( noise_input[window_end * 2]       <= amplitude_threshold &&
          noise_input[1 + (window_end * 2)] <= amplitude_threshold &&
          window_start < window_end )
      window_end--;

   target_region = window_end - window_start + 1;        /* Get the target width */
   signature_output[1] = (float)target_region;

   Downsample( noise_input );                            /* Select amplitude features */

   return;
}
/* end EXTRACT_NOISY_FEATURES */



/* Subroutine to add gaussian noise to the azimuth and to each amplitude in the
noise-free RCS TOOLBOX data files ***********************************************/

void Add_Noise()
{
   azimuth_noise = azimuth_sigma * Gaussian_Random( &integer_seed );
   if( azimuth_noise > 3.0 * azimuth_sigma )
      azimuth_noise = 3.0 * azimuth_sigma;
   if( azimuth_noise < -3.0 * azimuth_sigma )
      azimuth_noise = -3.0 * azimuth_sigma;

   azimuth_estimate = data[a].aspect[b][c][1] + azimuth_offset + azimuth_noise;

   for( i = 0; i < (range_bins * 2); i++ )       /* Loop over all amplitude values */
   {
      amplitude_noise = amplitude_sigma * Gaussian_Random( &integer_seed );
      if( amplitude_noise > 3.0 * amplitude_sigma )
         amplitude_noise = 3.0 * amplitude_sigma;
      if( amplitude_noise < -3.0 * amplitude_sigma )
         amplitude_noise = -3.0 * amplitude_sigma;

      noise_input[i] = data[a].input[b][c][i] + amplitude_noise;
   }

   return;
```

```
}
/* end ADD_NOISE */



/* Subroutine to print one line of formatted, normalized, noisy data ************/

void Print_Noisy_Signature()
{
   fprintf( RECURRENT_DATA, "%6.1f  ", data[a].aspect[b][c][0] );
   fprintf( RECURRENT_DATA, "%6.1f  ", data[a].aspect[b][c][1] );

   fprintf( RECURRENT_DATA, "%9.6f  ", ( signature_output[0] - az_mean ) / az_sigma );
   fprintf( RECURRENT_DATA, "%9.6f  ", ( signature_output[1] - width_mean )
                                                              / width_sigma );

   for( i = 2; i < 2 + (feature_bins * 2); i++ )
      fprintf( RECURRENT_DATA, "%9.6f  ", ( signature_output[i] - amp_mean )
                                                              / amp_sigma );

   for( i = 0; i < aircraft; i++ )
   {
      if( i == a )
         fprintf( RECURRENT_DATA, "%4.1f  ", node_on );
      else
         fprintf( RECURRENT_DATA, "%4.1f  ", node_off );
   }

   fprintf( RECURRENT_DATA, "\n" );

   return;
}
/* end PRINT_NOISY_SIGNATURE */


/* end FORMAT_RCST */
```

Appendix C. RECURRENT Source Code

This appendix contains the RECURRENT source code. It is based on a real-time recurrent learning algorithm [Williams and Zipser, 1989:270-280]. This computer program is an updated and expanded version of RECNET [Lindsey, 1991:58-74].

```
/*  recurrent.c  ********************************************************************

This program is a real-time recurrent learning algorithm modified for radar
target identification.  Input vectors are tagged with target azimuth and elevation
values.  Vectors are arranged in sequences, with a fixed sequence length.  Each
sequence represents high range resolution radar data measured as the target aspect
angles change.

Each line in the input files have the following format:
{ [elevation tag], [azimuth tag], [azimuth feature value], [target width feature value],
[LV amplitude 1], [LH amplitude 1], [LV amplitude 2], [LH amplitude 2]... (etc, for all
desired range bins describing each signature), [class 1 output node desired value],
[class 2 output node desired value]... (etc, for all target classes) }

This format applies only to the radar target identification problem.  The real time
recurrent learning algorithm in this program is written general enough to handle
other problems, such as XOR and function prediction.

Original code written Summer 1991 by Randall L. Lindsey
Updated and modified Fall 1992 by Eric T. Kouba

*****************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define bias                    1
#define exp_sigmoid_threshold   0.2   /* If the different between desired and actual */
#define tanh_sigmoid_threshold  0.4   /* output node values is less than threshold,  */
                                      /* then the output node is "right".            */

void Exit_Message();        /* Subroutines from UTILITIES.C */
FILE *Open_File2();
FILE *Open_File3();
int Get_Integer();
float Uniform_Random();     /* RAN1.C from Numerical recipies in C. */
int integer_seed;
float *Vector();
float **Matrix();
float ***Matrix_3D();

FILE *SETUP;          /* Contains net architecture and data file size parameters */
FILE *DATA;           /* Input file for training and tes data */
FILE *WEIGHTS;        /* Input and output file for the weights matrix */
FILE *SIGNATURES;     /* Output file, giving node by node and line by line listing of
                         desired versus actual output values. */
FILE *LENGTH;         /* Output file, giving epoch accuracy based on sequence length. */
FILE *SEQUENCES;      /* Output file, classifies each sequence (not very useful). */
FILE *EPOCHS;         /* Output file, giving means squared error, alpha, percent
                         right, and percent good for all data during training. */
```

```
char *directory;          /* Target directory to read inputs and write outputs. */
char *weights_filename;   /* Existing weights file name for restarting the net. */
char comments[80];
int lower_limit;
int inputs;               /* Number of input nodes */
int outputs;              /* Number of output nodes */
int hidden;               /* Number of hidden nodes */
int output_layer;         /* Output plus hidden nodes = number of output layer nodes. */
int input_layer;          /* Bias plus input plus output plus hidden = input layer. */
int epochs;               /* Training epochs.  Set = 0 for a testing run. */
float alpha;              /* Initial learning step size */
int sigmoid_flag;         /* Marks type of non-linear function to use. */
int learn_flag;           /* Batch mode flag to show when to do weight updates. */

struct {
   int sequences;            /* Number of sequences */
   int signatures;           /* Number of signatures per sequence */
   int all_data;             /* Total number of data vectors */
   float *elevation_tag;     /* Aspect angle tags are not used as features */
   float *azimuth_tag;
   float **z;                /* External and feedback inputs */
   float **d;          /* Desired outputs */
} data[3];             /* data[0] for training, data[1] for check, data[2] for test */
                       /* Variable "c" switches between data files. */
float *right_length;   /* Number of sequences that were classified "right" */
float *good_length;    /*   and "good", for all possible sequences lengths. */

float **w;               /* Weight matrix */
float **delta_w;         /* Weight update based on a single training vector */
float **batch_delta_w;   /* Batch weight update */
float *s;                /* Result of vector*matrix multiply, before sigmoid */
float *y;                /*    vector*matrix multiply with sigmoid applied */
float *y_prime;          /* Derivative of node outputs w.r.t. weights */
float ***p_t;            /* Previous partial derivatives matrix */
float ***p_t_plus1;      /* Current partial derivatives matrix */
float ***p_temp;
float sum;
float kronecker;         /* Delta function */
float *e;                /* Desired - actual = error for output nodes */
float J[2];              /* Mean squared error per output node */
float *mean_desired;     /* Average desired node value over a given sequence */
float *mean_output;      /* Average actual node output value over a given sequence. */

float threshold;             /* For getting a node "right" */
int right_outputs;           /* Number of nodes with abs(desired - actual) < threshold */
int right_signatures;        /* Number of signatures classified right in a sequence */
int right_last_signature;    /* flag if the last signature in a sequence was "right" */
int epoch_right_signatures;  /* Total number of signatures classified within threshold */
int right_sequences_vote;    /* Number of sequences classified "right" by voting all   */
int right_sequences_mean;    /*  signatures, by taking the average signature outputs,  */
int right_sequences_last;    /*  and by using only the last signature in the sequence. */

float max_output;            /* Value of the maximum actual output */
int max_output_index;        /* Index of the node with max actual output */
float max_desired;           /* Value of maximum desired output */
int max_desired_index;       /* Index of the node with max desired output */
int good_signatures;         /* Signature classified good if index of max desired and  */
int good_last_signature;     /*   max actual are the same node. */
int epoch_good_signatures;
int good_sequences_vote;
int good_sequences_mean;
int good_sequences_last;
```

```
int net_flag;          /* Flag for either training or testing the net. */
int a;                 /* Epochs loop counter */
int b;                 /* Sequences loop counter */
int c;                 /* Data file pointer, 1 for training, 2 for check, 3 for test */
int t;                 /* Time counter, for signatures per sequence */
int u;                 /* Alternate time counter, used only for sequence length analysis */
int i, j, k, l;        /* generic counters */

void Read_Setup();     /* Subroutines */
void Read_Data();
void Allocate_Memory();
void Initialize_Weights();
void Read_Weights();
void Save_Weights();
void Print_Headers();
void Net();
void Initialize_Epoch();
void Initialize_Sequence();
void Initialize_Signature();
void Propagate();
void Compute_Error();
void Classify_Signature();
void Compute_Output();
float Exp_Sigmoid( float x );
float Tanh_Sigmoid( float x );
void Compute_Delta_Weights();
void Compute_P();
void Learn();
void Print_Signature();
void Classify_Sequence();
void Print_Sequence();
void Adjust_Alpha();
void Print_Sequence_Length();
void Print_Epoch();



/* main program for RECURRENT ************************************************/

void main( int argc, char *argv[] )
{
   if( argc != 2 && argc != 3 )
      Exit_Message( "Usage:  recurrent <directory> [weights_filename]" );
   directory = argv[1];                  /* Target directory for input and output files */
   if( argc == 3 )
      weights_filename = argv[2];        /* Filename for an existing weights matrix. */


   SETUP = Open_File2( directory, "/train.setup", "r" );
   Read_Setup( c = 0 );
   fclose( SETUP );
   if( epochs != 0 )
   {                          /* Read training data if there are training epochs to run */
      DATA = Open_File2( directory, "/train", "r" );
      Read_Data( c = 0 );
      fclose( DATA );
   }


   SETUP = Open_File2( directory, "/check.setup", "r" );
   Read_Setup( c = 1 );
   fclose( SETUP );
```

```
DATA = Open_File2( directory, "/check", "r" );
Read_Data( c = 1 );
fclose( DATA );


SETUP = Open_File2( directory, "/test.setup", "r" );
Read_Setup( c = 2 );
fclose( SETUP );

DATA = Open_File2( directory, "/test", "r" );
Read_Data( c = 2 );
fclose( DATA );


Allocate_Memory();                      /* Dynamic allocation for matrix memory */

if( argc == 2 )                         /* If no weights filename specified, */
{
   Initialize_Weights();                /* Generate a random weight matrix. */
   WEIGHTS = Open_File2( directory, "/weights.initial", "w" );
}
else
{
   WEIGHTS = Open_File3( directory, "/", weights_filename, "r" );
   Read_Weights();
   fclose( WEIGHTS );
   WEIGHTS = Open_File2( directory, "/weights.restart", "w" );
}
Save_Weights();                         /* Save the starting weights */
fclose( WEIGHTS );


WEIGHTS = Open_File2( directory, "/weights.final", "w" );
EPOCHS = Open_File2( directory, "/epochs", "w" );
Print_Headers( net_flag = 0 );          /* Print screen and file headers */

for( a = 0; a < epochs; a++ )           /* Loop over all training epochs */
{
   Net( c = 0, net_flag = 0 );          /* Train the net, perform weight updates */
   Net( c = 1, net_flag = 1 );          /* Check data, no updates */
}

Save_Weights();
fclose( WEIGHTS );

                                        /* Training complete, print results */
SIGNATURES = Open_File2( directory, "/signatures.check", "w" );
SEQUENCES = Open_File2( directory, "/sequences.check", "w" );
LENGTH = Open_File2( directory, "/length.check", "w" );
Print_Headers( net_flag = 2 );
fprintf( LENGTH, "!check" );

Net( c = 1, net_flag = 2 );             /* Print results versus check data file */

fclose( SIGNATURES );
fclose( SEQUENCES );
fclose( LENGTH );


SIGNATURES = Open_File2( directory, "/signatures.test", "w" );
SEQUENCES = Open_File2( directory, "/sequences.test", "w" );
LENGTH = Open_File2( directory, "/length.test", "w" );
Print_Headers( net_flag = 2 );
fprintf( LENGTH, "!test" );
```

```
   Net( c = 2, net_flag = 2 );            /* Print results versus test data file */

   fclose( SIGNATURES );
   fclose( SEQUENCES );
   fclose( LENGTH );
   fclose( EPOCHS );
}
/* end MAIN */



/* Subroutine to read in the net architecture and data file size parameters **********/

void Read_Setup( int c )
{
   if( c == 0 )                                        /* If training setup file */
   {
      inputs = Get_Integer( SETUP, lower_limit = 1 );
      outputs = Get_Integer( SETUP, lower_limit = 1 );
      hidden = Get_Integer( SETUP, lower_limit = 0 );

      output_layer = outputs + hidden;
      input_layer = bias + inputs + outputs + hidden;

      printf( "%d input, %d output, %d hidden nodes\n\n", inputs, outputs, hidden );
   }
   else                                        /* Skip for check and test setup files */
      for( i = 0; i < 3; i++ )
         fgets( comments, 80, SETUP );

   data[c].sequences = Get_Integer( SETUP, lower_limit = 1 );  /* Number of sequences */
   data[c].signatures = Get_Integer( SETUP, lower_limit = 2 );
                                                          /* Signatures per sequence */
   data[c].all_data = data[c].sequences * data[c].signatures;

   if( c == 0 )                                          /* If training setup file */
   {
      epochs = Get_Integer( SETUP, lower_limit = 0 );

      fscanf( SETUP, "%f", &alpha );
      fgets( comments, 80, SETUP );
      if( alpha <= 0.0 )
         Exit_Message( "alpha <= 0.0 in setup file" );

      sigmoid_flag = Get_Integer( SETUP, lower_limit = 0 );
      if( sigmoid_flag == 1 || sigmoid_flag == 2 )
         threshold = exp_sigmoid_threshold;
      else if( sigmoid_flag == 3 )
         threshold = tanh_sigmoid_threshold;
      else
         Exit_Message( "bad output flag in setup file" );

      learn_flag = Get_Integer( SETUP, lower_limit = 0 );
      if( learn_flag != 1 && learn_flag != 2 && learn_flag != 3 )
         Exit_Message( "bad learn flag in setup file" );
   }

   return;
}
/* end READ_SETUP */
```

```
/* Subroutine to read all vectors in a data file ********************************/

void Read_Data( int c )
{
   data[c].elevation_tag = Vector( 0, data[c].all_data - 1 );
   data[c].azimuth_tag = Vector( 0, data[c].all_data - 1 );
   data[c].z = Matrix( 0, data[c].all_data - 1, 0, input_layer - 1 );
   data[c].d = Matrix( 0, data[c].all_data - 1, 0, outputs - 1 );

   for( i = 0; i < data[c].all_data; i++ )
   {
      fscanf( DATA, "%f", &data[c].elevation_tag[i] );     /* Aspect angle tags are not */
      fscanf( DATA, "%f", &data[c].azimuth_tag[i] );       /* used as feature values */

      data[c].z[i][0] = 1.0;                               /* Uniform bias */
      for( j = 0; j < inputs; j++ )
         fscanf( DATA, "%f", &data[c].z[i][j + bias] );    /* Input features */
      for( j = 0; j < output_layer; j++ )
         data[c].z[i][j + bias + inputs] = 0.0;       /* Initialize all feedback inputs */

      for( j = 0; j < outputs; j++ )
         fscanf( DATA, "%f", &data[c].d[i][j] );           /* Desired outputs */
   }

   printf( "%d sequences, %d signatures\n\n", data[c].sequences, data[c].signatures );

   return;
}
/* end READ_DATA */



/* Subroutine to dynamically allocate matrix memory ******************************/

void Allocate_Memory()
{
   w = Matrix( 0, output_layer - 1, 0, input_layer - 1 );         /* Weights */
   delta_w = Matrix( 0, output_layer - 1, 0, input_layer - 1 );   /* Weight updates */
   batch_delta_w = Matrix( 0, output_layer - 1, 0, input_layer - 1 );  /* Update batch */
   s = Vector( 0, output_layer - 1 );          /* Vector*matrix multiply, before sigmoid */
   y = Vector( 0, output_layer - 1 );          /* Vector*matrix multiply, after sigmoid */
   y_prime = Vector( 0, output_layer - 1 );     /* Derivative of outputs w.r.t. weights */
   p_t = Matrix_3D( 0, output_layer - 1, 0, input_layer - 1, 0, output_layer - 1 );
                                               /* Previous partial derivatives matrix */
   p_t_plus1 = Matrix_3D( 0, output_layer - 1, 0, input_layer - 1, 0, output_layer - 1 );
                                               /* Current partial derivatives matrix */
   e = Vector( 0, outputs - 1 );                /* Error = desired - actual */

   right_length = Vector( 1, data[1].signatures - 1 );  /* Counters for sequence */
   good_length = Vector( 1, data[1].signatures - 1 );   /*   length analysis */

   mean_desired = Vector( 0, outputs - 1 );    /* Average desired output for a sequence */
   mean_output = Vector( 0, outputs - 1 );     /* Average actual output for a sequence */

   return;
}
/* end ALLOCATE_MEMORY */
```

```
/* Subroutine to generate a random weight matrix ******************************/

void Initialize_Weights()
{
   srand( time( NULL ));          /* Use the cheap Unix random number generator */
   integer_seed = -rand();        /* to seed the high-class Numerical Recipies  */
   Uniform_Random( &integer_seed );                                  /* generator */
   printf( "random weight matrix\n" );

   for( i = 0; i < output_layer; i++ )
   {
      for( j = 0; j < input_layer; j++ )
      {
         w[i][j] = 2 * Uniform_Random( &integer_seed ) - 1.0;       /* U[-1,+1] */
         printf( "%f  ", w[i][j] );
      }
      printf( "\n" );
   }
   printf( "\n" );

   return;
}
/* end INITIALIZE WEIGHTS */



/* Subroutine to read an existing weight matrix ******************************/

void Read_Weights()
{
   printf( "reading weight matrix\n" );

   for( i = 0; i < output_layer; i++ )
   {
      for( j = 0; j < input_layer; j++ )
      {
         fscanf( WEIGHTS, "%f", &w[i][j] );
         printf( "%f  ", w[i][j] );
      }
      printf( "\n" );
   }
   printf( "\n" );

   return;
}
/* end READ_WEIGHTS */



/* Subroutine to save a weight matrix to a file ******************************/

void Save_Weights()
{
   for( i = 0; i < output_layer; i++ )
   {
      for( j = 0; j < input_layer; j++ )
         fprintf( WEIGHTS, "%f  ", w[i][j] );
      fprintf( WEIGHTS, "\n" );
   }

   return;
}
```

```
/* end SAVE_WEIGHTS */



/* Subroutine to print screen and file headers.  The exclamation points allow epoch
files to be read directly by "NEYPLOT" for plotting. ************************/

void Print_Headers( int net_flag )
{
   if( net_flag == 0 )                              /* Training and check file headers */
   {
      fprintf( EPOCHS, "!\t\t\t\tsignatures\t\t\t\tvote\t\t\t\tmean\t\t\t\tlast\n" );
      fprintf( EPOCHS, "!epoch\tdistance    alpha\tright\t\tgood\t\tright\t\tgood\t\t" );
      fprintf( EPOCHS, "right\t\tgood\t\tright\t\tgood\n" );

      printf( "\t\t\t\tsignatures\tvote\t\tmean\t\tlast\n" );
      printf( "epoch\tdistance    alpha\tright\tgood\tright\tgood\t" );
      printf( "right\tgood\tright\tgood\n" );
   }

   if( net_flag == 2 )                                          /* Test file headers */
   {
      fprintf( SEQUENCES, "start\t\t\t\tvote\t\t\tmean\t\t\tlast\n" );
      fprintf( SEQUENCES, "elevation azimuth\tright       good\t" );
      fprintf( SEQUENCES, "right\t\tgood\t\tright\tgood\n" );

      fprintf( SIGNATURES, "elevation azimuth\tactual\t\t\tdesired\n" );
   }

   return;
}
/* end PRINT_HEADERS */



/* Primary subroutine for real time recurrent learning ************************/

void Net( int c, int net_flag )
{
   Initialize_Epoch();                              /* Set epoch counters to zero. */

   for( b = 0; b < data[c].sequences; b++ )
   {
      Initialize_Sequence();                        /* Set sequence counters to zero. */

      for( t = 0; t < data[c].signatures; t++ )
      {
         Initialize_Signature();                    /* Set signature counters to zero. */
         Propagate();                               /* Vector*matrix multiply */
         Compute_Error();               /* Difference betwen desired and actual outputs */
         Classify_Signature();          /* Classify outputs for the current time step */

         if( net_flag == 2 )            /* If testing */
            Print_Signature();

         Compute_Output();              /* Apply the sigmoid or tanh non-linearity */

         if( net_flag == 0 )            /* If training */
         {
            Compute_Delta_Weights();    /* Compute weight updates for current time step */
            Compute_P();                /* Perform gradient descent, compute new
                                           partial derivatives matrix */
```

```
            if( learn_flag == 1 )       /* Check flag */
               Learn();                 /* Apply weight updates after every time step */
         }
      }

      Classify_Sequence();

      if( net_flag == 2 )                                   /* If testing */
      {
         Print_Sequence();
         if(( b + 1 ) % outputs == 0 )
            fprintf( SEQUENCES, "\n" );
      }

      if( net_flag == 0 && learn_flag == 2 )          /* If training, check flag */
         Learn();                   /* Apply weight updates after every sequence */
   }

   if( net_flag == 0 )               /* If training */
   {
      if( learn_flag == 3 )          /* Check flag */
         Learn();                    /* Apply weight updates at end of epoch */

      Adjust_Alpha();                /* Decrease learning step size, if required */

      fprintf( EPOCHS, "!%4d:\t", a + 1 );
      printf( "%4d:\t", a + 1 );
   }
   else if( net_flag == 1 )          /* If running through check data during training */
   {
      fprintf( EPOCHS, "!check:\n%d\t", a + 1 );
      printf( "check:\t" );
   }
   else if( net_flag == 2 && c == 1 )  /* If running through check data during testing */
   {
      Print_Sequence_Length();        /* Print net accuracy versus sequence length */

      fprintf( EPOCHS, "check:\t" );
      printf( "check:\t" );
   }
   else                               /* If running through test data */
   {
      Print_Sequence_Length();        /* Print net accuracy versus sequence length */

      fprintf( EPOCHS, "!test:\t" );
      printf( "test:\t" );
   }
   Print_Epoch();                     /* Print cumulative epoch results */

   return;
}
/* end NET */



/* Subroutine to zero epoch counters ********************************************/

void Initialize_Epoch()
{
   epoch_right_signatures = 0;
   right_sequences_vote = 0;
   right_sequences_mean = 0;
   right_sequences_last = 0;
```

```
   epoch_good_signatures = 0;
   good_sequences_vote = 0;
   good_sequences_mean = 0;
   good_sequences_last = 0;

   J[1] = 0.0;                              /* Mean square error for current epochs */
   for( i = 0; i < output_layer; i++ )
      for( j = 0; j < input_layer; j++ )
         batch_delta_w[i][j] = 0.0;         /* Initialize batch updates */

   for( t = 1; t < data[1].signatures; t++ )
   {
      right_length[t] = 0.0;
      good_length[t] = 0.0;
   }

   return;
}
/* end INITIALIZE_EPOCHS */



/* Subroutine to zero sequence counters ***************************************/

void Initialize_Sequence()
{
   right_signatures = 0;
   right_last_signature = 0;
   good_signatures = 0;
   good_last_signature = 0;

   for( i = 0; i < outputs; i++ )
   {
      mean_desired[i] = 0.0;
      mean_output[i] = 0.0;
   }

   for( i = 0; i < output_layer; i++ )
   {
      y[i] = 0.0;               /* Actual outputs */
      y_prime[i] = 0.0;         /* Derivative of outputs w.r.t weights */
      for( j = 0; j < input_layer; j++ )
         for( k = 0; k < output_layer; k++ )
            p_t[i][j][k] = p_t_plus1[i][j][k] = 0.0;  /* Partial derivatives matricies */
   }

   return;
}
/* end INITIALIZE_SEQUENCE */



/* Subroutine to zero signature counters **************************************/

void Initialize_Signature()
{
   right_outputs = 0;
   max_output = -99.0;
   max_output_index = -1;
   max_desired = -99.0;
   max_desired_index = -1;
```

```
   for( i = 0; i < outputs; i++ )
      e[i] = 0.0;                          /* Error */

   for( i = 0; i < output_layer; i++ )
   {
      s[i] = 0.0;                           /* Node activation */
      for( j = 0; j < input_layer; j++ )
         delta_w[i][j] = 0.0;               /* Initialize weight updates (not batch) */
   }

   return;
}
/* end INITIALIZE_SIGNATURE */



/* Subroutine to compute input vector * weight matrix multiply for current time step ***/

void Propagate()
{
   for( i = 0; i < output_layer; i++ )
      data[c].z[b * data[c].signatures + t][i + bias + inputs] = y[i];

   for( i = 0; i < output_layer; i++ )
      for( j = 0; j < input_layer; j++ )
         s[i] += w[i][j] * data[c].z[b * data[c].signatures + t][j];

   return;
}
/* end PROPAGATE */



/* Subroutine to compute desired - actual output node values ***********************/

void Compute_Error()
{
   if( t == 0 )                          /* No error computations for first time step */
      return;

   for( i = 0; i < outputs; i++ )
   {
      e[i] = data[c].d[b * data[c].signatures + t][i] - y[i];
      J[1] += 0.5 * e[i] * e[i];          /* Sum up the mean squared error over all */
   }                                      /* sequences and signatures */

   return;
}
/* end COMPUTE_ERROR */



/* Subroutine to classify the node outputs for the current time step ****************/

void Classify_Signature()
{
   if( t == 0 )
      return;

   for( i = 0; i < outputs; i++ )
```

```
      if( fabs( data[c].d[b * data[c].signatures + t][i] - y[i] ) <= threshold )
         right_outputs++;
   if( right_outputs == outputs )        /* If all actual and desired outputs are close, */
   {
      right_signatures++;                /* Increment the "right" counters. */
      epoch_right_signatures++;
      if( t == data[c].signatures - 1 )
         right_last_signature = 1;
   }

   for( i = 0; i < outputs; i++ )
   {
      if( data[c].d[b * data[c].signatures + t][i] > max_desired )
      {
         max_desired = data[c].d[b * data[c].signatures + t][i];
         max_desired_index = i;          /* Find index of max desired output node */
      }
      if( y[i] > max_output )
      {
         max_output = y[i];
         max_output_index = i;           /* Find index of max actual output node */
      }
   }
   if( max_desired_index == max_output_index )
   {                                     /* If indicies match, then max pick worked */
      good_signatures++;                 /* Increment "good" counters. */
      epoch_good_signatures++;
      if( t == data[c].signatures - 1 )
         good_last_signature = 1;
   }

   for( i = 0; i < outputs; i++ )
   {                                /* Sum up outputs for each signature in a sequence */
      mean_desired[i] += data[c].d[b * data[c].signatures + t][i] /
                                                   ( data[c].signatures - 1 );
      mean_output[i] += y[i] / ( data[c].signatures - 1 `;
   }

   return;
}
/* end CLASSIFY_SIGNATURE */



/* Subroutine to print the node outputs for the current time step, during testing *****/

void Print_Signature()
{
   fprintf( SIGNATURES, "%6.1f   ", data[c].elevation_tag[b * data[c].signatures + t] );
   fprintf( SIGNATURES, "%6.1f   ", data[c].azimuth_tag[b * data[c].signatures + t] );
   for( i = 0; i < outputs; i++ )
      fprintf( SIGNATURES, "%9.6f   ", y[i] );         /* Actual outputs */
   for( i = 0; i < outputs; i++ )                      /* and desired outputs */
      fprintf( SIGNATURES, "%5.2f   ", data[c].d[b * data[c].signatures + t][i] );

   if( t == 0 )
      fprintf( SIGNATURES, "start" );
   else
   {
      if( right_outputs == outputs )
      {
         fprintf( SIGNATURES, "       " );
         right_length[t] += 1.0;
```

```
      }
      else
         fprintf( SIGNATURES, "wrong  " );

      if( max_desired_index == max_output_index )
      {
         fprintf( SIGNATURES, "      " );
         good_length[t] += 1.0;
      }
      else
         fprintf( SIGNATURES, "bad    " );
   }
   fprintf( SIGNATURES, "\n" );

   return;
}
/* end PRINT_SIGNATURE */



/* Subroutine to apply the non-linearity function ********************************/

void Compute_Output()
{
   if( sigmoid_flag == 1 )      /* Use a combination of linear outputs and */
   {                            /* sigmoid hidden nodes. */
      for( i = 0; i < outputs; i++ )
         y[i] = s[i];
      for( i = 0; i < hidden; i++ )
         y[i + outputs] = Exp_Sigmoid( s[i + outputs] );
   }

   else if( sigmoid_flag == 2 )                         /* Use standard sigmoid */
      for( i = 0; i < output_layer; i++ )
         y[i] = Exp_Sigmoid( s[i] );

   else                                                 /* Use tanh function */
      for( i = 0; i < output_layer; i++ )
         y[i] = Tanh_Sigmoid( s[i] );

   return;
}
/* end COMPUTE_OUTPUT */



/* Subroutine for the sigmoid function *******************************/

float Exp_Sigmoid( float x )
{
   static float maximum_value = 50.0;

   if( x > maximum_value )
      return 1.0;

   else if( x < -maximum_value )
      return 0.0;

   else
      return 1.0 / ( 1.0 + exp( -x ));
}
/* end EXP_SIGMOID */
```

```
/* Subroutine for the tanh function *******************************/

float Tanh_Sigmoid( float x )
{
   static float maximum_value = 25.0;

   if( x > maximum_value )
      return 1.0;

   else if( x < -maximum_value )
      return -1.0;

   else
      return tanh( x );
}
/* end TANH_SIGMOID */



/* Subroutine to compute weight updates for the current time step ******************/

void Compute_Delta_Weights()
{
   for( i = 0; i < output_layer; i++ )
      for( j = 0; j < input_layer; j++ )
      {
         for( k = 0; k < outputs; k++ )
            delta_w[i][j] += alpha * e[k] * p_t[i][j][k];
         batch_delta_w[i][j] += delta_w[i][j];          /* Sum into batch update */
      }                 /* Apply updates after each time step, sequence, or epoch */

   return;
}
/* end COMPUTE_DELTA_WEIGHTS */



/* Subroutine to compute the new partial derivatives matrix ******************/

void Compute_P()
{
   if( sigmoid_flag == 1 )                /* Linear outputs, sigmoid hidden nodes */
   {
      for( i = 0; i < outputs; i++ )
         y_prime[i] = 1.0;
      for( i = 0; i < hidden; i++ )
         y_prime[i + outputs] = y[i + outputs] * ( 1.0 - y[i + outputs] );
   }

   else if( sigmoid_flag == 2 )           /* Sigmoid for entire output layer */
      for( i = 0; i < output_layer; i++ )
         y_prime[i] = y[i] * ( 1.0 - y[i] );

   else                                   /* Tanh function for output layer */
      for( i = 0; i < output_layer; i++ )
         y_prime[i] = 1.0 - ( y[i] * y[i] );

   for( i = 0; i < output_layer; i++ )
```

```
      for( j = 0; j < input_layer; j++ )
         for( k = 0; k < output_layer; k++ )
         {
            kronecker = 0.0;
            if ( i == k )
               kronecker = 1.0;
            sum = 0.0;

            for( l = 0; l < output_layer; l++ )
               sum += w[k][l + bias + inputs] * p_t[i][j][l];

            p_t_plus1[i][j][k] = y_prime[k] *
                        (sum + kronecker * data[c].z[b * data[c].signatures + t][j] );
         }                                  /* Compute new P matrix based on old P matrix */

   p_temp = p_t;
   p_t = p_t_plus1;                    /* Swap pointers, now P_T is the current P matrix */
   p_t_plus1 = p_temp;

   retu ;
}
/* end COMPUTE_P */



/* Subroutine to apply the weight updates and reset batch mode update ******************/

void Learn()
{
   for( i = 0; i < output_layer; i++ )
      for( j = 0; j < input_layer; j++ )
      {
         w[i][j] += batch_delta_w[i][j];
         batch_delta_w[i][j] = 0.0;
      }

   return;
}
/* end LEARN */



/* Subroutine to classify a sequence based on time step outputs ******************/

void Classify_Sequence()
{
   if( right_signatures >= data[c].signatures / 2 )
      right_sequences_vote++;                          /* Majority vote scheme */
   if( good_signatures >= data[c].signatures / 2 )
      good_sequences_vote++;

   Initialize_Signature();              /* Reset signature stats (dual use variables) */

   for( i = 0; i < outputs; i++ )
      if( fabs( mean_desired[i] - mean_output[i] ) <= threshold )
         right_outputs++;
   if( right_outputs == outputs )
      right_sequences_mean++;                /* Average output per time step */

   for( i = 0; i < outputs; i++ )
   {
      if( mean_desired[i] > max_desired )
```

```
      {
         max_desired = mean_desired[i];
         max_desired_index = i;
      }
      if( mean_output[i] > max_output )
      {
         max_output = mean_output[i];
         max_output_index = i;
      }
   }
   if( max_desired_index == max_output_index )
      good_sequences_mean++;                  /* Average output per time step */

   if( right_last_signature == 1 )
      right_sequences_last++;                 /* Only look at last time step */
   if( good_last_signature == 1 )
      good_sequences_last++;

   return;
}
/* end CLASSIFY_SEQUENCE */



/* Subroutine to print sequence results (not very useful) ******************/

void Print_Sequence()
{
   fprintf( SIGNATURES, "\n" );                          /* Print aspect angle tags */
   fprintf( SEQUENCES, "%6.1f   ", data[c].elevation_tag[b * data[c].signatures] );
   fprintf( SEQUENCES, "%6.1f\t\t", data[c].azimuth_tag[b * data[c].signatures] );

   if( right_signatures >= data[c].signatures / 2 )          /* Voting records */
      fprintf( SEQUENCES, "%2d          ", right_signatures );
   else
      fprintf( SEQUENCES, "%2d  wrong   ", right_signatures );

   if( good_signatures >= data[c].signatures / 2 )           /* Voting records */
      fprintf( SEQUENCES, "%2d     \t", good_signatures );
   else
      fprintf( SEQUENCES, "%2d  bad\t", good_signatures );

   if( right_outputs == outputs )                        /* Average time step response */
      fprintf( SEQUENCES, "%2d          ", right_outputs );
   else
      fprintf( SEQUENCES, "%2d  wrong   ", right_outputs );

   if( max_desired_index == max_output_index )
      fprintf( SEQUENCES, "        \t\t" );
   else                                                  /* Average time step response */
      fprintf( SEQUENCES, "     bad\t\t" );

   if( right_last_signature == 1 )                          /* Last time step only */
      fprintf( SEQUENCES, "\t" );
   else
      fprintf( SEQUENCES, "wrong\t" );

   if( good_last_signature == 1 )                          /* Last time step only */
      fprintf( SEQUENCES, "\t" );
   else
      fprintf( SEQUENCES, "bad\t" );

   fprintf( SEQUENCES, "\n" );
```

```
   return;
}
/* end PRINT_SEQUENCE */



/* Subroutine to decrease learning step size, if required ************************/

void Adjust_Alpha()
{
   if( ( J[1] / data[0].all_data ) < 0.0005 )        /* If error less than an arbitrary */
   {                                                 /* amount, exit training */
      printf( "exit after epoch %d\n", a + 1 );
      a = epochs;                                    /* (but print results first) */
   }

   if( a < 5 || J[1] < J[0] )      /* For first 5 epochs or while error is decreasing, */
      J[0] = J[1];                 /* do nothing. */
   else if( J[1] < 1.01 * J[0] )    /* If error increases by less than 1%, remember the */
      ;                             /* previous lowest mean squared error value in case */
   else                             /* error increases again. */
   {
      alpha *= 0.5;          /* If error increases by more than 1%, decrease alpha */
      J[0] = J[1];
   }

   return;
}
/* end ADJUST_ALPHA */



/* Subroutine to print net accuracy as sequence length varies.
The output file is formatted for reading and plotting by NXYPLOT ********************/

void Print_Sequence_Length()
{
   fprintf( LENGTH, "\n!sig\t   right\t   good\n" );
   printf( "\nsig\t   right\t   good\n" );

   for( u = 1; u < data[1].signatures; u++ )
   {                        /* bug: limited to >check file< signatures per sequence */
      fprintf( LENGTH, "%2d\t", u + 1 );
      fprintf( LENGTH, "%6d\t", (int)right_length[u] );
      fprintf( LENGTH, "%4.1f%%\t", 100.0 * right_length[u] / (float)data[c].sequences );
      fprintf( LENGTH, "%5d\t", (int)good_length[u] );
      fprintf( LENGTH, "%4.1f%%\n", 100.0 * good_length[u] / (float)data[c].sequences );

      printf( "%2d\t", u + 1 );
      printf( "%6d\t", (int)right_length[u] );
      printf( "(%4.1f%%)\t", 100.0 * right_length[u] / (float)data[c].sequences );
      printf( "%6d\t", (int)good_length[u] );
      printf( "(%4.1f%%)\n", 100.0 * good_length[u] / (float)data[c].sequences );
   }

   fprintf( LENGTH, "\n" );
   printf( "\n" );

   return;
}
/* end PRINT_SEQUENCE_LENGTH */
```

```
/* Subroutine to print composite results for an entire epoch (to file and screen) ******/

void Print_Epoch()
{                   /* Mean squared error per output node, and learning step size */
   fprintf( EPOCHS, "%9.6f   %8.6f\t", J[1] / data[c].all_data, alpha );
                                              /* Straight count of signatures */
   fprintf( EPOCHS, "%4d  ", epoch_right_signatures );
   fprintf( EPOCHS, "%4.1f%%\t", (float)epoch_right_signatures /
(float)data[c].all_data*100.0 );
   fprintf( EPOCHS, "%4d  ", epoch_good_signatures );
   fprintf( EPOCHS, "%4.1f%%\t", (float)epoch_good_signatures /
(float)data[c].all_data*100.0 );
                                                    /* Voting scheme results */
   fprintf( EPOCHS, "%3d  ", right_sequences_vote );
   fprintf( EPOCHS, "%4.1f%%\t", (float)right_sequences_vote /
(float)data[c].sequences*100.0 );
   fprintf( EPOCHS, "%3d  ", good_sequences_vote );
   fprintf( EPOCHS, "%4.1f%%\t", (float)good_sequences_vote /
(float)data[c].sequences*100.0 );
                                                     /* Average response results */
   fprintf( EPOCHS, "%3d  ", right_sequences_mean );
   fprintf( EPOCHS, "%4.1f%%\t", (float)right_sequences_mean /
(float)data[c].sequences*100.0 );
   fprintf( EPOCHS, "%3d  ", good_sequences_mean );
   fprintf( EPOCHS, "%4.1f%%\t", (float)good_sequences_mean /
(float)data[c].sequences*100.0 );
                                                     /* Last time step results */
   fprintf( EPOCHS, "%3d  ", right_sequences_last );
   fprintf( EPOCHS, "%4.1f%%\t", (float)right_sequences_last /
(float)data[c].sequences*100.0 );
   fprintf( EPOCHS, "%3d  ", good_sequences_last );
   fprintf( EPOCHS, "%4.1f%%\t", (float)good_sequences_last /
(float)data[c].sequences*100.0 );

   fprintf( EPOCHS, "\n" );

   printf( "%9.6f   %8.6f\t", J[1] / data[c].all_data, alpha );

   printf( "%4.1f%%\t", (float)epoch_right_signatures / (float)data[c].all_data * 100.0 );
   printf( "%4.1f%%\t", (float)epoch_good_signatures / (float)data[c].all_data * 100.0 );

   printf( "%4.1f%%\t", (float)right_sequences_vote / (float)data[c].sequences * 100.0 );
   printf( "%4.1f%%\t", (float)good_sequences_vote / (float)data[c].sequences * 100.0 );

   printf( "%4.1f%%\t", (float)right_sequences_mean / (float)data[c].sequences * 100.0 );
   printf( "%4.1f%%\t", (float)good_sequences_mean / (float)data[c].sequences * 100.0 );

   printf( "%4.1f%%\t", (float)right_sequences_last / (float)data[c].sequences * 100.0 );
   printf( "%4.1f%%\t", (float)good_sequences_last / (float)data[c].sequences * 100.0 );

   printf( "\n" );

   return;
}
/* end PRINT_EPOCH */

/* end RECURRENT */
```

Appendix D. UTILITIES Source Code

This appendix contains the UTILITIES source code. These subroutines are used by function calls from FORMAT_RCST and RECURRENT.

```
/* utilities.c ********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>


void Exit_Message( char *message )
{
   printf( "%s\nexit\n", message );
   exit( 1 );
}


FILE *Open_File2( char *pre, char *ext1, char *mode )
{
   char *new_name;
   FILE *NEW_FILE;

   new_name = malloc( strlen( pre ) + strlen( ext1 ) + 1 );
   strcpy( new_name, pre );
   strcat( new_name, ext1 );

   NEW_FILE = fopen( new_name, mode );
   if( NEW_FILE == NULL )
   {
      printf( "Cannot find file:  %s\nexit\n", new_name );
      exit( 1 );
   }

   printf( "%s\t%s\n", mode, new_name );
   return NEW_FILE;
}


FILE *Open_File3( char *pre, char *ext1, char *ext2, char *mode )
{
   char *new_name;
   FILE *NEW_FILE;

   new_name = malloc( strlen( pre ) + strlen( ext1 ) + strlen(ext2 ) + 1 );
   strcpy( new_name, pre );
   strcat( new_name, ext1 );
   strcat( new_name, ext2 );

   NEW_FILE = fopen( new_name, mode );
   if( NEW_FILE == NULL )
   {
      printf( "Cannot find file:  %s\nexit\n", new_name );
      exit( 1 );
```

```
   }

   printf( "%s\t%s\n", mode, new_name );
   return NEW_FILE;
}


FILE *Open_File4( char *pre, char *ext1, char *ext2, char *ext3, char *mode )
{
   char *new_name;
   FILE *NEW_FILE;

   new_name = malloc(  strlen( pre ) + strlen( ext1 ) + strlen( ext2 )
                                                      + strlen( ext3 ) + 1 );
   strcpy( new_name, pre );
   strcat( new_name, ext1 );
   strcat( new_name, ext2 );
   strcat( new_name, ext3 );

   NEW_FILE = fopen( new_name, mode );
   if( NEW_FILE == NULL )
   {
      printf( "Cannot find file:  %s\nexit\n", new_name );
      exit( 1 );
   }

   printf( "%s\t%s\n", mode, new_name );
   return NEW_FILE;
}


int Get_Integer( FILE *TEMP, int lower_limit )
{
   int variable;
   char comments[100];

   fscanf( TEMP, "%d", &variable );
   fgets( comments, 100, TEMP );
   if( variable < lower_limit )
      Exit_Message( "Integer error in setup file" );

   return variable;
}


/*  UNIFORM_RANDOM is the RAN1 routine from Numerical Recipies in C. */

#define M1     259200
#define IA1      7141
#define IC1     54773
#define RM1   ( 1.0 / M1 )
#define M2     134456
#define IA2      8121
#define IC2     28411
#define RM2   ( 1.0 / M2 )
#define M3     243000
#define IA3      4561
#define IC3     51349

extern float Uniform_Random( int *integer_seed )
{
```

```
   static int iff = 0;
   static long ix1;
   static long ix2;
   static long ix3;
   static int j;
   static float r[98];
   float temp;

   if ( *integer_seed < 0 || iff == 0 )
   {
      iff = 1;
      ix1 = ( IC1 - ( *integer_seed )) % M1;
      ix1 = ( IA1 * ix1 + IC1 ) % M1;
      ix2 = ix1 % M2;
      ix1 = ( IA1 * ix1 + IC1 ) % M1;
      ix3 = ix1 % M3;
      for( j = 1; j <= 97; j++ )
      {
         ix1 = ( IA1 * ix1 + IC1 ) % M1;
         ix2 = ( IA2 * ix2 + IC2 ) % M2;
         r[j] = ( ix1 + ix2 * RM2 ) * RM1;
      }
      *integer_seed = 1;
   }
   ix1 = ( IA1 * ix1 + IC1 ) % M1;
   ix2 = ( IA2 * ix2 + IC2 ) % M2;
   ix3 = ( IA3 * ix3 + IC3 ) % M3;
   j = 1 + (( 97 * ix3 ) / M3 );
   if( j > 97 || j < 1 )
      Exit_Message( "Uniform_Random:  positive seed value" );
   temp = r[j];
   r[j] = ( ix1 + ix2 * RM2 ) * RM1;

   return temp;
}

#undef M1
#undef IA1
#undef IC1
#undef RM1
#undef M2
#undef IA2
#undef IC2
#undef RM2
#undef M3
#undef IA3
#undef IC3


/*  GAUSSIAN_RANDOM is the GASRAN routine from Numerical Recipies in C. */

extern float Gaussian_Random( int *integer_seed )
{
   static int iset = 0;
   static float gset;
   float fac;
   float r;
   float v1;
   float v2;
   float Uniform_Random();

   if( iset == 0 )
   {
```

```
      do
      {
         v1 = 2.0 * Uniform_Random( integer_seed ) - 1.0;
         v2 = 2.0 * Uniform_Random( integer_seed ) - 1.0;
         r = v1 * v1 + v2 * v2;
      }
      while( r >= 1.0 || r == 0.0 );

      fac = sqrt( -2.0 * log( r ) / r );
      gset = v1 * fac;
      iset = 1;
      return v2 * fac;
   }
   else
   {
      iset = 0;
      return gset;
   }
}



int *Integer_Vector( int nl, int nh)       /* from Numerical Recipies in C. */
{
   int *v;

   v = ( int * ) malloc( (unsigned)( nh - nl + 1 ) * sizeof( int ));
   if ( !v )
      Exit_Message( "Integer_Vector:  allocation failure" );
   v -= nl;

   return v;
}



float *Vector( int nl, int nh )       /* from Numerical Recipies in C. */
{
   float *v;

   v = ( float * ) malloc( (unsigned)( nh - nl + 1 ) * sizeof( float ));
   if ( !v )
      Exit_Message( "Vector:  allocation failure" );
   v -= nl;

   return v;
}



int **Integer_Matrix( int nrl, int nrh, int ncl, int nch )
                                                /* from Numerical Recipies in C. */
{
   int i;
   int **m;

   m = ( int ** ) malloc( (unsigned)( nrh - nrl + 1 ) * sizeof( int* ));
   if ( !m )
      Exit_Message( "Integer_Matrix:  allocation failure 1" );
   m -= nrl;

   for( i = nrl; i <= nrh; i++ )
   {
      m[i] = ( int * ) malloc( (unsigned)( nch - ncl + 1 ) * sizeof( int ));
```

```
      if ( !m[i] )
         Exit_Message( "Integer_Matrix:  allocation failure 2" );
      m[i] -= ncl;
   }

   return m;
}



float **Matrix( int nrl, int nrh, int ncl, int nch )
                                                /* from Numerical Recipies in C. */
{
   int i;
   float **m;

   m = ( float ** ) malloc( (unsigned)( nrh - nrl + 1 ) * sizeof( float* ));
   if ( !m )
      Exit_Message( "Matrix:  allocation failure 1" );
   m -= nrl;

   for( i = nrl; i <= nrh; i++ )
   {
      m[i] = ( float * ) malloc( (unsigned)( nch - ncl + 1 ) * sizeof( float ));
      if ( !m[i] )
         Exit_Message( "Matrix:  allocation failure 2" );
      m[i] -= ncl;
   }

   return m;
}



float ***Matrix_3D( int nrl, int nrh, int ncl, int nch, int ndl, int ndh )
                                                /* from the RECNET computer code */
{
   int i, j;
   float ***m;

   m = (float ***) malloc( (unsigned)( nrh - nrl + 1 ) * sizeof( float** ));
   if ( !m )
      Exit_Message( "Matrix_3D:  allocation failure 1" );
   m -= nrl;

   for( i = nrl; i <= nrh; i++ )
   {
      m[i] = (float **) malloc( (unsigned)( nch - ncl + 1 ) * sizeof( float* ));
      if ( !m[i] )
         Exit_Message( "Matrix_3D:  allocation failure 2" );
      m[i] -= ncl;

      for( j = ncl; j <= nch; j++ )
      {
         m[i][j] = (float *) malloc( ( unsigned )( ndh - ndl + 1 ) * sizeof( float ));
         if ( !m[i][j] )
            Exit_Message( "Matrix_3D:  allocation failure 3" );
         m[i][j] -= ndl;
      }
   }

   return m;
}

/* end UTILITIES */
```

Bibliography

Barton, David K. *Modern Radar System Analysis.* Norwood, MA: Artech House, 1988.

Bogler, Philip L. *Radar Principles with Applications to Tracking Systems.* New York: John Wiley and Sons, 1990.

Bramley, M. and others. *User's Guide to RCS TOOLBOX, Version 2.0.* Lincoln Manual 168, Group 93, Massachusetts Institute of Technology, Lincoln Laboratory, Lexington MA, September 1991.

Brown, Joe R. and others. "Comparison of Two Neural Net Classifiers to a Quadratic Classifier for Millimeter Wave Radar," *SPIE 1294, Applications of Artificial Neural Networks.* 217-224 (1990).

Cohen, Marvin N. "A Survey of Radar-Based Target Recognition Techniques," *SPIE 1470, Data Structures and Target Classification*, edited by Vibeke Libby. 223-242 (1991).

DeWitt, Mark R. *High Range Resolution Radar Target Identification Using the Prony Model and Hidden Markov Models.* MS thesis, AFIT/GE/ENG/92D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

Farhat, Nabil H. "Microwave Diversity Imaging and Automated Target Identification Based on Models of Neural Networks," *Proceedings of the IEEE, 77(5)*: 670-680 (May 1989).

Libby, Edmund W. *Multisensor Data Fusion and Target Recognition.* Unpublished PhD prospectus. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, February 1992a.

-----. "Conversations with the Author Regarding High Range Resolution Radar Target Identification." (AFIT PhD student, Wright-Patterson AFB OH), September - November 1992b.

Lindsey, Randall L. *Function Prediction Using Recurrent Neural Networks.* MS thesis, AFIT/GE/ENG/91D-02. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

Mensa, Dean L. *High Resolution Radar Cross Section Imaging.* Norwood MA: Artech House, 1991.

Parsons, Thomas W. *Voice and Speech Processing.* New York: McGraw-Hill, 1987.

Robinson, Tony. "Conversation with the Author Regarding Recurrent Neural Network Applications." (electronic mail message from Cambridge University, United Kingdom), May 1992.

Sacchini, Joseph J. *Development of Two-Dimensional Parametric Radar Signal Modeling Estimation Techniques with Application to Target Identification.* PhD dissertation, Ohio State University, Columbus OH, 1992a.

-----. "Conversations with the Author Regarding High Range Resolution Radar Target Identification." (AFIT ENG professor, Wright-Patterson AFB OH), November 1992b.

Skolnik, Merrill I. *Introduction to Radar Systems*. New York: McGraw-Hill, 1980.

Stimson, George W. *Introduction to Airborne Radar.* El Segundo CA: Hughes Aircraft Company, 1983.

Trebits, Robert N. "Radar Cross Section," *Techniques of Radar Reflectivity Measurement*, edited by Nicholas C. Currie. Dedham MA: Artech House, 1984.

Williams, Ronald J. and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," *Neural Computation, 1*: 270-280 (1989).

Vita

Eric T. Kouba was born on 29 July 1965 in San Angelo, Texas. In 1982, he graduated from McCallum High School in Austin, Texas. Also in 1982, he entered the Air Force Reserve Officer Training Corps at the University of Illinois. In 1987, he graduated from the University of Illinois with a Bachelor of Science degree in Nuclear Engineering and received his commission. In October 1987, Captain Kouba was assigned to the Air Force Astronautics Laboratory, Edwards AFB, California, where he worked as a satellite power system engineer. In April 1991, he transferred to Space Command and entered the Graduate of Space Operations program at the Air Force Institute of Technology.

Permanent address: 7610 Rustling Road
Austin, Texas 78731

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE

RECURRENT NEURAL NETWORKS FOR RADAR TARGET IDENTIFICATION

5. FUNDING NUMBERS

6. AUTHOR(S)

Eric T. Kouba, Captain, USAF

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

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GSO/ENG/92D-02

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

Rick Mitchell
WL/AARA
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A real-time recurrent learning algorithm was applied to a five class radar target identification problem. The wideband radar was assumed to measure both kinematic (tracking information expressed as estimated aspect angles) and high range resolution data from a single, isolated aircraft. The aspect angles (azimuth and elevation) of the aircraft relative to the radar were assumed to be constantly changing. This created temporal sequences of high range resolution radar signatures that changed as the aspect angles changed. These sequences were used as input features to a recurrent neural network for three radar target identification test cases. The first test case demonstrated the feasibility of using recurrent neural networks for radar target identification. The second test case demonstrated the relationship between sequence length and target recognition accuracy. For the third test case, the recurrent net achieved 96% test set accuracy under the following conditions: 5 aircraft classes, azimuth range between 60° and 90°, elevation range between +5° and -5°, 1° signature granularity, and signatures corrupted by 5 dBsm scintillation noise.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Neural networks, Recurrent neural networks, Real-time recurrent learning algorithm, Radar target identification, Wideband radar, High range resolution radar, Temporal sequences, Sequence analysis	**95**
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	**Unclassified**	**Unclassified**	**Unlimited**