Design and simulation of advanced fault tolerant flight control schemes

Srikanth Gururajan
West Virginia University

Design and Simulation of Advanced Fault Tolerant Flight Control Schemes

Srikanth Gururajan

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in Aerospace Engineering

Department of Mechanical and Aerospace Engineering
Morgantown, West Virginia
2006

Keywords : Fault Tolerance, Neural Networks, SFDIA, AFDIA, Beowulf Cluster, Parallel Programming, AVDS

Abstract

Design and Simulation of Advanced Fault Tolerant Flight Control Schemes

Srikanth Gururajan

This research effort describes the design and simulation of a distributed Neural Network (NN) based fault tolerant flight control scheme and the interface of the scheme within a simulation/visualization environment. The goal of the fault tolerant flight control scheme is to recover an aircraft from failures to its sensors or actuators. A commercially available simulation package, Aviator Visual Design Simulator (AVDS), was used for the purpose of simulation and visualization of the aircraft dynamics and the performance of the control schemes.

For the purpose of the sensor failure detection, identification and accommodation (SFDIA) task, it is assumed that the *pitch*, *roll* and *yaw* rate gyros onboard are without physical redundancy. The task is accomplished through the use of a Main Neural Network (MNN) and a set of three De-Centralized Neural Networks (DNNs), providing analytical redundancy for the *pitch*, *roll* and *yaw* gyros. The purpose of the MNN is to detect a sensor failure while the purpose of the DNNs is to identify the failed sensor and then to provide failure accommodation. The actuator failure detection, identification and accommodation (AFDIA) scheme also features the MNN, for detection of actuator failures, along with three Neural Network Controllers (NNCs) for providing the compensating control surface deflections to neutralize the failure induced *pitching*, *rolling* and *yawing* moments. All NNs continue to train on-line, in addition to an offline trained baseline network structure, using the Extended Back-Propagation Algorithm (EBPA), with the flight data provided by the AVDS simulation package.

The above mentioned adaptive flight control schemes have been traditionally implemented sequentially on a single computer. This research addresses the implementation of these fault tolerant flight control schemes on parallel and distributed computer architectures, using Berkeley Software Distribution (BSD) sockets and Message Passing Interface (MPI) for inter-process communication.

Dedicated to my Family

Acknowledgements

I would like to begin by thanking my parents and my sister for their unwavering support, love, encouragement and patience during all the years I was working on my dissertation. Thank you for putting up with me, I know I have been difficult to handle sometimes. I would like to thank my wife for being understanding, encouraging and patient with me through thick and thin. Without all your support, I would not be where I am today. I love you all.

I would like to thank my committee chairman and research advisor Dr. Marcello Napolitano for his mentoring, guidance, and support throughout my doctoral program. I am grateful to you for giving me the opportunity to work on varied and challenging research opportunities. You have always been there to help me and show me the way. Thank You.

I would like to acknowledge and thank my committee members Dr. Bojan Cukic and Dr. Giampiero Campa, for their invaluble advice, guidance and support throughout my doctoral program.

I would like to thank Dr. Gary Morris and Dr. Wade Huebsch for taking time from their busy schedules to review and contribute their thoughts towards this research effort.

I would like to extend my heartfelt thanks to my colleagues in the lab Dr. Brad Seanor, Dr. Yu Gu, Larry Rowe, Josh Effland and Jason Jarell for all their help and thoughtful inputs to my dissertation. All the time we spent arguing back and forth about important and mundane issues made life more lively and interesting.

I would like to thank John Mathews and Eric Saunders for all their help in setting up my cluster. I would also like to thank Chuck Coleman, David Estep, Lee Metheney, and Clifford Judy from the MAE department for their help with equipment and other issues.

Last but not the least, I would like to thank my friends, especially Bala and Sushant, who were there to support and encourage me when I needed it most. Thank you very much.

Nomenclature

English

a	Acceleration, ft/sec^2
a	Node in input layer
b	Node in hidden layer
b	Wing span, ft
B	Node in hidden layer
c	Node in output layer
$\bar{c}$	Mean aerodynamic chord, ft
C	Node in output layer
d	Target Neural Network output
e	Neural Network training error
g	Acceleration due to gravity, ft/sec^2
J	Error cost function
I	Moment of inertia, slug-ft^2
k	Discrete time index
l	Pattern for the neural network input data
L	Rolling moment
L	Lower bound of modified sigmoid activation function
m	Aircraft mass, slugs
M	Pitching moment
N	Yawing moment
p	Aircraft angular velocity, x body axis (roll rate), rad/sec
q	Aircraft angular velocity, y body axis (pitch rate), rad/sec
r	Aircraft angular velocity, z body axis (yaw rate), rad/sec
R	Auto or cross correlation function
S	Wing plan form area, ft^2
t	Time, sec
T	Slope of modified sigmoid activation function
u	Velocity along body fixed x-axis, ft/sec
U	Upper bound of modified sigmoid activation function
v	Velocity along body fixed y-axis, ft/sec
V	Interconnection weight vector between input and hidden layer nodes
w	Velocity along body fixed z-axis, ft/sec
W	Interconnection weight vector between hidden and output layer nodes
y	Neural Network output

Greek

α	Angle of attack, rad or deg
α	Neural Network momentum rate
β	Angle of sideslip, rad or deg
δ	Control surface deflection, rad or deg
δ	Neural Network output and hidden layer error term
Δ	Error update term
η	Neural Network learning rate

θ	Pitch Euler angle, rad or deg
Θ	Neural Network hidden layer neuron threshold
ρ	Density of aircraft
ρ	Magnitude of sensor failure
Γ	Neural Network output layer neuron threshold
ϕ	Roll Euler angle, rad or deg
ψ	Yaw Euler angle, rad or deg

Subscripts

A	Aileron
A	Aerodynamic
E	Elevator
h	Index of nodes in input layer
i	Index of nodes in hidden layer
m	number of nodes in hidden layer
n	number of nodes in input layer
p	number of nodes in output layer
L	Left side
R	Right side
R	Rudder
T	Thrust
x	Along the x-direction
y	Along the y-direction
z	Along the z-direction

Vectors

O	Neural network output
Y	Parameters to be estimated

Acronyms

AFA	Actuator Failure Accommodation
AFDI	Actuator Failure Detection and Identification
AFDIA	Actuator Failure Detection, Identification, and Accommodation
AVDS	Aviator Visual Design Simulator
BPA	Back Propagation Algorithm
BSD	Berkeley Software Distribution
COTS	Commodity Off The Shelf
DAEE	Decentralized Neural Network Error Estimation Mean
DVEE	Decentralized Neural Network Error Estimation Variance
DNN	De-Centralized Neural Network
DQEE	De-Centralized Quadratic Estimation Error
EBPA	Extended Back Propagation Algorithm
FCS	Flight Control System
FDI	Failure Detection and Identification
GR	Generalized Likelihood Ratio
GUI	Graphical User Interface
MIMD	Multiple Instruction/Multiple Data
MIMD	Multiple Instruction/Single Data

MLP	Multi Layer Perceptron
MMKF	Multiple Model Kalman Filter
MNN	Main Neural Network
MPI	Message Passing Interface
MQEE	Main Quadratic Estimation Error
NN	Neural Network
NNC	Neural Network Controller
OQEE	Output (of NN) Quadratic Estimation Error
PVM	Parallel Virtual Machine
ROC	Receiver Operating Characteristic
SFA	Sensor Failure Accommodation
SFDI	Sensor Failure Detection and Identification
SFDIA	Sensor Failure Detection, Identification, and Accommodation
SISD	Single Instruction/Single Data
SIMD	Single Instruction/Multiple Data
UDSCB	User Defined Simulation Code Block

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

Operational safety is of paramount importance for both commercial and military aircraft and more recently for Unmanned Aerial Vehicles. Over the last decade a significant research effort has been directed towards increasing operational safety by developing fault tolerant flight control schemes that would assist the pilot in recovering the aircraft from failures to sensor systems and more importantly, structural damage to the onboard actuator systems.

The idea of *Sensor Failure Detection, Identification and Accommodation* (SFDIA) is an important concept in the implementation of fault tolerant flight control algorithms, especially if measurements from the onboard sensors are used in the feedback loop of flight control laws. In fact, if failures in the sensors used in feedback loops are left undetected and unaccounted for, they may lead to closed loop instabilities and even catastrophic flight conditions. To avoid this problem, most modern commercial and military aircraft feature triple physical redundancies on their sensors and use polling schemes to obtain correct sensor values. Analytical redundancy of sensor systems on-board such aircraft could lead to a reduction in the extra weight attributed to the physical redundancy on sensors, effectively increasing the payload capacity of the aircraft.

Aircraft actuator systems on board typically include drive servos for the control surfaces, landing gear, hydraulic subsystems and throttle. *Actuator Failure Detection Identification and Accommodation* (AFDIA), refers to failures on the on-board actuator systems. Failures on actuator systems for elevators, ailerons or rudder, could involve the surfaces being stuck at some angular deflections. Actuator failures also include the scenarios wherein there is a physical loss of either a part of or the entire control surfaces, resulting in a loss of efficiency of the failed control surface and/or a combination of the two.

Typically, actuator failures result in changes in the aircraft trim conditions leading to dynamic coupling between lateral and longitudinal modes, and could potentially lead to unrecoverable flight conditions. Although, some aircraft might be equipped with redundant actuator systems that could handle failures on drive servos, typically, aircraft do not have physically redundant control surfaces, unlike redundant sensor systems. Therefore, the need for analytical methods to compensate for failures on any of the control surfaces is much more critical.

The fundamental task of a fault tolerant flight control system is the detection, identification, and accommodation of the previously mentioned sensor and actuator failures. The task of dealing with sensor failure is delegated to the SFDIA scheme, which can be split into two distinct parts as *Sensor Failure Detection and Identification* (SFDI), designed to monitor the sensor systems without physical redundancy for signs of failure and the corresponding identification and isolation of such failures. Typical failures on sensor subsystems include the introduction of a sudden bias in the sensor outputs, flat line sensor outputs at magnitude, or slow moving drift in the sensor outputs. The *Sensor Failure Accommodation* (SFA) phase of the task would be responsible for taking over the function of the failed sensor, providing analytical redundancy and alternate inputs to the flight control system to accommodate for the failed sensor.

Similarly, the task of dealing with actuator failures is split into two distinct sections as *Actuator Failure Detection and Identification* (AFDI), which detects and identifies any failure on the actuators and the *Actuator Failure Accommodation* (AFA) phase, which would determine the control actions needed to compensate for the identified actuator failure and restore the aircraft back to a stable flight condition with acceptable handling characteristics. In this research effort, actuator failures refer to physical failures on the control surfaces on an aircraft with or without a loss of control surface. The issues of failure detection, identification, and accommodation on the electrical, electro-mechanical or hydraulic subsystems are not addressed. Following control surface failures and subsequent identification, the accommodation phase is designed to be achieved by making use of the remainder of the functional and healthy control surfaces.

Failure detection and accommodation schemes based on theoretical schemes such as the *Generalized Likelihood Ratio* [18] and *Multiple Model Kalman Filtering* (MMKF) [18] have been utilized in the past, but have a significant drawback in that the applicability of such schemes are limited to linear time invariant systems, with the system model identical to the Kalman filter or the observer model.

An aircraft can be considered as a time varying, non-linear system with system and measurement noise. Based on this characteristic, different approaches have to be taken in designing the failure detection and accommodation scheme. One of the approaches is the use of Artificial Neural Networks (ANN), which are capable candidates for the task, mainly due to their inherent non-linear nature and learning and adaptation capabilities. The selection of ANN as the basis of the adaptive scheme comes with its own advantages and limitations in that the networks can be trained either off-line, with pre-recorded flight data, or on-line with currently available flight data. With off-line learning and no subsequent updating of the network architecture with current data, a network would have a frozen architecture once training cycles are completed, thereby losing the advantage of learning from the newer on-line information. If such a network is implemented in the adaptive scheme, it might not be able to detect, identify or accommodate for a failure situation it had not been trained for, making the entire approach inadequate. This tilts the balance in favor of training a network on-line; however in this situation, care needs to be taken when implementing such a scheme as on-line learning has its own set of advantages and disadvantages. The obvious advantage of on-line learning is that the network can adapt itself to the real-time data from the aircraft, giving it a better knowledge of current conditions. On the other hand, the issues of concern are the amount of training that would be necessary for a network to reach acceptable levels of learning, the time required to do so and the complexity of network structure. There is also an increased potential for on-line learning to drive the neural network unstable, leading to catastrophic conditions. Therefore, on-line training of NN for use in an adaptive scheme for fault tolerance in flight control systems should at the least be a compromise between learning accuracy and the total time taken for achieving the required level of accuracy.

In recent years, there has been an increase in the use of Unmanned Aerial Vehicles (UAV) in both civilian as well as military sectors for a wide range of applications including aerial reconnaissance of military targets, monitoring traffic, geological mapping, search and rescue among others. Typically with the use of UAVs, there is a premium on total weight and cost, in order to maximize their payload potential and as a consequence, the use of redundant sensors is uncommon.

With the added interest in UAVs, significant amount of research effort has been directed towards practical implementations of experimental fault tolerant flight control schemes. Unmanned Aerial Vehicles and Micro Aerial Vehicles have been the more popular platforms of choice for such experimental research programs. UAV/MAVs provide flexibility and a distinct advantage as compared to manned aircraft in terms of cost, weight and time and human factors. They are not expensive, do not take much time to build, and are easier to rebuild, easier to transport and require minimal to no certification from the Federal Aviation Administration.

Typically, these aerial platforms feature Commodity, Off The Shelf (COTS) computational platforms as well as sensor and actuator systems. Due primarily to the strict limitations on the power available for such onboard systems, the computational power of the on-board processors are much less than the traditional power available on desktop computational platforms. As a result, the extent of the complexity of NN based fault tolerant flight control schemes is severely limited. A workaround for this computational bottleneck is to take the “divide and conquer” approach. By making use of a computational cluster, made of COTS components, with lower individual computational capacity and a smaller appetite for power consumption, more complex fault tolerant flight control schemes can be implemented. This research effort addresses this issue by decomposing a fairly complex sequential implementation of a NN based fault tolerant flight control scheme onto a cluster of interconnected computers and executing it in parallel.

1.1 Research Objectives

The objectives of this research effort are to design and simulate Neural Network based *Sensor Failure Detection Identification and Accommodation* (SFDIA) and *Actuator Failure Detection Identification and Accommodation* (AFDIA) schemes, within an aircraft simulation and visualization environment. Furthermore, the NN based fault tolerant flight control schemes are implemented on a distributed computation platform, based on a Beowulf cluster of individual commodity computers.

The SFDIA scheme focuses on providing analytical redundancy to three important sensors onboard an aircraft, namely the *pitch*, *roll* and *yaw* rate gyros. It is assumed that these rate gyros onboard the aircraft are without physical redundancy and the NNs are designed to provide the requisite analytical redundancy. It is also assumed that all the other sensor systems on board are reliable and perform without failures of any kind, for the purposes of this research, although this concept of analytical redundancy can be extended to any number of sensor systems without any loss of generality.

The AFDIA scheme will focus on providing fault tolerance following failures on two sets of primary control surfaces, namely the elevators, and the ailerons. For the purpose of this research, it is assumed that the airplane does not have any unconventional control surfaces such as differential or symmetric canards, thrust vectoring, differential stabilators, etc.

This research effort is broken into different phases as described below:

1. Initial development of the neural network based adaptive control schemes;
2. Design and development of an interface within the Matlab environment to provide flexibility in the creation of the NN structures for the fault tolerant flight control schemes;
3. Design and implementation of a communication and data transfer scheme between the Matlab interface, the graphics visualization package, which allows the user to fly an aircraft model interactively, and the distributed implementation of the fault tolerant flight control scheme;
4. Implementation of the fault tolerant flight control schemes on the Beowulf cluster based parallel computational platform;

A more detailed description of the different phases of this research is provided below.

Phase #1:

This phase deals with the initial development of the NN based adaptive flight control algorithms for the purposes of *SFDIA* and *AFDIA*. This included the selection of the aircraft model, the learning algorithm for the NN, the number of individual networks to train for each sensor without physical redundancy (*pitch*, *roll*, *yaw rate*), the selection of inputs to each of the network structures, the failure detection and identification scheme, and finally the failure accommodation scheme.

Within this phase, the overall structure of the SFDIA scheme was designed to have a Main Neural Network (MNN) and a set of three Decentralized Neural Networks, DNNs, for each of the three rate gyros, *roll*, *pitch* and *yaw* rate gyros, assumed to be without physical redundancy, and named as the P-DNN, Q-DNN and the R-DNN respectively. The individual DNNs are designed to produce estimates of the respective rate gyros only, while the MNN produces the estimates of all the three rate gyros. The errors between the estimates from the DNN and the MNN are continuously monitored and provide a means for detecting and identifying failures on the sensors as well as accommodating for such failures.

The structure of the AFDIA scheme was built similar to the SFDIA scheme, with the MNN designed to produce the estimates of the *roll*, *pitch* and *yaw* rate gyros. Three decentralized neural networks, named as the *roll-controller*, *pitch-controller* and the *yaw-controller* produce estimates of the ailerons, elevators and rudder deflections. When a failure is injected on either the elevator or the aileron, it produces a measurable change in the auto and cross- correlation terms, which are then used to detect and identify the failures. Upon detection and identification of an actuator failure, these NNCs provide the compensating deflections to the remainder of the “healthy” control surfaces, to accommodate for the failure.

Phase #2:

Within this phase, a Graphical User Interface (GUI) was designed in Matlab, to provide the user with an interactive tool to choose the parameters that define the structure of the NNs, and set the various parameters associated with the SFDIA, AFDIA and the SFDIA+AFDIA schemes. This GUI tool gives the grater flexibility to choose the input variables to the NNs, change the time history window of the inputs, the learning and momentum rates for each network and the number of hidden layer neurons. These parameters are typically hard coded into any implementation of MLP based NN based schemes, with flexibility limited to selecting the learning and momentum rates and the number of neurons in the hidden layers.

Phase #3:

Phase 3 of the research study involved the design and implementation of a data communication and exchange structure to facilitate the flow of information between the Matlab GUI, the graphical simulation/visualization environment and the Beowulf cluster based implementation of the fault tolerant flight control schemes. A commercially available simulation/visualization package, Aviator Visual Display Simulator (AVDS) is chosen for this purpose. The choice of AVDS over others such as X-Plane, Microsoft Flight Simulator was influenced primarily by the vast experience in using this package among various members of our research group. This phase also involved the design and implementation of a Berkeley Software Distribution (BSD) socket based communication channel and data exchange scheme, between the NNs and the AVDS environment to facilitate two way data transfer between the nodes executing the network learning and the node executing the simulation/visualization. The interface between the Beowulf cluster and the AVDS environment was designed to make use of TCP/IP sockets. AVDS package runs on a Microsoft Windows 2000 based computer, designated as “*node8*” with a private IP address of 192.168.1.8/255.255.255.0. The primary node in the cluster named as “*node1*” is designated as the master node and has an IP address of 192.168.1.1/255.255.255.0 with the other nodes having IP addresses in the order

192.168.1.2/255.255.255.0 – 192.168.1.6/255.255.255.0 and named as "*node2*", "*node3*", "*node4*", "*node5*", "*node6*" and "*node7*". Message Passing Interface is used to facilitate communication and synchronization between the nodes on the cluster. Chapter 5 discusses the setup of the cluster and the communication schemes in detail.

Phase #4:

In Phase 4, a sequential execution mode was built and tested for training the NNs with the on-line data from AVDS environment. This served as a testbed for validating the communication schemes between the Matlab interactive GUI, the Beowulf cluster and the AVDS environment, which was designed and implemented in Phase 3. This process involved the implementing the NNs in a sequential execution mode, with all the NN modules in the scheme executing on the same local machine with AVDS running on a dedicated computing node. A TCP/IP socket based communication channel is used to transfer data from AVDS simulation environment into the dedicated computing node on which the NNs are implemented. The NNs that were used for this training purpose were the MNN, P-DNN, Q-DNN and the R-DNN. Within this structure, the order of execution of the NNs was hard coded with the MNN executing first, followed by the P-DNN, Q-DNN and the R-DNN, for each frame of data received from AVDS.

The structure of the sequential execution of the SFDIA/AFDIA algorithm leads to intuitive decomposition of the tasks for parallel implementation. The next part of phase 4 involved analyzing the sequential SFDIA/AFDIA algorithm and identifying the tasks that could be parallelized. The algorithm was then broken up into fairly independent and separate tasks that could be executed in parallel, dealing primarily with the issues of data transfer and synchronization between the individual tasks.

The initial setup of this parallel implementation is similar to that of the sequential implementation, with the structure of the NNs set using the Matlab based GUI. This information about the structures of the NN is then sent to the computer supporting the MNN, the master node, using TCP/IP sockets. The master node then sends this data to all the other computers on the cluster, designated as the slave nodes, using MPI directives. The DNNs for both the SFDIA/AFDIA purposes are then created using the corresponding information. After the initialization phase, the MNN sends across the parameters of the simulation to AVDS, using a separate TCP/IP socket. Thus during each simulation run, there are two TCP/IP sockets open, one connecting the master node to Matlab and the other connecting the master node to AVDS. Once the simulation starts, AVDS sends back frames of data at a pre-specified frequency to the MNN, through socket and MNN consequently distributes this data frame to all the slave nodes in the cluster, using MPI directives. The MNN as well as the individual nodes then begin executing the NN code, the outputs of which are communicated back to the MNN using MPI directives. The MNN then computes the corresponding training error and performs the Failure Detection, Identification and Accommodation (FDIA) functions. In case of actuator failures, the compensating control surface deflections, as computed by the AFDIA scheme are then transmitted back to AVDS simulation using the appropriate socket.

This document is organized as follows. Chapter 2 describes approaches/methods adopted by other researchers to address the issues of fault tolerance in flight control systems, with respect to failures on both the sensor and actuator systems. Chapter 3 describes the AVDS simulation/visualization environment; Chapter 4 presents the derivation of the aircraft equations of motion, Neural Network theory, the Extended Back-Propagation Algorithm and background information on parallel computing. Chapter 5 describes the design and development of the structure of the sequential and parallel implementations. Chapter 6 describes the SFDIA and AFDIA algorithms and the results from the parallel implementations on the Beowulf cluster.

Chapter 2 - Literature Review

The task of recovering from failures to various sub-systems on-board an aircraft and the associated issues of detecting, identifying and accommodating them has serious implications on the performance of airplanes, both manned and unmanned. Considering the avionics, power, engine, hydraulics and communication sub-systems to be fully functional and free from failure, the on-board failures can be classified into two broad categories as sensor and actuator failures.

Typical military and commercial aircraft have multiple redundant sensor systems and use polling methods, among other schemes to get the actual sensor measurements. While this approach is the traditional safeguard against failed sensor systems, restrictions on the payload capacity, the power available on board aircraft and the desire to reduce complexity has spurred researchers towards the development of analytical redundancy wherein computational methods are used to provide estimates of the outputs of the sensor systems without analytical redundancy. This approach is attractive especially when applied to low cost UAVs that in recent years have gained wide acceptance in applications like surveillance and monitoring, search and rescue, reconnaissance among others.

This section will give a brief overview of techniques developed by other researchers for sensor failure detection, identification and accommodation. Typically, a sensor failure detection scheme involves observing the measurements from sensors, which under nominal conditions follow known and predictable patterns in the presence of system and measurement noise. Conventional failure detection mechanisms are based on hardware redundancy, using a voting scheme [16, 17], where the measurements from three identical sensors are compared to each other. If the measurement metrics from one of the redundant sensors is significantly different from the other two, it is then classified as failed and thus removed from consideration among the redundant sensors and a statistical mean of the measurements from the other two sensors is then used as representing the sensor output in the flight control system. Such a voting scheme is simple to implement and capable detecting hard failures fast, but has the drawback of increased hardware complexity, power requirements, and increased weight on board the aircraft. These disadvantages render this scheme unsuitable in situations such as low cost and low weight UAVs, where reduced cost, power and weight are critical factors. Analytical redundancy presents itself as an attractive alternative in such situations. Analytical redundancy refers to the use of mathematical models of the sensors to estimate their outputs. The estimates of the actual signals that are computed by these analytical methods are then compared against the sensor outputs, generating an error signal, which can then be used to signal a failure.

Analytical approaches to failure detection and accommodation have included the use of *Multiple Model Kalman Filtering* (MMKF) [17, 18] with ‘*n*’ observation models generated offline and stored in the onboard computer. Of the ‘*n*’ models, ‘*n-1*’ represent different types of sensor failures and one represents the nominal condition. Monitoring the Bayesian probability statistics of the outputs from all the models and checking which of the models matches with the actual output, provides failure detection and identification of the failure mode. The drawback of this method is that its effectiveness depends on a pre-determined set of failure models, making it less effective in case of failures that are not previously modeled. Furthermore, this is based on the operation of a Kalman filter and is limited to a linear model of the aircraft, making it sensitive to modeling errors. NNs are interesting analytical tools as they can be applied to systems that are either linear or non-linear. Over the last decade, several schemes have been proposed using NN as a tool for FDI [19-25]. An adaptive NN augmented observer by Sreedhar [21] uses a sliding mode observer to characterize the un-modeled dynamics to train the neural network while a new approach for sensor failure detection identification and accommodation using a neural network by Guo and Nurre [19] is useful when the relationship between the sensors is too complex to be modeled by a Kalman filter estimator. The use of NN for the system identification and control of non-linear systems is discussed in Narendra [22,23]. Also, Polycarpou [24,25] discusses systematic procedures for constructing non-linear estimation algorithms and stable learning schemes using the Lyapunov synthesis approach. The technical literature [26-35] discusses the use of neural networks in aircraft flight control.

Previous work at West Virginia University [33-39] has addressed the issue of sensor and actuator failure detection, identification and accommodation, using various approaches including Neural Networks, Fuzzy Logic, Kalman predictor and Eigenstructure Assignment. An [1] addresses the issue of improving the NN based SFDIA schemes for detecting soft failures which are characterized by accumulation of errors in the sensors over a period of time, and also the integration of the sensor failure and actuator failure scenarios. This task was divided into two sub-divisions, namely Sensor Failure Detection and Identification (SFDI), to monitor the extent of deterioration of the sensor systems and Sensor failure accommodation (SFA), to deal with the accommodation of the failed sensor by means of the NN scheme. Similarly, the actuator failure section is divided into two tasks as Actuator Failure Detection and Identification (AFDI), monitoring the extent of the cross coupling of the longitudinal and lateral direction dynamics, a situation that indicates a loss of symmetry corresponding to an actuator failure and Actuator Failure Accommodation (AFA) task, which deals with the controller actions to recover from failure. The SFDIA scheme is important particularly when the inputs from the sensors are used in feedback loops of the flight control systems. If left undetected and uncompensated, such inaccuracies would eventually build up in the system, resulting in closed loop instability and potentially unrecoverable flight conditions. Current solutions to overcome this problem have been to have multiple physical redundancies, typically triple physical redundancy, but this has its own drawbacks in terms of the weight of the redundant sensors, the power required to maintain them and increased hardware complexity.

Although accommodating for sensor failures is an important issue, the task of identifying and accommodating for failures to the actuator systems, particularly the control surfaces on aircraft take a higher precedence. Actuator failures during flight more often than not lead to catastrophic and unrecoverable flight conditions. Since there are limitations on the forms of physical redundancy that can be incorporated into an aircraft to compensate for actuator failures, software based approaches that attempt to compensate for the failures using the remainder of the healthy control surfaces have been extensively studied in recent years. Aircraft dynamics are non-linear in nature, especially after failures on the actuator systems. Solutions to the control of such non-linear systems has been to decompose the non-linear system into a number of linear systems, around various operating points, allowing for control schemes using gain scheduling. On-line parameter estimation based gain scheduling approaches [51] have been used for implementing fault tolerant flight control systems.

Gain scheduling approach to fault tolerance relies on the reconfiguration and re-computation of controller gains for the predetermined set of linear models. The dependence on a set of predefined models and the extensive computational burden in the redesign of the controller gains is a drawback for this method.

In recent years, several experimental programs have addressed the issue of fault tolerance through flight testing. The Reconfigurable Control for Tailless Fighter Aircraft (RESTORE) [52-54] program and the Intelligent Flight Control System (IFCS) [55-65] are two examples of many such experimental programs. The purpose of the RESTORE program was the development and evaluation of reconfigurable flight control algorithms. Unlike traditional control systems, the control laws within the RESTORE project were designed to compensate for unknown failures on aircraft control surfaces by adapting themselves to the changing aircraft dynamics. The controller designed by the Boeing team, within the RESTORE project is designed to be modular and is based on dynamic inversion and the neural network designed to regulate the inversion errors between the pre-determined aircraft model and the true aircraft dynamics. This error in inversion could be caused by an actuator failure or damage to the aircraft structure. The response from the control algorithms to compensate for the failures onboard the aircraft was distributed to the remainder of the functional surfaces by means of a control allocation module. Two flights were flown in December 1998 proving the feasibility of the adaptive control approach. The Tailless Fighter Aircraft X-36 is shown below in Figure 2-1.

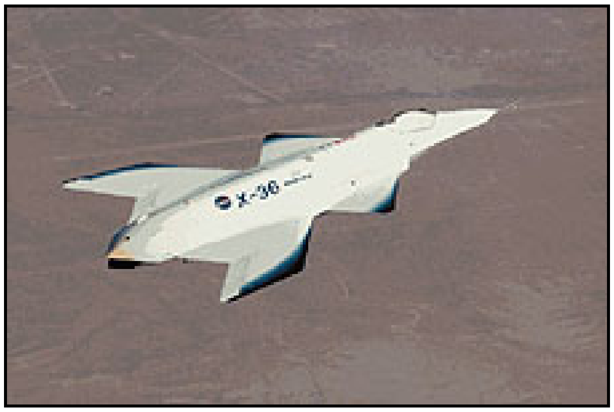


Figure 2-1 : Tailless Fighter Aircraft X-36

The modular control architecture of the RESTORE project is as shown in the following figure.

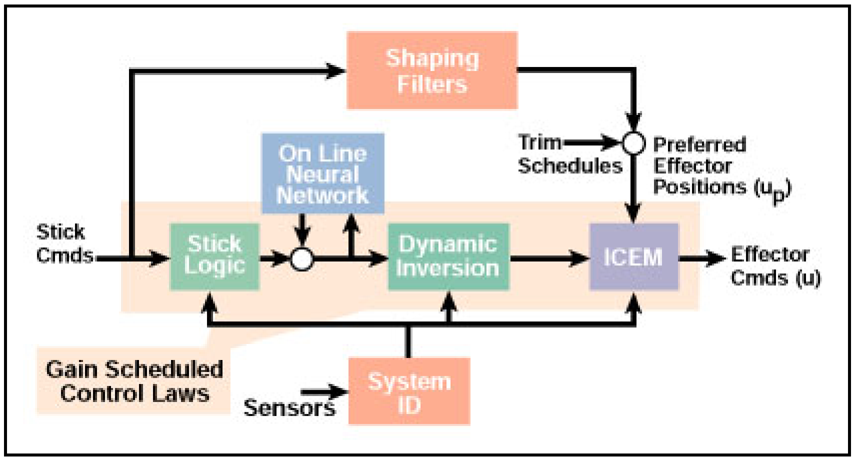


Figure 2-2 : Boeing RESTORE Controller

The Intelligent Flight Control System was established at NASA Dryden Flight Research Center to design aircraft flight controls that can optimize aircraft flight performance characteristics under both nominal and failure modes. The IFCS is designed to use the self learning concepts of NNs into flight control software to enable the pilot to maintain control of aircraft with major systems failure or combat damage. The first generation of the IFCS, referred to as the GEN-1 approach is based on a direct adaptive neural network based flight control system. The Gen-1 controller was designed to estimate the changes in the aircraft aerodynamics as a result of failure modes. This information is provided to the flight control scheme which then identifies the stability and control derivatives of the aircraft, as an indicator of the aircraft dynamic characteristics and is then used to stabilize the aircraft. Flight tests of the Gen-1 controller were flown onboard a test-bed aircraft, the NASA NF-15B (NASA 837) shown below in Figure 2-3.

NASA Dryden Flight Research Center Photo Collection
http://www.dfrc.nasa.gov/gallery/photo/index.html
NASA Photo: EC02-0281-2 Date: December 6, 2002 Photo by: Carla Thomas
NASA Dryden's highly modified F-15B, tail number 837, seen here with the Boron Mine as a backdrop, resumed Intelligent Flight Control System (IFCS) project flights on Dec. 6, 2002.

Figure 2-3 : IFCS Gen-1 Test bed Aircraft (NASA 837)

The general block diagram of the IFCS Gen-1 controller is shown below in Figure 2-4.

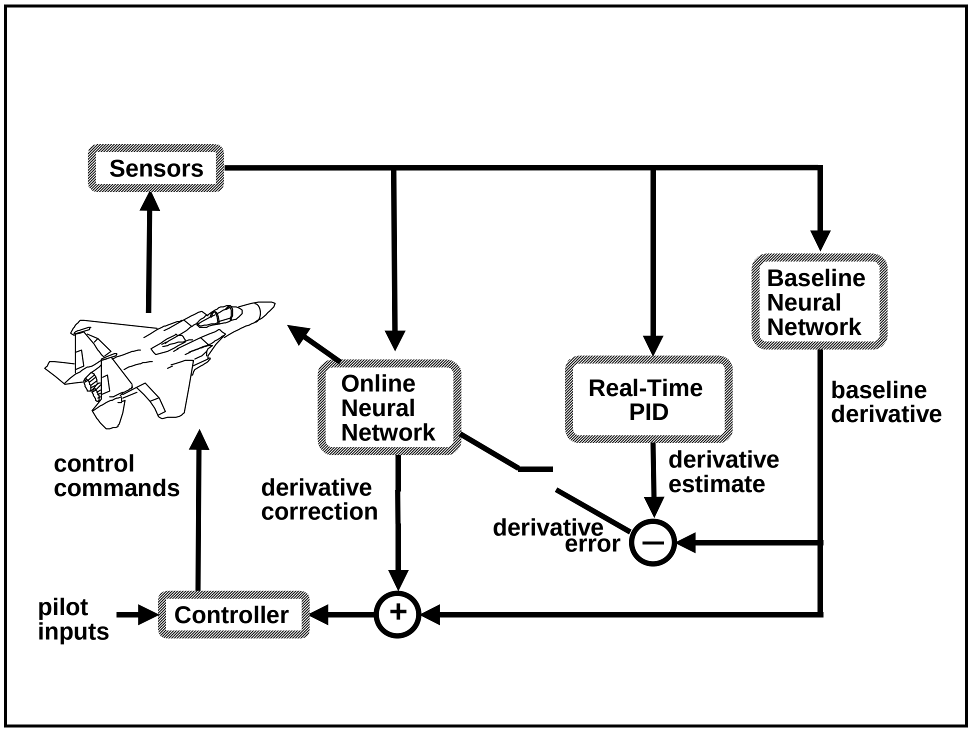


Figure 2-4 : Architecture of IFCS Gen-1 Controller

The IFCS Gen-2 controller concept is based on a dynamic inversion controller with a model following command path. The general block diagram of the Gen-2 IFCS flight controller is shown in Figure 2-5.

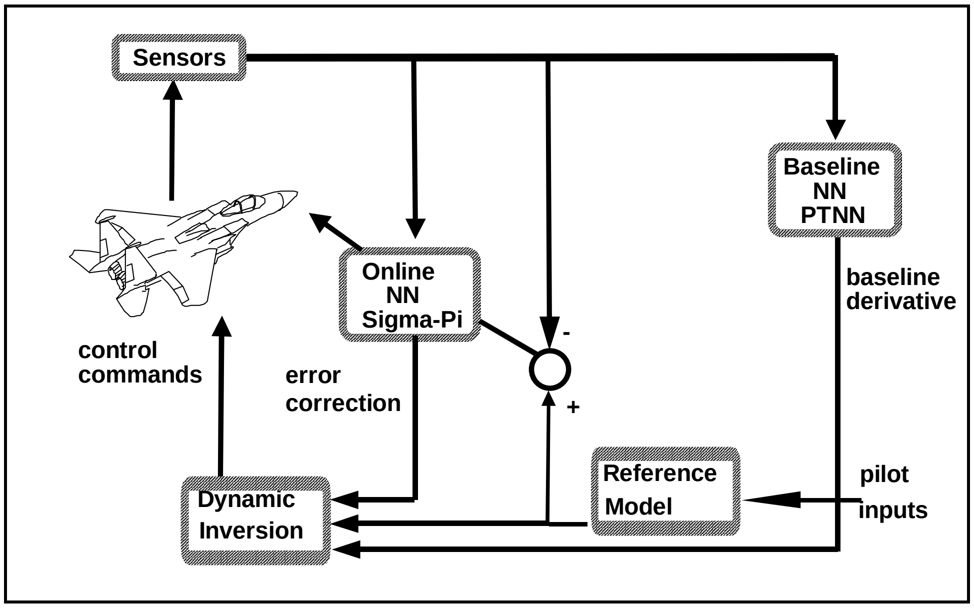


Figure 2-5 : Architecture of IFCS Gen-2 Controller

The feedback errors that are generated are controlled with a Proportional + Integral (PI) controller and this basic system is augmented with an adaptive NN that operates directly on feedback errors. The NN then adjusts the system for the erroneous behavior or changes in the aircraft characteristics after failure. The dynamic inversion part of the control system produces acceleration commands that are translated into control surface deflections by a control allocation scheme. Initial PID test flights with IFCS NNs pre-trained to the F-15's aerodynamic database was test flown in the spring of 1999. It must be noted that a portion of the current flight testing activities at West Virginia University is directed towards the implementation of the Gen-2 controller scheme on the WVU YF-22 UAV research platform.

Flight controls research at West Virginia University has addressed the issues of flight testing fault tolerant flight control laws [50]. Gu describes the "design development and flight testing of a Neural Network based Fault Tolerant flight Control System to accommodate for actuator failures". Within the research effort, the author demonstrates the ability of a set of flight control systems to maintain aircraft handling qualities in the presence of failures on actuators. The author describes actuator failures on the Ailerons as the Right Aileron locked at Trim position and the failures on the Elevator as the Right Elevator locked at Trim position. The test bed aircraft used in that research effort is shown in Figure 2-6.

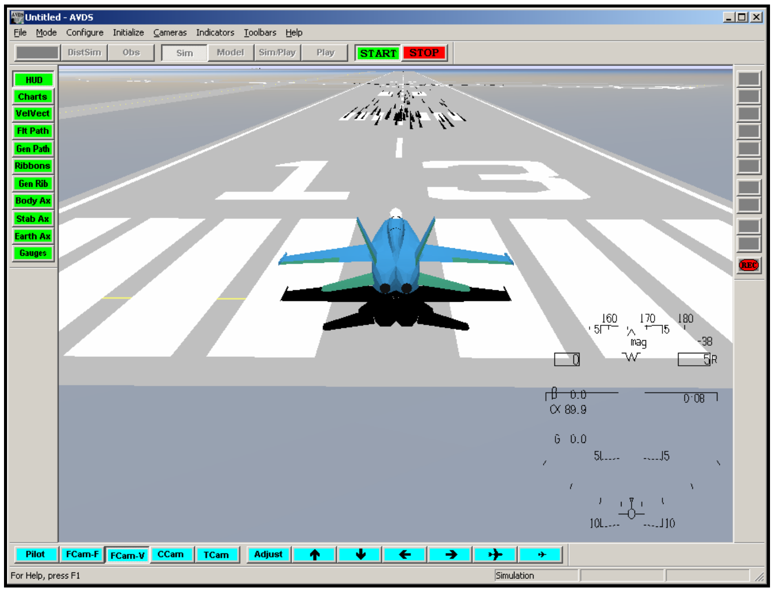
Figure 2-6 : WVU YF-22

Conventionally, NN based fault tolerant flight control schemes are implemented on a standalone computational platform, although the inherent structure of the NN makes it a prime candidate for implementation on parallel computer architectures. Implementing the neural networks in parallel is effective in reducing the training time and also paves the way for incorporating more complex training algorithms. Recent work [40,41], among many others, has addressed the topic of implementation of NN on parallel computers by splitting the network structure into component modules and executing them concurrently. This research effort attempts to decompose a sequential implementation of a NN based fault tolerant flight control scheme into a set of high level modules, with each module executing an independent task. This differs from the network level decomposition approach of other researchers [40,41].

Chapter 3 - Visualization Environment

3.1 Aviator Visual Design Simulator (AVDS)

An important segment of this research is the use visualization environment to graphically display the results of the simulation. The demands on the visualization environment are that it should be able to render three-dimensional images of the aircraft and at the same time be capable of supporting an interactive simulation, accepting user-generated inputs via a joystick. The Aviator Visual Design Simulator (AVDS) is a commercially available flight simulation package [6] that satisfies the above requirements and provides desirable configuration options such as user definable aircraft models and dynamics. The AVDS software package is a product of a joint research effort between Artificial Horizons Inc. and the United States Air Force and is a set of PC based simulation and visualization tools, designed for use by students, teachers, engineers and designers for aircraft research and development purposes. A typical AVDS simulation/visualization screen is shown in Figure 3.1.



Untitled - AVDS
File Mode Configure Initialize Cameras Indicators Toolbars Help
DistSim Obs Sim Model Sim/Play Play START STOP
HUD
Charts
VelVect
Flt Path
Gen Path
Ribbons
Gen Rib
Body Ax
Stab Ax
Earth Ax
Gauges
Pilot FCam-F FCam-V CCam TCam Adjust
For Help, press F1
Simulation

Figure 3-1 : AVDS Simulation/Visualization Environment

Some of the appealing features of the AVDS simulation package are:

- AVDS can be used interactively to simulate aircraft design and provide animated results of batch simulations;

- AVDS can be used for testing flight control systems design with the user-definable interactive simulation model;
- AVDS contains user definable aircraft images including articulated control surfaces;
- AVDS is designed to accept user defined aircraft models, by means of three-dimensional lookup tables;
- AVDS is capable of interactive simulation with external hardware inputs and hardware-in-the-loop simulations;
- AVDS is capable of distributing simulation and visualization tools over a network of computers, forming a rudimentary distributed computing and simulation network.

The recommended system configuration for supporting the AVDS simulation package are listed below:

- Pentium^R processor based personal computer;
- Microsoft^R Windows 95/98, WinNT^R or later operating system;
- A graphics accelerator;
- At least 32 MB RAM;
- At least 110 MB of disk space.

3.2 Architecture of AVDS

This section describes the architecture of the AVDS simulation environment as installed on a Microsoft Windows based personal computer. When AVDS is installed, it creates seven sub directories under the installation directory. The following section describes each of the seven directories and its file contents.

- "**bin**" directory

  The "bin" directory contains three files, namely, "**avds.exe**", "**avds.std.ini**" and "**avds.ini**". The avds.std.ini is a text file written in the standard windows "ini" format, and is used to store the multicast addresses, while the avds.ini file contains the file and path names for the AVDS setup files.

- **"craft**" directory

This directory contains a file named “**craftcap**”, that is used to identify the aircraft image definition files and sample aircraft image definition files. This enables the user to change the appearance of existing aircraft images and add new aircraft image files.

- **“model**” directory

  This directory contains a file “**modelcap**” that is used to identify user-defined models, a “**makefile**” used to compile user-defined code into a dynamic linked library, a sample flight control system file and a file with the aircraft model source code. The sample source code enables the user to modify the structure of the existing aircraft models or add new flight control system or aircraft files. This modified source code is then compiled into a “**dll**” using the “**makefile**” and added to the simulation using the “**modelcap**” file.

- “**ports**” directory

  Similar to the craft and model directories, the ports directory contains files that define the airport and other structural images used in the simulation. In addition, this directory contains the “**portscap**” file that is used to identify the airport image definition files, and the airport image definition files themselves.

- “**terrain**” directory

  This directory contains data files that AVDS uses in rendering the terrain, with the latitude and longitude of the terrain blocks derived from the name of the respective files.

- “**userfiles**” directory

  The “**userfiles**” directory contains files that can be used to configure the simulation or playback on the AVDS window. Among other files, this directory contains the files, “**fighter.ply**” and “**file.ply**”**,** which are sample configuration files used for aircraft initialization and playback configuration window, “**fighter.av**”, which is a parameter lookup table that can be updated between simulation runs. This directory also contains the “**avds.init**” file, a global setup file that saves all the default settings for AVDS.

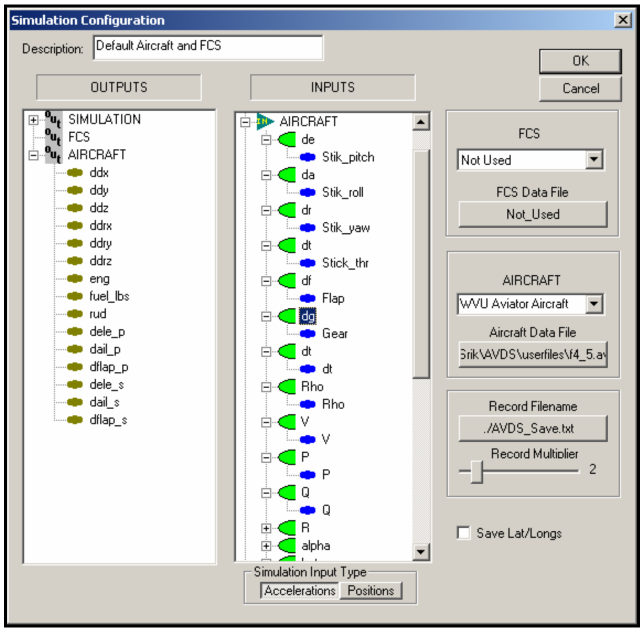
- "**utilities**" directory

  The utilities directory contains files and tools that used to convert AutoCAD files to AVDS format, for incorporating new aircraft images into AVDS, sample code and libraries to develop networked AVDS simulations.

3.3 Interactive Simulation

The interactive simulation mode gives the user the tools to simulate the aircraft dynamics in near real time while representing the vehicle dynamics graphically and interactively passing commands to the simulation. The interactive simulation model consists of two parts, namely the model structure and model parameters. The model structure can either be pre-defined or user defined and is built up with subroutines that define how the model parameters produce the vehicle dynamics. The model parameters are specified in a parameter file and read into AVDS when the interactive simulation begins, allowing the user an extra degree of freedom to change the model parameters at any time between simulations, without the need to recompile the program. A text file specifies the model parameters in the form of a lookup table and are scheduled based on the aircraft velocity, in Mach number, but can be based on other parameters as well. The lookup table can be extended to be multi-dimensional; up to three dimensions, but is restricted to be one dimensional within this research effort.

The interactive simulation parameters can be set before the start of the simulation, by using the simulation configuration window, shown in Figure 3.2, from the Configure menu of AVDS. A representation of the simulation configuration window is shown below.



Description: Default Aircraft and FCS

FCS: Not Used

FCS Data File: Not_Used

AIRCRAFT: WVU Aviator Aircraft

Aircraft Data File: 5rik\AVDS\userfiles\f4_5.av

Record Filename: ./AVDS_Save.txt

Record Multiplier: 2

☐ Save Lat/Longs

Simulation Input Type: Accelerations Positions

Figure 3-2 : Simulation Configuration Window

The simulation configuration window has two smaller windows within it, with the window on the left indicating the "OUTPUTS" from the simulation environment and the window on the right representing the "INPUTS" to the simulation environment. On the right hand side, the 'Inputs' window represents the inputs that are fed into the simulation, the FCS, aircraft model, to a file for recording and the strip charts. These are similar to the parameters in the "OUTPUTS" window. The FCS and the aircraft drop down menus offer the user a range of choices in the type of the flight control system to be integrated with the type of aircraft. These "FCS" and "AIRCRAFT" options can be pre-defined with options that come packaged with AVDS or be user-defined, according to the systems and approach that the user designs. These extra options can be incorporated into the AVDS scheme using user defined simulation code blocks. The following section describes the procedure associated with creating or modifying user defined simulation code blocks and including them in the AVDS scheme.

3.4 User Defined Simulation Code Blocks

Both the aircraft models and flight control systems can be modified in several ways, first, by either changing the aircraft parameter values in the lookup tables or second, by re-writing the simulation code blocks. This section describes in some detail the process of modifying the simulation code blocks.

To redefine the structure of the FCS or the aircraft, the user must add new code or modify the existing User Defined Simulation Code Blocks (UDSCBs), written in C or a language that can interface with C. Once these files are written, they are compiled into object files and are loaded dynamically by the AVDS at runtime for use within the interactive simulations. The AVDS user manual [6] describes the steps for adding a user defined simulation code block as follows:

1. Construct the source code to reflect the desired behavior of the flight control system or the aircraft model, remembering to include the header file **usr_cblock.h**. This header file contains information about the user defined simulation code block structure.
2. Define and set the number of user defined simulation code blocks in the variable 'nBlocks'.
3. Define the user defined simulation code blocks.
4. Add the simulation code blocks to the execution list. Thus, the user defined simulation code block will be reflected as an option to choose from in the simulation configuration window.
5. Prototype the initialization and the update functions. These functions will be executed during the initialization of the model and during the run time, respectively.
6. Declare the user defined simulation code blocks.
7. Add the simulation initialization and update functions using the output[], input[] and param[] arrays for simulation inputs, outputs and lookup parameters respectively.
8. Compile and link user defined simulation code block source code into a dynamic link library (dll).
9. Add the name of the dll to the modelcap file. AVDS is designed to recognize the dynamic link library modules from the modelcap file.
10. Configure the interactive simulation from the simulation configuration window.

Appendix A contains detailed descriptions of the three dimensional lookup tables, and the user defined simulation code block header information. Appendix B shows a sample of the User Defined Simulation Code Block header information.

Chapter 4 - Theoretical Framework

4.1 Mathematical Modeling and Aircraft Equations of Motion

Simulation and visualization of aircraft dynamics on personal computers requires that a set of mathematical equations model the dynamics of the aircraft under review. This section addresses that issue and derives the equations of motion of a rigid aircraft, within the earth's atmosphere. There are a few assumptions that are made during the process of deriving the equations of motion, and are noted accordingly. The derivations are handled in extensive detail in texts on aircraft dynamics [2, 3].

The first step towards deriving the rectilinear and rotational equations of motion of an aircraft is the choice of coordinate systems in which the equations hold true. The first reference frame, namely the earth fixed system, is assumed to be an inertial reference system, where Newton's laws of motion hold true. The assumption is that the earth fixed axis system is an inertial frame which allows us to neglect of the rotational velocity of the earth, a result that would be acceptable for even supersonic flight. The earth fixed axis system is oriented in such a way that the x-axis is directed due north, the y-axis is directed due east and the positive z-axis is directed into the x-y plane, according to the right hand rule. The next reference frame, referred to as the aircraft carried vertical axis system, is obtained by translating the origin of the earth fixed reference frame to coincide with the center of gravity of the aircraft. The positive directions of each axis in this system are parallel to the directions of the respective axes within the earth fixed reference frame. The third reference frame, called the body fixed axis, is attached to the airplane itself, with the origin coinciding with the center of gravity of the airplane. The positive direction of the body fixed x-axis is then oriented to pass through the nose of the aircraft, the positive y-axis through the right wing of the aircraft and the positive z-axis is directed into the x-y plane, according to the right hand rule. The fourth reference frame is called the wind reference system and is used to describe the aircraft's flight path. The positive x-axis in this reference frame is oriented in the direction of the total velocity vector of the aircraft, with the angle of attack α, and the angle of sideslip β, defined by the orientation of this reference frame with respect to the body fixed axes. Thus, the airplane dynamics can now be described using these reference frames. For the purpose of this research, the F4 aircraft model is selected, primarily due to the benign nature of its dynamics and the fact that the stability and control derivatives that describe the aircraft are readily available [2]. Figures 4.1 and 4.2 show a three view of a F4 aircraft and the relative orientations of the reference frames.

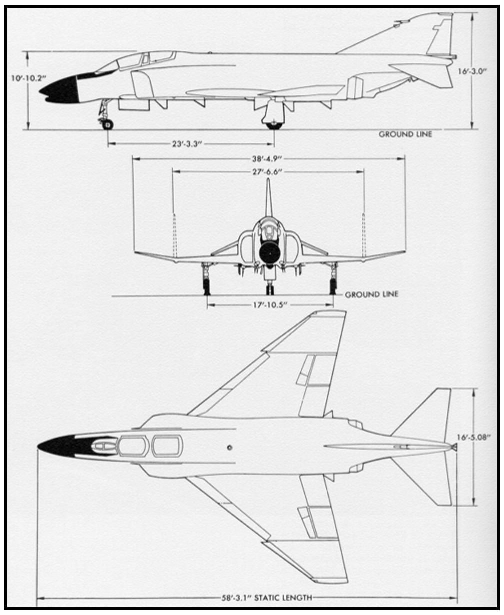


Figure 4-1 : 3-View Drawing of F4 Aircraft

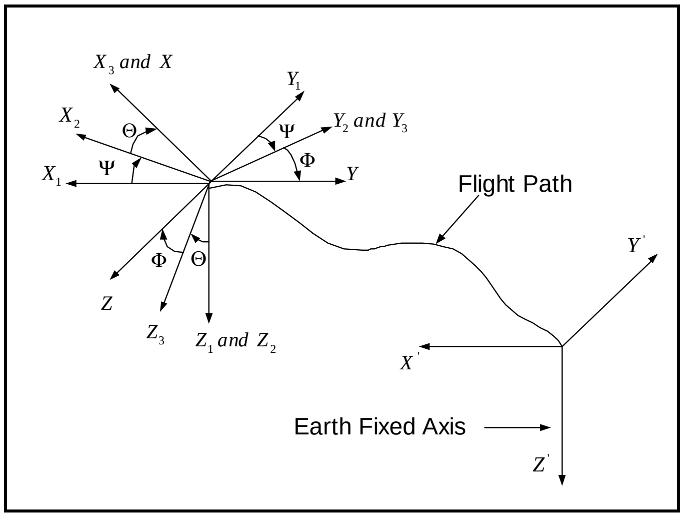


Figure 4-2 : Airplane Orientation with Euler Angles

The equations of motion for an aircraft are developed from the Newton's second law of motion, which states that "***the time derivatives of linear and angular momentum are equal to***

the externally applied forces and moments respectively [2]”. This results in a vector-integral equation of motion, given as:

$$\frac{d}{dt}\int_v \rho_A \frac{d\vec{r'}}{dt}dv = \int_v \rho_A \vec{g}\,dv + \int_S \vec{F}ds \qquad \text{Eq. 4.1}$$

for linear momentum and

$$\frac{d}{dt}\int_v \vec{r'}\times\rho_A \frac{d\vec{r'}}{dt}dv = \int_v \vec{r'}\times\rho_A \vec{g}\,dv + \int_S \vec{r'}\times\vec{F}\,ds \qquad \text{Eq. 4.2}$$

for angular momentum.

The integrals $\int_V$ and $\int_S$ represent the total volume and surface area of the airplane. The above equations are for rigid aircrafts whose external geometry is known. Also, the total mass and mass distribution of the airplane are assumed to be constant with time and the forces $\rho_A \vec{g}\,dv$ and $\vec{F}\,ds$ assumed to be the only external forces acting on the airplane.

The vector-integral equations Eq. 4.1 and Eq. 4.2 can be further reduced to the following form with respect to the earth frame:

$$m(\dot{\vec{V}}_P + \vec{\omega}\times\vec{V}_P) = m\vec{g} + \vec{F}_A + \vec{F}_T \qquad \text{Eq. 4.3}$$

and $\int_V \vec{r}\times\left\{\dot{\vec{\omega}}\times\vec{r} + \vec{\omega}\times\left(\vec{\omega}\times\vec{r}\right)\right\}\rho_A dv = \vec{M}_A + \vec{M}_T \qquad \text{Eq. 4.4}$

These equations are the vector forms of the aircraft equations of motion with respect to the earth reference frame. However, this form is not very friendly to express the steady state and time history response of the system and so, has to be written in the scalar component forms. Thus, the scalar component representation of the linear and angular momentum equations with respect to a the body reference frame is given as

$$m(\dot{U} - VR + WQ) = mg_x + F_{A_x} + F_{T_x} \qquad \text{Eq. 4.5}$$

$$m(\dot{V} + UR - WP) = mg_y + F_{A_y} + F_{T_y} \qquad \text{Eq. 4.6}$$

$$m(\dot{W} - UQ + VP) = mg_z + F_{A_z} + F_{T_z} \qquad \text{Eq. 4.7}$$

and

$$I_{xx}\dot{P} - I_{xz}\dot{R} - I_{xz}PQ + (I_{zz} - I_{yy})RQ = L_A + L_T \qquad \text{Eq. 4.8}$$

$$I_{yy}\dot{Q} + (I_{xx} - I_{zz})PR + I_{xz}(P^2 - Q^2) = M_A + M_T \qquad \text{Eq. 4.9}$$

$$I_{zz}\dot{R} - I_{xz}\dot{P} + (I_{yy} - I_{xx})PQ + I_{xz}QR = N_A + N_T \qquad \text{Eq. 4.10}$$

The above equations are not sufficient to solve for the time histories of motion, primarily because of the fact that the gravity components of Eq. 4.5, Eq. 4.6 and Eq. 4.7 depend on the orientation of the airplane with respect to the earth fixed coordinate system.

The flight path of an airplane relative to the earth fixed axis system can be determined from the knowledge of the velocity components *U*, *V* and *W* in the body fixed axis system and the Euler angles. Thus, the relationship between the earth fixed axes velocity components and the body fixed axes velocity components of the airplane is given as:

$$\begin{Bmatrix} U_1 \\ V_1 \\ W_1 \end{Bmatrix} = \begin{Bmatrix} \dot{x}' \\ \dot{y}' \\ \dot{z}' \end{Bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{Bmatrix} U \\ V \\ W \end{Bmatrix} \qquad \text{Eq. 4.11}$$

The flight path of the airplane in terms of $x'(t)$, $y'(t)$ and $z'(t)$ is found by integrating the terms in equation 4.11. The Euler angles in Eq.4.11 are functions of time and are determined by integrating the following equations:

$$\dot{\phi} = P + Q\sin\phi\tan\theta + R\cos\phi\tan\theta \qquad \text{Eq. 4.12}$$

$$\dot{\theta} = Q\cos\phi - R\sin\phi \qquad \text{Eq. 4.13}$$

$$\dot{\psi} = (Q\sin\phi + R\cos\phi)\sec\theta \qquad \text{Eq. 4.14}$$

Also, the components of the gravitational force are given as:

$g_x = -g\sin\theta$ Eq. 4.15

$g_y = g\sin\phi\cos\theta$ Eq. 4.16

$g_z = g\cos\phi\cos\theta$ Eq. 4.17

Using the set of equations 4.5-4.17, the time history of the aircraft dynamics can be obtained by numerical integration.

The following section discusses the theoretical basis of Neural Networks and the learning algorithms used to train the networks.

4.2 Neural Network Theory

A Neural Network can be defined as "***A massively parallel distributed processor that has a natural propensity for storing experimental knowledge and making it available for use***" [4, 42]. They are also commonly referred to in literature as *neurocomputers*, *connectionist networks*, *parallel-distributed processors* etc. Neural Networks derive their flexibility and computing power through their ability to learn and generalize, producing reasonable outputs for inputs not encountered during the learning phase, and the presence of a massively parallel, distributed structure of individual neurons. A schematic representation of the non-linear model of a neuron is shown below in figure 4.3.

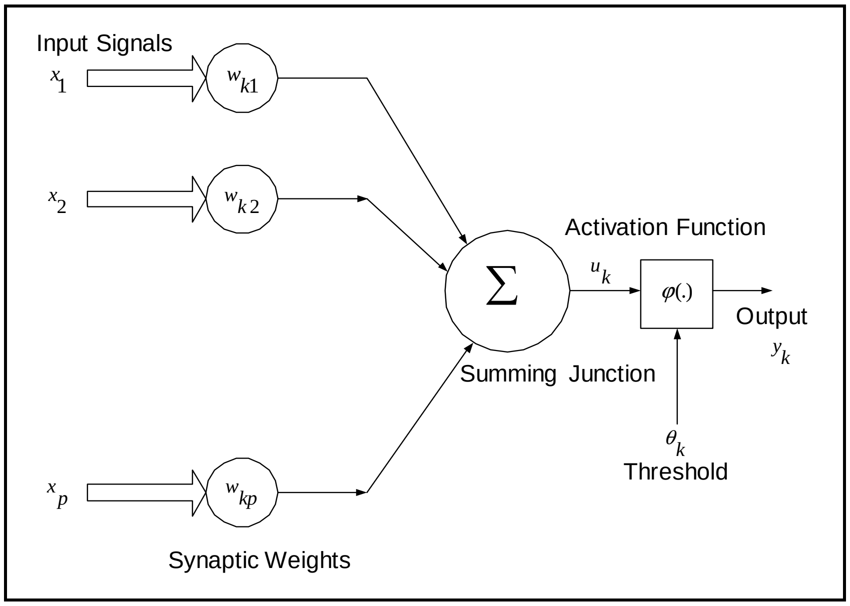


Figure 4-3 : Non-Linear Model of a Neuron

The fundamental elements of a neuron include the *synapses*, the *summing junction* and the *activation function*. The nature and functionalities of each element are described in the following section:

- *Synapses* are the connecting links between two individual neurons *i* and *j* in adjacent layers in the network architecture, and are characterized by an interconnection weight w_{ij}. The input signal at the neuron *j* is the product of the output of the neuron *i* and the interconnection weight w_{ij}.
- The *summing junction* is essentially a linear combiner, and sums all the input signals to the neuron, weighted by the respective synaptic weights.
- The *activation function* is used to limit the amplitude of the output from the neuron and is also sometimes referred to as the *squashing function*. Typically, the *activation function* normalizes the output of the neuron to a closed interval [0,1] or sometimes [-1,1].
- The model of the neuron also includes an externally applied threshold constant, θ_k, that has the effect of lowering the net input of the activation function.

Mathematically, a neuron can be described using the relationship

$$u_k = \sum_{j=1}^{p} w_{kj} x_j \qquad \text{Eq. 4.18}$$

and $y_k = \varphi(u_k - \theta_k)$ Eq. 4.19

where $x_1, x_2 .... x_p$ are the inputs to each neuron in the input layer and $w_{k1}, w_{k2} .... w_{kp}$ are the synaptic weights associated with each input. The value u_k is the output of the linear combiner or the summer, θ_k is the threshold value, $\varphi(.)$ is called the activating function and y_k is the final output of the individual neuron. The activation function defines the output of the neuron and can be classified into three basic types as *Threshold*, *Piecewise-Linear* and *Sigmoid* functions.

- A *Threshold* function is mathematically defined as

$$\varphi(v) = \begin{cases} 1 & if\ v \geq 0 \\ 0 & if\ v \leq 0 \end{cases} \qquad \text{Eq. 4.20}$$

- The mathematical representation of the *Piecewise-Linear* function is given as

$$\varphi(v) = \begin{cases} 1 & if\ v > 0 \\ 0 & if\ v = 0 \\ -1 & if\ v < 0 \end{cases} \qquad \text{Eq. 4.21}$$

- The *Sigmoid* function is the most common form of activation function used in neural networks and is defined as a strictly increasing function that is smooth and asymptotic. Mathematically, a typical sigmoid function can be expressed as

  $$\varphi(v) = \frac{1}{1+e^{-av}} \qquad \text{Eq. 4.22}$$

  where the parameter *a* is called the slope of the sigmoid function. By varying the slope parameter, we can obtain various sigmoid functions.

The following section discusses some of the more common neural network architectures and their properties.

4.3 Neural Network Architectures

The architecture of a neural network is the order of the interconnections between the individual neurons that make up the overall network. In general, there are four different classes of network architectures, namely *Single Layer Feedforward* networks, *Multilayer Feedforward* networks, *Recurrent* networks and *Lattice Structures*.

- *Single Layer Feedforward* networks are the simplest form of a layered neural network, and have their component neurons organized in a single layer. This is strictly a feedforward network, where the source nodes of the 'Input' layer project onto the nodes in the 'Output' layer, but not vice versa. An example of a single layer feedforward network would be the linear associative memory, in which an output pattern is associated with an input pattern by adapting the interconnection weights between the neurons.
- *Multilayer Feedforward* network architectures have more than one layer of interconnected neurons, referred to as the *Hidden Layer*, between the 'Input' layer and 'Output' layer. The computation nodes associated with the 'Hidden' layer are called as *hidden neurons* or *hidden layer neurons* and the presence of these neurons enables a network to extract higher order statistics and approximate non-linear functions effectively. The inputs to the activation function of each neuron

in the ‘Hidden’ layer are then the weighted sum of the outputs from the neurons in the ‘Input’ layer. These inputs are then passed through the activation function giving the corresponding output at each node in the Hidden layer, which then serve as inputs to the neurons of the next layer, until the ‘Output’ layer is reached where the output from each neuron gives the overall response of the network to the input pattern.

- *Recurrent* networks are distinct from a feedforward network in the aspect that they have at least one *feedback* loop in the network architecture. In this architecture, the output of a neuron is fed back as an input to other neurons in the same layer, resulting in a *feedback* network. The output of a neuron is fed to itself and all the other neurons in the same layer, results in a *self-feedback* network architecture.
- A *lattice structure* network architecture is characterized by the presence of one, two or ‘N’ higher dimensional array of neurons with a corresponding set of input layer neurons. The *lattice structure* can be considered to be a feedforward network with the neurons in the output layer arranged in rows and columns, where the dimension of the lattice characterizes the number of dimensions in space in which the graph lies.

For the purpose of this research, a *multilayer feedforward* network was with one hidden layer chosen as the structure of the neural network. The next section discusses the *multilayer feedforward* network in more detail, with respect to its architecture and the learning algorithm used to train the networks.

4.4 Extended Backpropagation Algorithm

The *multilayer feedforward* network architecture, also referred to as *Multilayer Perceptrons* (MLPs), is one of the more important network architectures. Typically it consists of an *input layer* of sensory units, one or more *hidden layers* of computation nodes and one *output layer* of computational neurons. A popular approach to training the MLPs in a supervised manner is the *error-backpropagation* algorithm, based on the *error-correction* learning rule. The purpose of this rule is to minimize a cost function based on the error $e_k(t) = d_k(t) - y_k(t)$, where $d_k(t)$ is the desired output from the network and $y_k(t)$ is the actual response of the network, to the input presented, such that the actual response of each neuron in the output layer approaches the target response in some statistical sense. Figure 4.4 shows a schematic representation of a *multilayer feedforward* neural network.

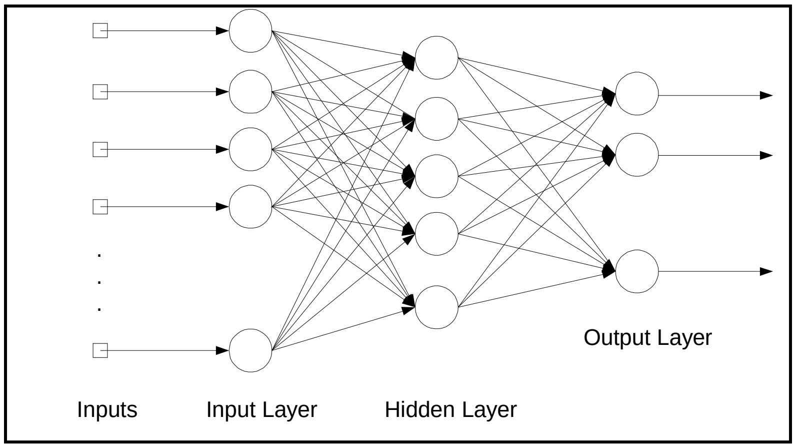


Figure 4-4 : Multilayer Perceptron with One Hidden Layer

The *error-backpropagation* training process consists of two distinct phases, namely the *forward* phase and the *backward* phase. In the *forward* phase, an input pattern is applied to the nodes in the input layer, which then propagates through the each of the hidden layers, till it reaches the output layer, where the output at the computational nodes is the overall response of the network to the input pattern that was presented. Once the overall response of the network is obtained, it is then compared to the target output and the difference between the two produces an error term. This error is then propagated backwards, leading to the term *backpropagation*, through the network structure and the corresponding interconnection weights are adjusted to make the response of the network move closer to the desired response. A detailed theoretical explanation of the derivation of the back propagation algorithm is provided in the text "Neural Networks: A Comprehensive Foundation" [4].

Although the back propagation algorithm is popular for training multilayer perceptrons, it has a few drawbacks that make the general algorithm unsuited for the current research. This section describes a few of the drawbacks of the backpropagation algorithm, particularly with reference to its activation function, the sigmoid function $\frac{1}{1+e^{-sum_j}}$, and also suggests solutions to overcome these drawbacks:

- The *backpropagation* algorithm, applied to a multilayer perceptron, is not guaranteed to converge to the global minimum, even when the learning rate is properly selected, as the performance index may have many local minima;
- Although the backpropagation algorithm can approximate any non-linear function, it requires extensive training, which makes it unsuitable for on-line applications.

A few approaches, among others, to overcome the above drawbacks of applying the backpropagation algorithm to train a multilayer perceptron are:

- Backpropagation algorithm with a momentum term used in updating the interconnection weights;
- Backpropagation with a variable learning rate;
- Backpropagation with batch training, network pruning and complexity penalty.

A different solution, the *Extended Back Propagation Algorithm* (EBPA), to overcome these problems was developed at WVU [43], based on a significantly modified sigmoid activation function $\frac{U-L}{1+e^{\frac{-net}{T}}}+L$. This uses a heterogeneous network, wherein each neuron in the hidden and the output layer has its own set of free parameters, namely the upper and lower bounds of the output range, as well as the slope of the sigmoid activation function. With respect to the multilayer perceptron architecture shown in Figure 4.5, the following derivations for the training the multilayer perceptron, using the EBPA, hold good. The modified schemes for the outputs at each neuron in the hidden and the output layers is given as

$$sum_i = \sum_{h=1}^{n} A_h V_{hi} + \Theta_i \qquad \text{Eq. 4.23}$$

$$B_i = f(sum_i, U_i, L_i, T_i) = \frac{U_i - L_i}{1+e^{-\frac{sum_i}{T_i}}} + L_i \;;\quad i = 1, 2, \ldots m \qquad \text{Eq. 4.24}$$

for the hidden layer neurons, and

$$sum_j = \sum_{i=1}^{m} B_i W_{ij} + \Gamma_j \qquad \text{Eq. 4.25}$$

$$C_j = f(sum_j, U_j, L_j, T_j) = \frac{U_j - L_j}{1 + e^{-\frac{sum_j}{T_j}}} + L_j \; ; \quad j = 1, 2, \ldots p \qquad \text{Eq. 4.26}$$

for the output layer neurons. This would constitute the forward phase of training, where the overall response of the network is given by C_j.

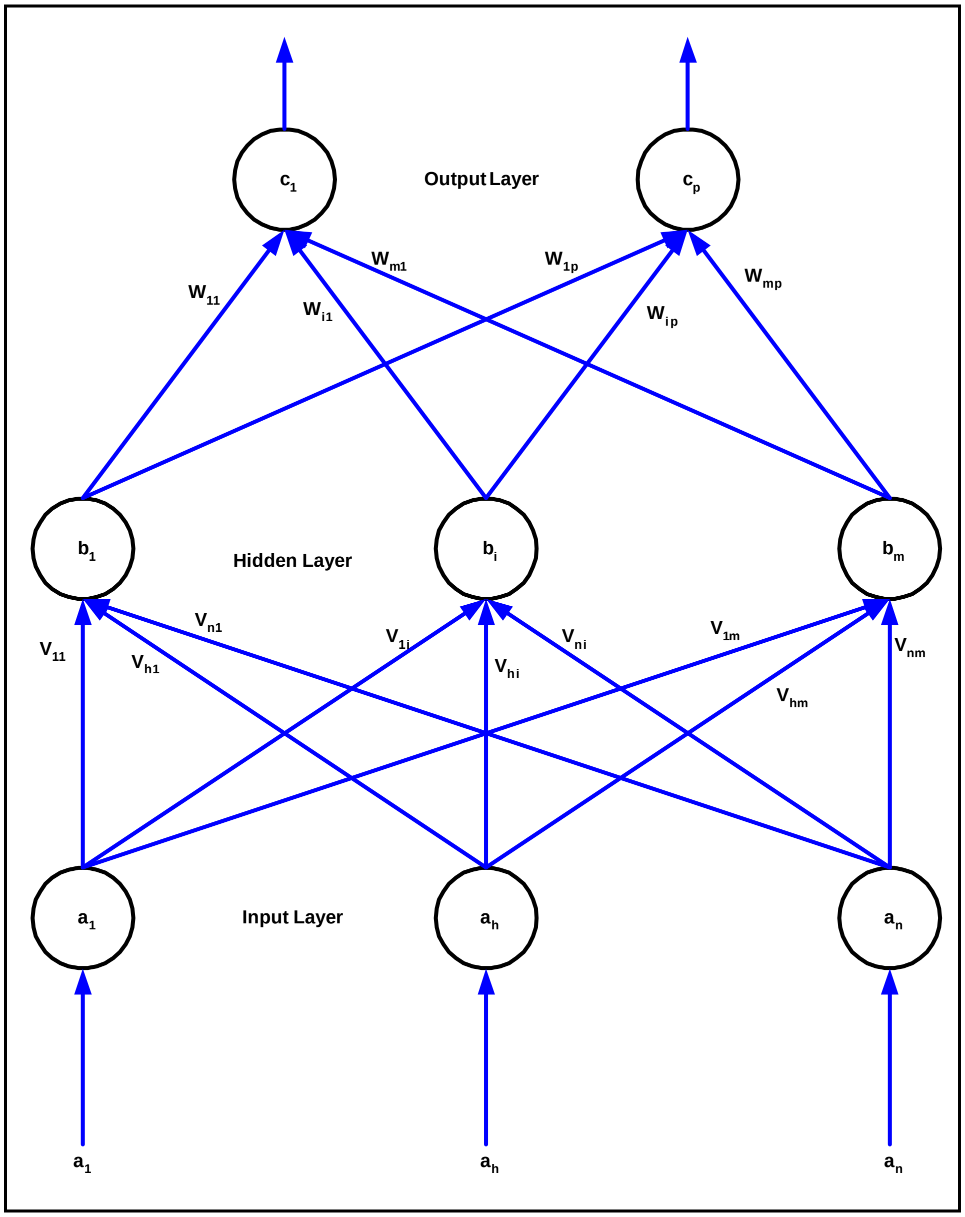


Figure 4-5 : Feedforward Multilayer Perceptron

In the *backpropagation* or the updating phase, the overall outputs of the network are compared to the target outputs the network is designed to track, and the error $e_k(t) = d_k(t) - y_k(t), \forall k$ *neurons in the output layer* is propagated backwards, to update the values of the interconnecting weights and thresholds at each neuron in both the hidden and output layers, as a means to minimize the difference between the overall output of the network and the actual parameter being tracked.

The first step in the back propagation step is to determine the partial derivatives of the network with respect to each of the parameters in the network. For the output layer, taking the partial derivatives with respect to the upper bound, lower bound and the gradient results in the following expressions:

$$f_{sum_j}{}' = \frac{\partial f(sum_j, U_j, L_j, T_j)}{\partial sum_j} = -\frac{(C_j - U_j)(C_j - L_j)}{T_j(U_j - L_j)} \qquad \text{Eq. 4.27}$$

$$f_{U_j}{}' = \frac{\partial f(sum_j, U_j, L_j, T_j)}{\partial U_j} = \frac{1}{1 + e^{-\frac{sum_j}{T_j}}} \qquad \text{Eq. 4.28}$$

$$f_{L_j}{}' = \frac{\partial f(sum_j, U_j, L_j, T_j)}{\partial L_j} = 1 - \frac{1}{1 + e^{-\frac{sum_j}{T_j}}} = 1 - f_{U_j}{}' \qquad \text{Eq. 4.29}$$

$$f_{T_j}{}' = \frac{\partial f(sum_j, U_j, L_j, T_j)}{\partial T_j} = \frac{sum_j(C_j - U_j)(C_j - L_j)}{T^2{}_j(U_j - L_j)} = -\frac{sum_j f_{sum_j}{}'}{T_j} \qquad \text{Eq. 4.30}$$

Similarly, at the hidden layer:

$$f_{sum_i}{}' = \frac{\partial f(sum_i, U_i, L_i, T_i)}{\partial sum_i} = -\frac{(B_i - U_i)(B_i - L_i)}{T_i(U_i - L_i)} \qquad \text{Eq. 4.31}$$

$$f_{U_i}{}' = \frac{\partial f(sum_i, U_i, L_i, T_i)}{\partial sum_i} = \frac{1}{1 + e^{-\frac{sum_i}{T_i}}} \qquad \text{Eq. 4.32}$$

$$f_{L_i}{}' = \frac{\partial f(sum_i, U_i, L_i, T_i)}{\partial sum_i} = 1 - \frac{1}{1 + e^{-\frac{sum_i}{T_i}}} = 1 - f_{U_i}{}' \qquad \text{Eq. 4.33}$$

$$f_{T_i}{}' = \frac{\partial f(sum_i, U_i, L_i, T_i)}{\partial sum_i} = \frac{sum_i(B_i - U_i)(B_i - L_i)}{T^2{}_i(U_i - L_i)} = -\frac{sum_i f_{sum_i}{}'}{T_i} \qquad \text{Eq. 4.34}$$

Once the partial derivatives are computed, the next step is the formulation of the error term to be minimized. At the output layer, the difference between the overall network output and the target network output constitutes the error term and are given as:

$$\delta_{sum_j} = f'_{sum_j}(Y_j - C_j) \qquad \text{Eq. 4.35}$$

$$\delta_{U_j} = f'_{U_j}(Y_j - C_j) \qquad \text{Eq. 4.36}$$

$$\delta_{L_j} = f'_{L_j}(Y_j - C_j) \qquad \text{Eq. 4.37}$$

$$\delta_{T_j} = f'_{T_j}(Y_j - C_j) \qquad \text{Eq. 4.38}$$

These errors are then propagated back to the neurons in the hidden layer, where the formulation of the error term at each neuron is not straight forward as they are at the output layer, as there are no well-defined target values at any neuron. Thus, the error terms at the hidden layer are computed using the chain rule, by multiplying the delta terms from the output layer with the interconnecting weights between the neurons in the hidden and the output layers:

$$\delta_{sum_i} = f'_{sum_i}\sum_{j=1}^{o}\delta_{sum_j} w_{ij} \qquad \text{Eq. 4.39}$$

$$\delta_{U_i} = f'_{U_i}\sum_{j=1}^{o}\delta_{U_j} w_{ij} \qquad \text{Eq. 4.40}$$

$$\delta_{L_i} = f'_{L_i}\sum_{j=1}^{o}\delta_{L_j} w_{ij} \qquad \text{Eq. 4.41}$$

$$\delta_{T_i} = f'_{T_i}\sum_{j=1}^{o}\delta_{T_j} w_{ij} \qquad \text{Eq. 4.42}$$

Once the errors and delta terms are determined at both the hidden and the output layers, the bias terms and interconnecting weights are then updated using the following relationships:

$$\Delta\Gamma_j(k) = \eta_{sum}\delta_{sum_j} + \alpha_{sum}\Delta\Gamma_j(k-1)\,; \quad j = 1, 2, \ldots o \qquad \text{Eq. 4.43}$$

$$\Delta\Theta_i(k) = \eta_{sum}\delta_{sum_i} + \alpha_{sum}\Delta\Theta_i(k-1); \; i = 1, 2, \ldots m \qquad \text{Eq. 4.44}$$

and the weights are updated as

$$\Delta w_{ij}(k) = \eta_{sum}\delta_{sum_j} B_i + \alpha_{sum}\Delta w_{ij}(k-1) \qquad \text{Eq. 4.45}$$

$$\Delta v_{hi}(k) = \eta_{sum}\delta_{sum_i} A_h + \alpha_{sum}\Delta v_{hi}(k-1) \qquad \text{Eq. 4.46}$$

Now the upper and lower bounds as well as the temperatures on all the neurons in both the hidden layer and the output layer are updated using:

$$\Delta U_j(k) = \eta_u \delta_{U_j} + \alpha_u \Delta U_j(k-1) \quad \text{Eq. 4.47}$$

$$\Delta L_j(k) = \eta_L \delta_{L_j} + \alpha_L \Delta L_j(k-1) \quad \text{Eq. 4.48}$$

$$\Delta T_j(k) = \eta_T \delta_{T_j} + \alpha_T \Delta T_j(k-1) \quad \text{Eq. 4.49}$$

$$\Delta U_i(k) = \eta_i \delta_{U_i} + \alpha_U \Delta U_i(k-1) \quad \text{Eq. 4.50}$$

$$\Delta L_i(k) = \eta_L \delta_{L_i} + \alpha_L \Delta L_i(k-1) \quad \text{Eq. 4.51}$$

$$\Delta T_i(k) = \eta_T \delta_{T_i} + \alpha_T \Delta T_i(k-1) \quad \text{Eq. 4.52}$$

In the above equations, the terms η and α represent the learning rate and the momentum rates. The learning rate is a factor that determines the amount by which each parameter is updated at each time step. The momentum term controls the extent of the contribution of the weights in the previous time step on the current weights. Additional degrees of freedom can be achieved by having different learning and momentum rates for each of the parameters. This learning algorithm has a few advantages over the traditional backpropagation algorithm, such as a more degrees of freedom with regards to learning due to the presence of the parameters *U*, *L*, and *T* in the activation function. Also, this learning algorithm is more likely to avoid the local minima problem that is traditionally associated with the backpropagation algorithm and potentially learn faster and more accurately than the traditional backpropagation algorithm. There are some downsides to this learning method in that this takes approximately 30%-50% more computations, due to the presence of the extra degrees of freedom in *U*, *L* and *T*, although this issue is of concern only when the method is applied to on-line learning.

4.5 Flynn's Taxonomy and Message Passing Interface

Enough! That is an expression hardly used by researchers in the same sentence as computational power. Ever since the dawn of the computer era, researchers and scientists have been clamoring for increased computational capacity, something that has not been satisfied even with the recent advances in semiconductor technology. Therefore, to meet the requirements for higher computational power, with the existing microprocessor technology, a new computing paradigm, *parallel computing*, was developed, wherein more than one distinct processor would work together, forming a *parallel computer*, to solve a given task. The parallel computer architectures were originally classified according to a popular scheme known as *Flynn's taxonomy*, developed by Flynn in 1966 [44]. This classified the parallel computers into four distinct classes, based on the number of instruction and data streams, as *single instruction single data* (SISD), *multiple instruction multiple data* (MIMD), *single instruction multiple data* (SIMD) and *multiple instruction single data* (MISD) architectures.

The classical von Neumann computer is an example of the SISD architecture and has just a single instruction stream and a single data stream.

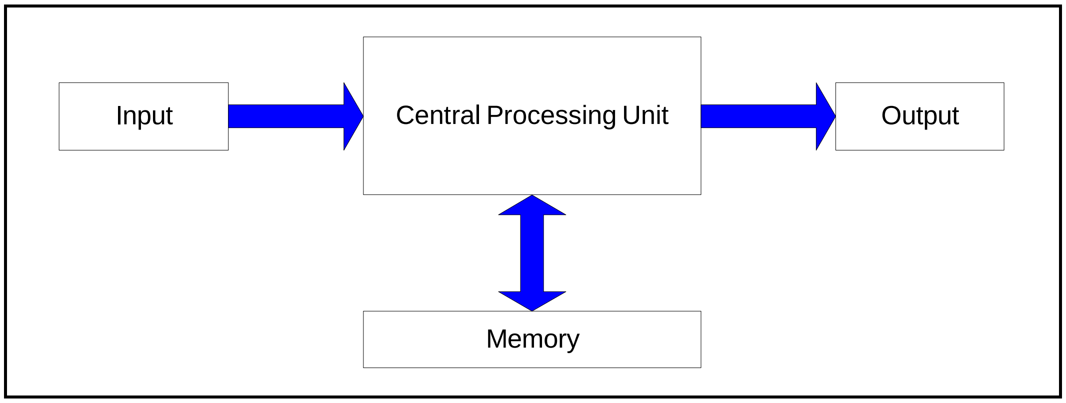


Figure 4-6 : Classic von Neumann Computer Architecture

The SIMD computer architecture has a single central processing unit exclusively dedicated to control and a bunch of slave arithmetic and logic units, each with its own memory. During each instruction cycle, the central processing unit sends an instruction to all the subordinate processors, which then execute it or remain idle. The general MIMD architecture is different from the SIMD architecture in the aspect that each processor in a MIMD system is a fully autonomous central processing unit, with its own control unit as well as arithmetic and logic unit. Thus, the MIMD systems are asynchronous in nature and are capable of executing its own set of instructions at its own pace.

The MIMD architecture is further divided into two distinct groups as *shared memory* and *distributed memory* architectures. The shared memory architecture consists of a collection of distinct processors and distinct memory modules, interconnected by a communication network. Figure 4.6 shows a schematic representation of generic shared memory architecture.

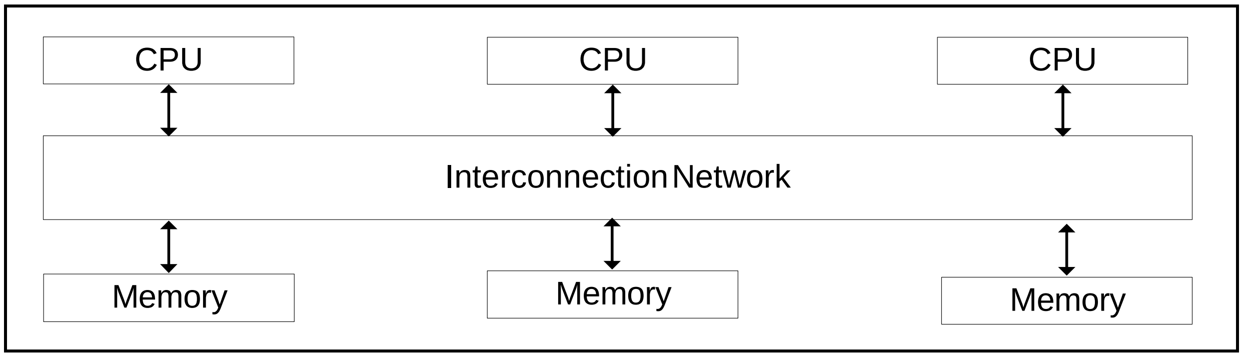


Figure 4-7 : Generic Shared Memory Architecture

On the other hand, with distributed memory architecture, each processor has its own private memory, and is connected to other processors by a common communication network. Figure 4.7 shows the schematic representation of generic distributed memory architecture.

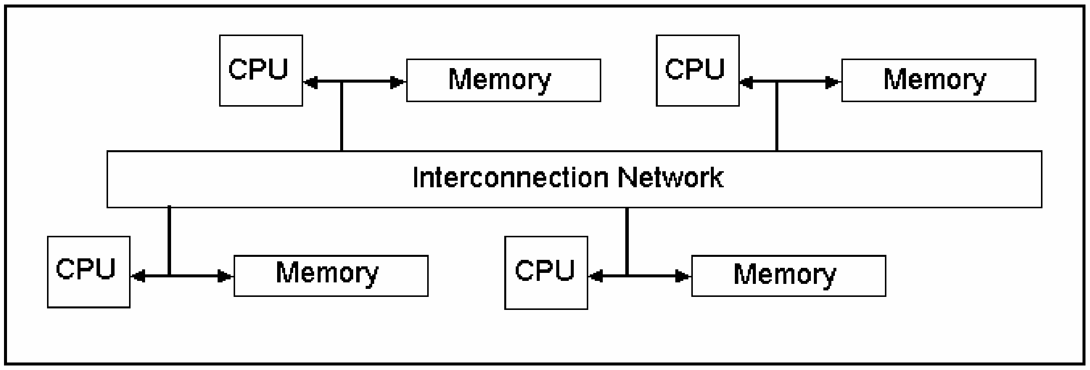


Figure 4-8 : Generic Distributed Memory Architecture

The more popular methods of programming shared memory systems involves one of the various forms of message passing, *Message Passing Interface* or *Parallel Virtual Machine*, where the individual processes coordinate their activities by explicitly sending and receiving messages. The popularity of the *message passing* paradigm can be attributed to a large extent to its portability and its ability to run on heterogeneous systems, from distributed memory systems, shared memory systems, network of workstations, or a combination of the above. The *Message Passing Interface* (MPI) is described as "a standardized portable message passing system, designed by a group of researchers from academia and industry, to function on a wide variety of parallel computers" [45]. MPI standard provides efficient and reliable communication interface between processes, avoiding memory-to-memory copying, and facilitates overlap of computation and communication, with the ability to offload the communication to a secondary processor. The

Parallel Virtual Machine (PVM) can be described as a software package that allows the user to create and access a concurrent computing system created from a network of loosely coupled processing elements including standalone workstations, vector processors or even parallel supercomputers. The distinguishing factor of PVM is that it makes the heterogeneous collection of computers appear as a single virtual machine, leading to its name [46].

In recent years, the publication of the OpenMP standard for shared memory parallelism has provided developers with another portable and scalable API for writing parallel programs. Salient features of OpenMP are that it is made up of a simple set of directives and can provide incremental parallelism and does not have to deal with message passing and allows the use of shared data.

For the purpose of this research, MPI is chosen as the communication paradigm, due mainly to the ease of programming and the availability of extensive support. MPI is portable and does not require any special compilers to work. Within this research effort, SIMD architecture is followed, with source code written as a single chunk of instruction sequence with each node on the cluster executing a section of the code, based on its rank. The individual processes execute in its own address space communicate with each other via calls to MPI primitives. This is a substantial advantage when applied to the adaptive flight control system scheme as different modules of the scheme can be programmed to attain its own goals and execute on different computers. Figure 4.8 shows an example of a parallel program written in C using MPI for communication [47]. In this example, six distinct processes, one of which is designated as the root process, are spawned on six different machines. The non-root processes send a greeting to the root, which then displays it as its output. The result of the program is shown in Figure 4.9.

```
 greetings.c -- greetings program

 Send a message from all processes with rank != 0 to process 0.
    Process 0 prints the messages received.

 Input: none.
 Output: contents of messages received by process 0.

/
include  <stdio.h>
include  <string.h>
include  "mpi.h"

ain(int argc, char* argv[]) {
  int  my_rank;                 /* rank of process          */
  int  p;                       /* number of processes      */
  int  source;                  /* rank of sender           */
  int  dest;                    /* rank of receiver         */
  int  tag = 0;                 /* tag for messages         */
  char message[100];            /* storage for message      */
  MPI_Status  status;         /* return status for  receive  */

  /* Start up MPI */
   MPI_Init(&argc, &argv);

  /* Find out process rank  */
       MPI_Comm_rank(MPI_COMM_WORLD,  &my_rank);

  /* Find out number of processes */
       MPI_Comm_size(MPI_COMM_WORLD,  &p);

  if (my_rank != 0) {
      /* Create message */
      sprintf(message, "Greetings from process %d!", my_rank);
      dest = 0;
           MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
 }
 else
 {
      /* my_rank == 0 */
     for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
            }
  }

  /* Shut down MPI */
   MPI_Finalize();
```

Figure 4-9 : Example of a MPI Based Program

```
----PBSPrologueReport----
Beforerunning, thereare    8IPC semaphoresand IPC sharedmemory segmentson node4
Greetingsfromprocess1!
Greetingsfromprocess2!
Greetingsfromprocess3!
Greetingsfromprocess4!
Greetingsfromprocess5!
node4
node4
node3
node3
node2
node2
PBSEpiloguesReport
PBSJOB(10126.energy):managedby node4
/usr/spool/PBS/aux/10126.energy
```

Figure 4-10 : Result of the MPI “greetings” Program

Chapter 5 - Design of the Experimental Setup

5.1 Matlab GUI

This chapter discusses the experimental setup that was used for the implementation of the NN based fault tolerant flight control system addressed within this research effort.

The experimental setup covers subtasks includes the design and implementation of a Matlab based *Graphical User Interface* (GUI) that allows the user to interactively setup the task at hand, the setup of the graphical simulation/visualization environment to allow for data exchange between itself and the flight control schemes, followed by the setup of the Beowulf cluster based computation platform.

The first sub-task involved the development of the GUI and was developed within the Matlab 5.3 environment for this research study. The initial GUI screen is shown below.

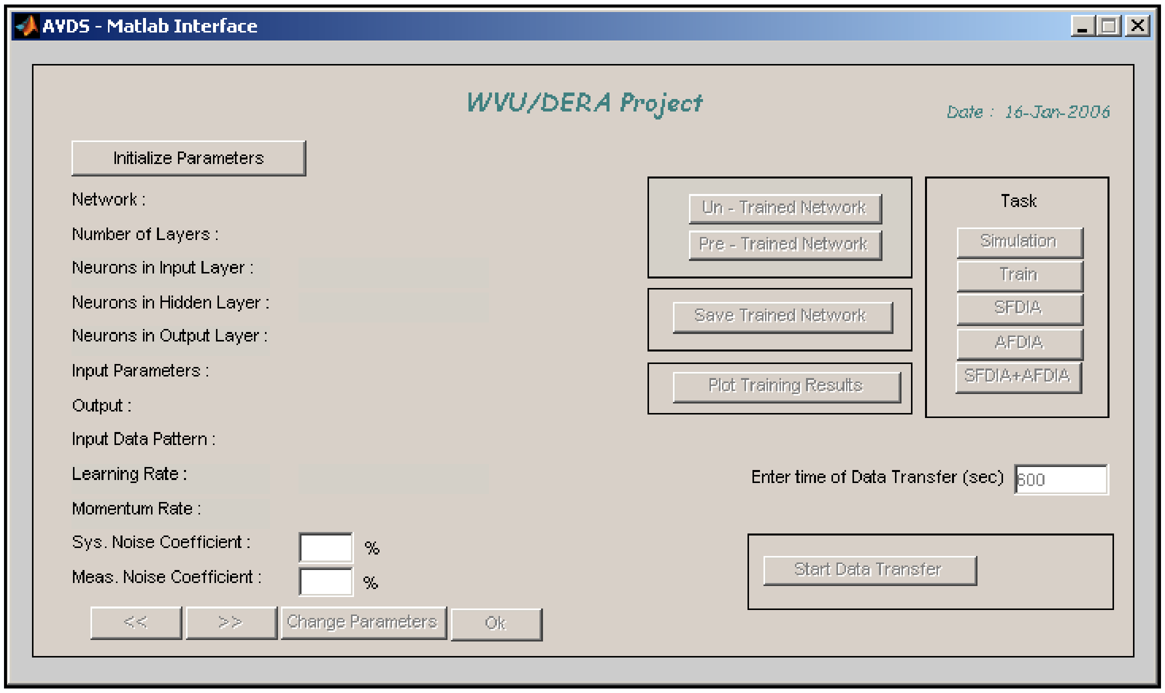
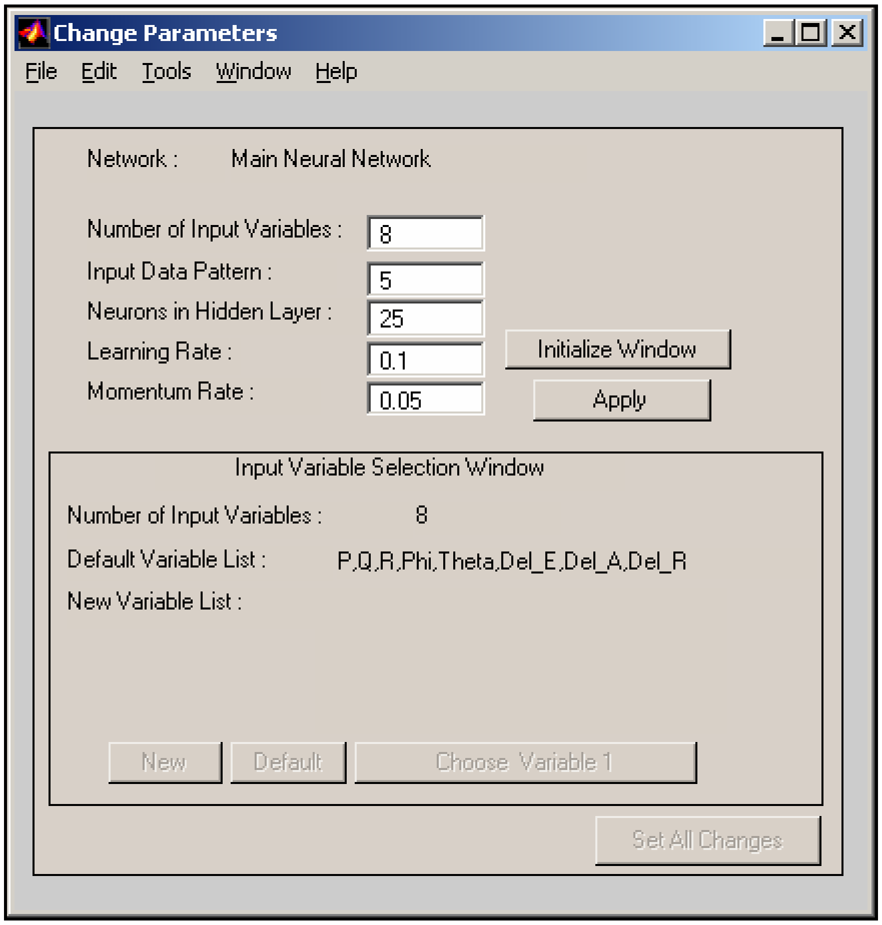


Figure 5-1 : Main User Interface

On this initial screen, when the "*Initialize Parameters*" button is clicked, it calls an initialization script which then initializes the handles for all the graphics objects on this screen, and sets the default parameters for all the seven NNs, including the MNN, P-DNN, Q-DNN, R-

DNN, ROLL-NNC, PITCH-NNC and the YAW-NNC. Once the graphics object handles are initialized, the script then activates the buttons that presents the user an option to select between pre-trained NNs or un-trained NNs. Once the choice of the networks are made, the script then activates the graphics objects on the main GUI and displays the current settings of the NN structures that were chosen. This GUI is setup in such a way that the user can scroll through the different NN structures using the scroll buttons.

The inner workings of the GUI can be explained with the help of an example. Taking the MNN as the test case, in order to alter its structure, the user has to scroll forward or backward to display the current setting of the MNN. Once this is done, clicking on the "*Change Parameters*" button will bring up the second layer of the GUI. The second layer of the GUI is shown below.



Network : Main Neural Network

Number of Input Variables : 8
Input Data Pattern : 5
Neurons in Hidden Layer : 25
Learning Rate : 0.1
Momentum Rate : 0.05

Number of Input Variables : 8
Default Variable List : P,Q,R,Phi,Theta,Del_E,Del_A,Del_R
New Variable List :

Figure 5-2 : Change Parameters Window

Within this GUI, when the "*Initialize Window*" button is clicked, the script determines which network was selected in the previous screen and then brings up the current setting of the network structure and displays the parameters in the appropriate fields. Traditionally, MLP based neural network structures have been hand coded into the program code. This makes it cumbersome to alter the structure of the networks if the need arises. This GUI based approach makes the design of the network structure flexible and the user can choose to vary any of the parameters that define the network. The first parameter in this window is the number of input variables that are used to estimate the outputs from the NN. The default setting for the MNN is 8 variables. The other parameters are the number of Hidden layer neurons, the input data pattern, the learning rate of the network and the momentum rate of the network. The input data patters in the value that determines the number of time steps back in time that the NN reaches to gather its input data, to estimate its outputs at the current time step. After changing any of these parameters, clicking on the "*Apply*" button sets the appropriate values in memory. The user also has an option to choose the input variables to the NN. When the "*New*" button is clicked, a popup window presents the user with a list of variables that are sent across from the simulation environment. The user can choose the inputs to the NN from this popup window. The variables that are chosen from this popup window are identified by their indexes and are stored in an array in the order that they were chosen. The popup window that allows the user to choose the input variables is shown below.

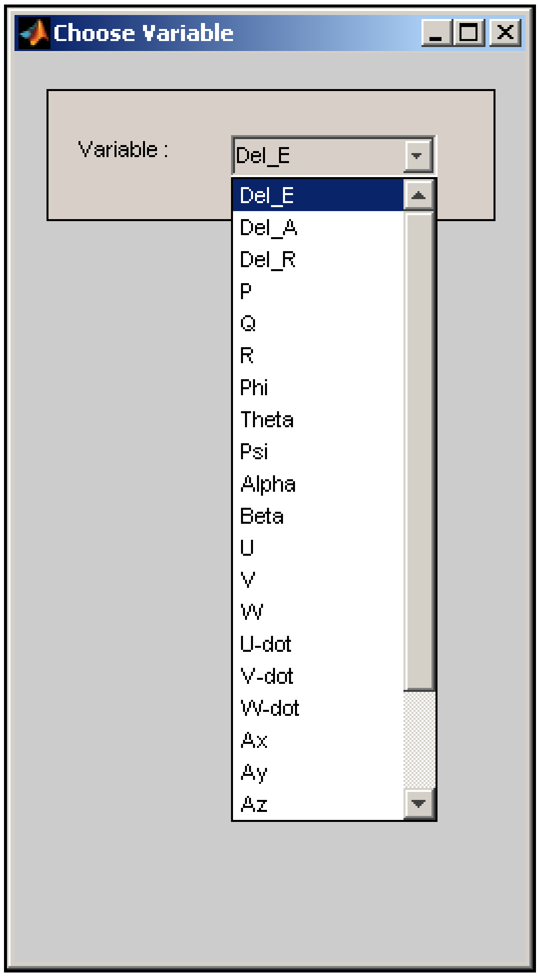
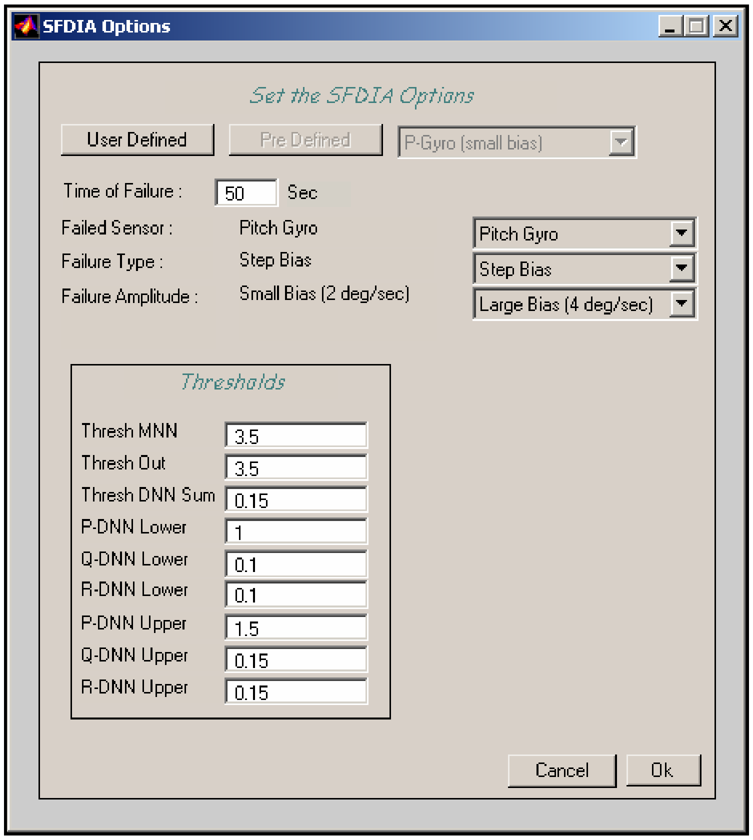


Figure 5-3 : Choose Variables

Once the variables for the NN are chosen and "*Apply Settings*" button is clicked, the user is taken back to the main GUI window. On the main GUI, once all the network parameters are set to the user's satisfaction, clicking on the "*Ok*" button would activate the buttons to select the task to be executed. The tasks that can be selected are "*Simulation*", "*Train*", "*SFDIA*", "*AFDIA*" and "*SFDIA+AFDIA*". "*Simulation*" allows the user to run a simulation of the aircraft dynamics within AVDS with predefined maneuvers that are generated offline. The "*Train*" option allows the user to train the networks for the purpose of the SFDIA or the AFDIA tasks. This training can be done either offline or on-line with data from the AVDS environment. If the "*Offline*" training option is chosen, then the user is allowed to "*fly*" the aircraft within AVDS and the resulting data is then stored on the computer. This data can then be accessed offline and used to train the NN. If the "*On-line*" training option is chosen, the user can "*fly*" the aircraft within AVDS and simultaneously train the NN with on-line data. The difference between the scheme training the NN offline and the scheme training the NNs on-line are in the way the inter-process communication schemes are setup. In the case of on-line training, data frames are sent across from the AVDS environment where the aircraft is flown to the NN structures at a pre defined frequency of 30 Hz. On the other hand, for training the networks offline, the aircraft is flown and the data store on the computer. This data is then used independently of AVDS to train the networks.

The "*SFDIA*" option brings up another GUI, shown below, wherein the user can set the parameters for the SFDIA task. This GUI allows the user to select between pre-defined failure scenarios or custom defined failure scenarios. The parameters of the failure scenarios include the gyro which is set to fail, the type of failure on the selected gyro and the magnitude of the failure. The user can then choose the time of injection of the failure on the rate gyros. The types of failure include step and ramp type failures. This also allows the user to set the thresholds for failure detection and identification sub-tasks.



Time of Failure : 50 Sec

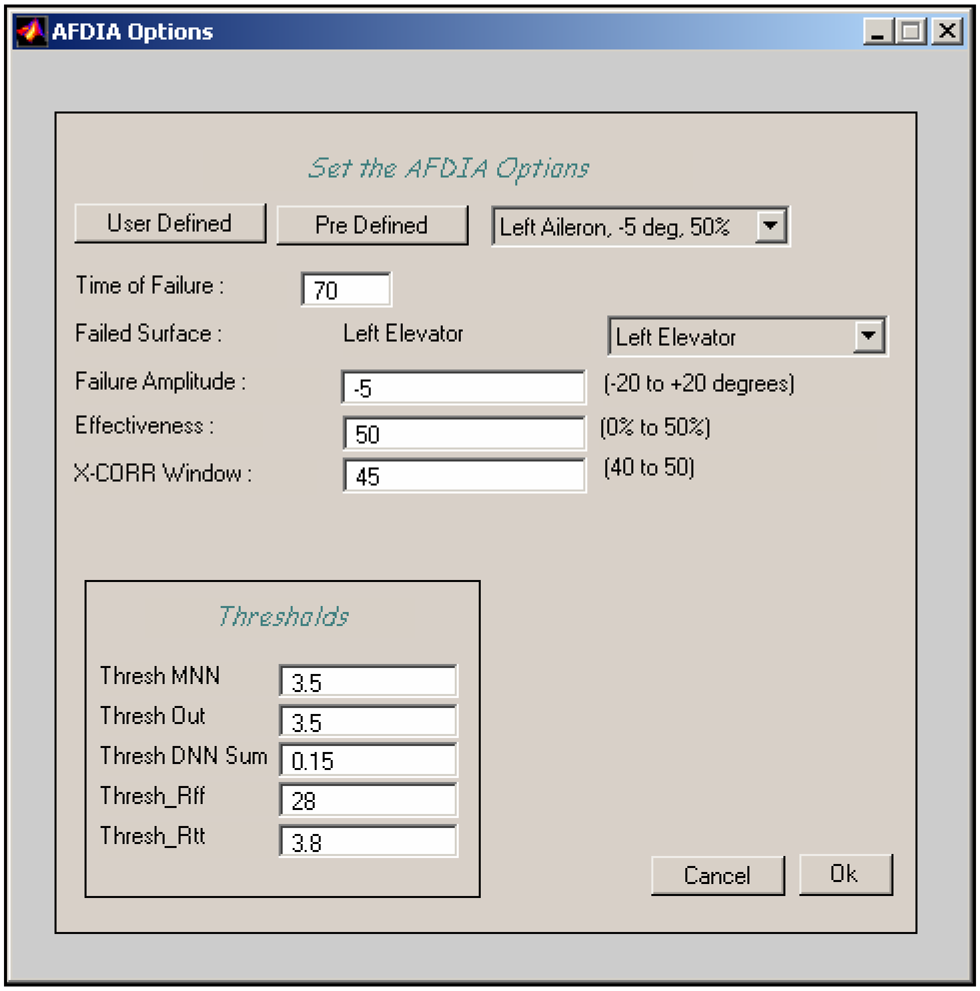
Failed Sensor :	Pitch Gyro	Pitch Gyro
Failure Type :	Step Bias	Step Bias
Failure Amplitude :	Small Bias (2 deg/sec)	Large Bias (4 deg/sec)

Thresh MNN	3.5
Thresh Out	3.5
Thresh DNN Sum	0.15
P-DNN Lower	1
Q-DNN Lower	0.1
R-DNN Lower	0.1
P-DNN Upper	1.5
Q-DNN Upper	0.15
R-DNN Upper	0.15

Figure 5-4 : SFDIA Task Window

Clicking on the "*Ok*" button on this GUI window sets the parameters for the SFDIA task in memory and returns control to the Main GUI.

Similar to the GUI for the SFDIA task, the "*AFDIA*" button on the main GUI brings up another window that allows the user to set the parameters for the AFDIA task. The GUI for this is shown below. The parameters for the failure scenarios in this case include the control surface that is set to fail on the aircraft, the magnitude of the failure and the effectiveness of the failed surface which defines the amount of surface that is assumed to be lost.



AFDIA Options

Set the AFDIA Options

Left Aileron, -5 deg, 50%

Time of Failure : 70
Failed Surface : Left Elevator   Left Elevator
Failure Amplitude : -5   (-20 to +20 degrees)
Effectiveness : 50   (0% to 50%)
X-CORR Window : 45   (40 to 50)

Thresholds

Thresh MNN 3.5
Thresh Out 3.5
Thresh DNN Sum 0.15
Thresh_Rff 28
Thresh_Rtt 3.8

Figure 5-5 : AFDIA Task Window

Within the AFDIA task window, the user has a choice to selecting between predefined failure scenarios or custom defined scenarios. With both options, the user has to set the time of injection of the failure and the thresholds for the detection and identification of the actuator failures. Clicking on the "*Ok*" button sets the parameters for the AFDIA task in memory and returns control to the main GUI.

The next task option that is available on the main GUI is the option for "*SFDIA+AFDIA*" task. Within this task, both sensor failures and actuator failures are introduced on the aircraft. The GUI that allows to user to define the parameters of failure injection, detection and identification for the SFDIA+AFDIA task is shown below. Within this GUI, the user can select the failure options, thresholds, and the time of failure injection in the same manner as they were selected for the individual tasks.

Once the parameters that define the specific task chosen are set, the control returns to the main window, where the total time of simulation can be entered. The user is now ready to run the simulation for the task at hand.

The data exchange scheme is based on the client-server model, set in such a way that the Matlab environment under which this GUI runs is running the client side socket, while the master node on the Beowulf cluster is running the server side socket. The communication scheme is shown in the Figure 5-7.

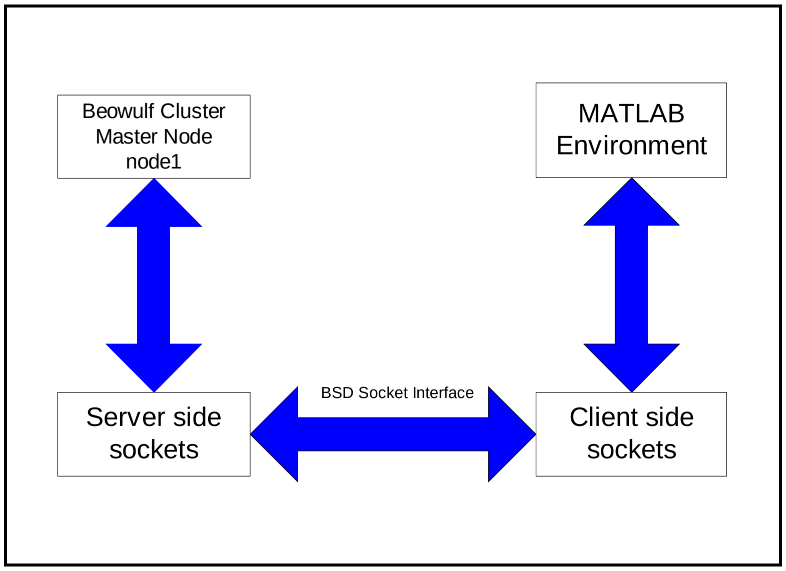


Figure 5-6 : Cluster - Matlab Interface

5.2 AVDS Interface

Towards the goal of creating a graphical front end to the adaptive flight control scheme, a commercially available visualization package, Aviator Visual Design Simulator (AVDS), designed as a joint project between Artificial Horizons Inc. and the United States Air Force is chosen. Although other graphics packages are available, such as X-Plane etc, this package was chosen mainly due to previous experience in using the package, within the research group. This

AVDS package gives the user the freedom to integrate their own control systems and aircraft model into the simulation and visualization of aircraft dynamics.

One of the goals of this phase is the creation of a "communication channel" between AVDS and the adaptive flight control scheme codes to allow a smooth exchange of data. The communication channel is based on a client-server model and uses BSD sockets for inter-process communication. The AVDS environment set as the client while the master node in the Beowulf cluster is set as the master. Figure 5-8 shows a schematic representation of the scheme. Within the source code that generates the aircraft dynamics and visualization in AVDS, the client side socket is initialized, during the initialization phase of AVDS. This client side socket then waits for the server to send the parameters of the simulation that AVDS uses to determine the length of simulation and the task that the user wants to run. If the task that is to be run is the AFDIA procedure, then the AVDS client receives the appropriate parameters from the Matlab environment, via the Beowulf cluster master node.

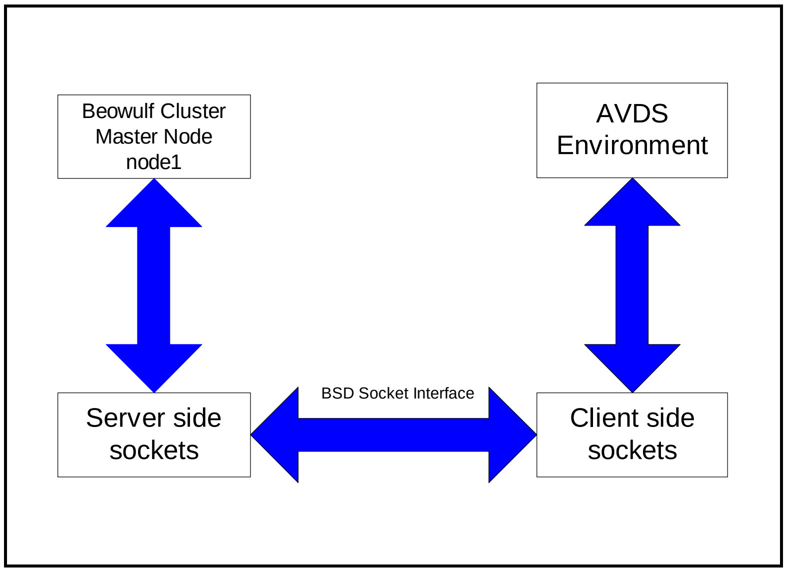


Figure 5-7 : Cluster – AVDS Interface

5.3 Beowulf Cluster Setup

The SFDIA and the AFDIA tasks are designed to be executed on a distributed computing environment. A Beowulf cluster made up of commodity components is used as the distributed computing environment within this research effort.

The cluster is made up of seven distinct compute nodes, *node1-7*and while *node8* supports the AVDS simulation/visualization. It must be noted that node8 is not explicitly a part of the cluster as it does not communicate directly with the compute nodes on the cluster and communicates with only the main node by means of sockets. The visualization node could theoretically exist on an entirely different network, although this would not be advantageous considering the network traffic and congestion it might encounter. Each of the compute nodes is a standalone computer, running RedHat 9.0 distribution of the Linux operating system, with kernel version 2.4.20-8. Table 5-1 lists the specifications of the compute nodes on the cluster. As mentioned earlier in Chapter 4, the inter-node communication on the cluster is based on the Message Passing Interface. For this research effort, MPICH 1.2.6 is used. Some of the features of this version of the portable implementation of the MPI standard include

- Complete MPI 1.2 standard compliance, including canceling of *send*
- Support for MPMD programs and heterogeneous clusters
- Support for a wide variety of computing environments, including clusters of Symmetric Multi Processors and massively parallel processors.

Table 5-1 : Specifications of Beowulf Cluster

Node	O.S	CPU	Memory	Task
node1	Linux	500 MHz	128 MB	Master/Compute node
node2	Linux	266 MHz	64 MB	Compute Node
node3	Linux	266 MHz	64 MB	Computer Node
node4	Linux	266 MHz	64 MB	Compute Node
node5	Linux	266 MHz	64 MB	Compute Node
node6	Linux	266 MHz	64 MB	Compute Node
node7	Linux	266 MHz	64 MB	Compute Node
node8	Windows	2000 MHz	1024 MB	AVDS

Each node on the cluster is equipped with a 10/100 Mbps Ethernet card and the nodes are interconnected via a Cisco Systems, Catalyst 3560G series 48 port switch.

The cluster is setup as a private network with the IP addresses of all the 8 nodes being 192.168.1.1 through 192.168.1.8. The 192.168.1.8 node is the Windows 200 based graphical interface machine. On the cluster, node1 is set as the master node and is in charge of setting up the communication channels between the compute nodes on the cluster and the graphics node on the Windows machine.

The cluster built for the purpose of implementing the fault tolerant flight control systems in parallel is shown in the following figure.

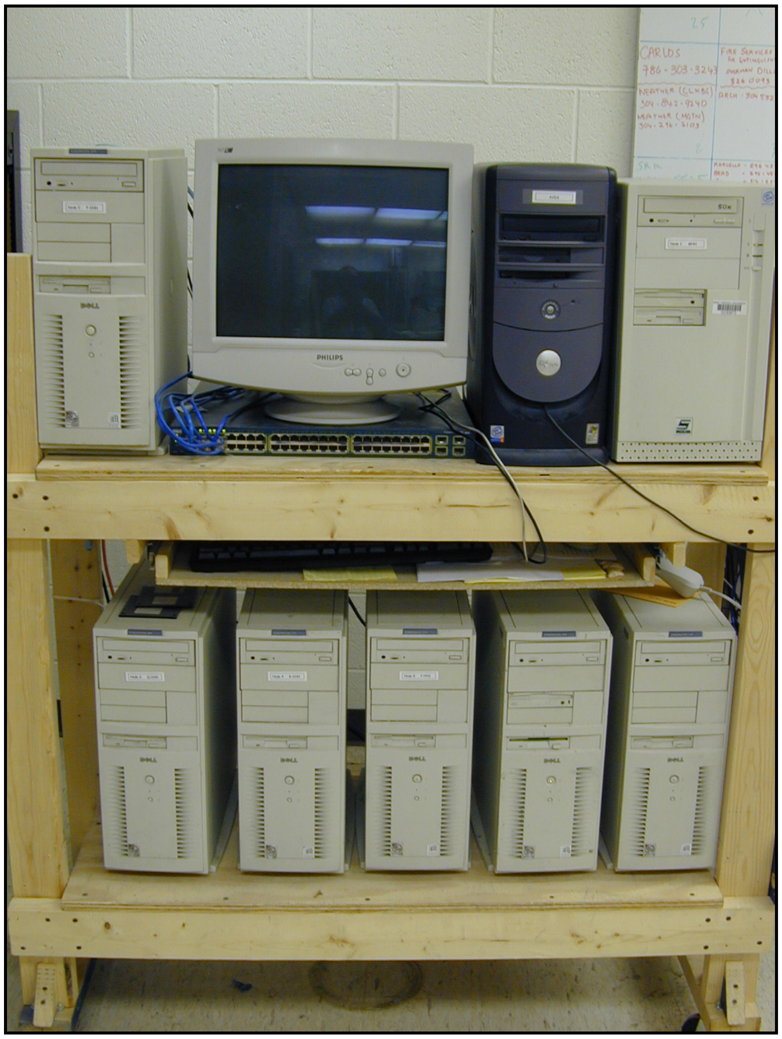


Figure 5-8 : Cluster for Parallel Implementation of FDIA

The procedure for setting up a simulation for any of the tasks is as follows. The source code for the task is first transferred to the master node and then compiled into a binary using the MPI directive

mpicc –o foo foo.c; rm –f foo.o

The compiled binary is then transferred to all the other compute nodes on the cluster. The compute nodes, including the master node are setup with a similar directory structure. Before the binary can be executed on the cluster, the mpd daemons have to be started on all the compute nodes. Once the daemons are started, they are connected in a ring, with the last node entering the ring between the last previously entered node and the first node. On the first node, the mpd daemons are started by issuing the command

mpd &

and a subsequent call to **mpdtrace** gives the port number at which the first daemon is listening. The other nodes connect to this port on the host machine, *node1*, using the command

mpd –h node1 –p port_number

Once all the compute nodes are in the ring, the binary can be executed on any number of processes using the command

mpirun –np number_of_processes foo

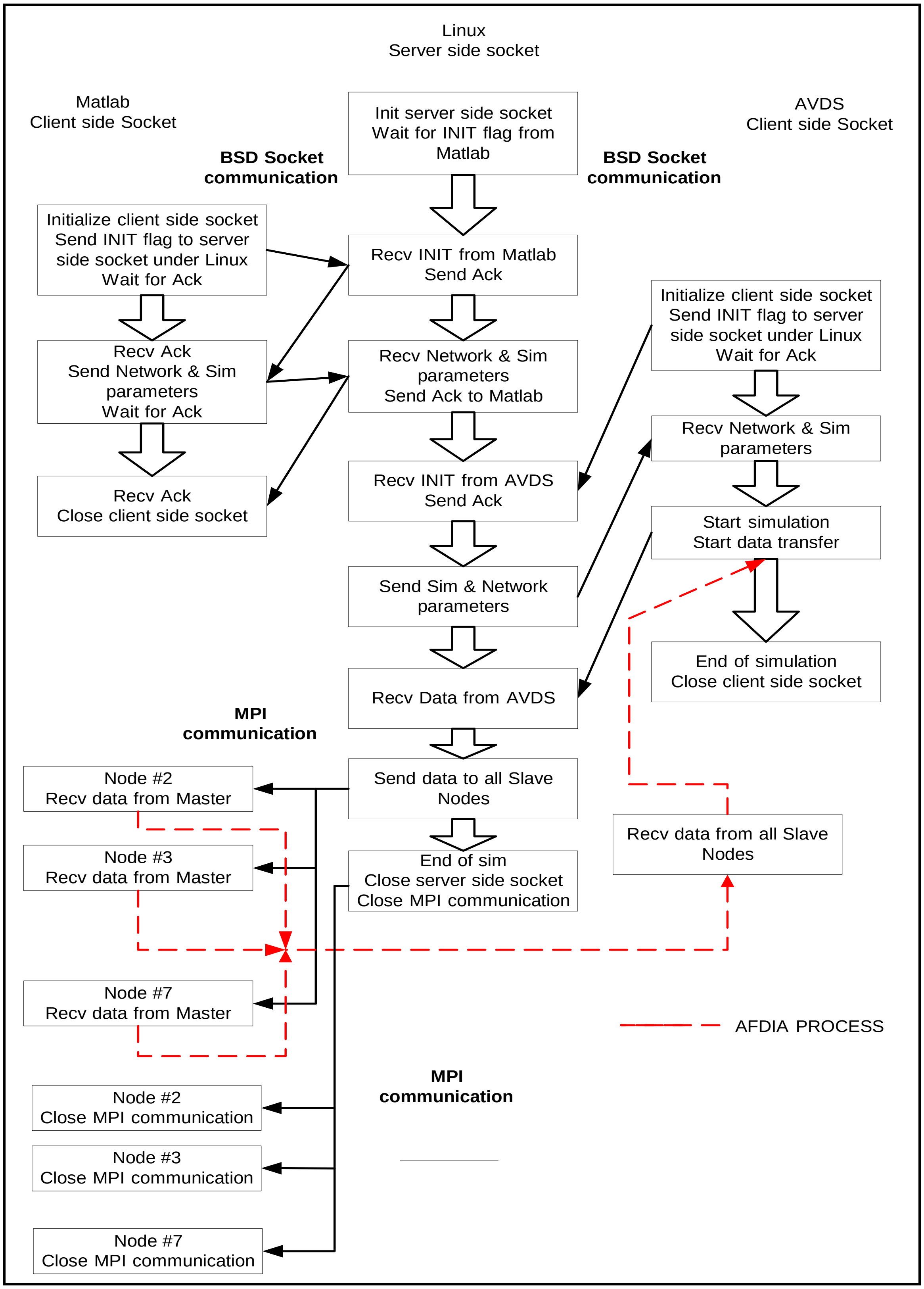
This command would set the binary running on all the nodes with the code is setup in such a way to execute a server side socket on the master node. Once the server socket is up and running on the master node, by clicking on the "*Start Data Transfer*" button on the main GUI in Matlab would create a client side socket within Matlab. This client would then connect to the server side socket on the master node. Once the master node receives the Initialization flag from the Matlab client, it sends an acknowledgement back to the client and then waits to receive the simulation parameters. Once the master node receives the simulation parameters, it distributes the same to all the compute nodes on the cluster using MPI_Send directive. Upon receiving the simulation parameters, all the compute nodes in the ring, including the master node, then proceed to initialize their respective NN structures, extracting whatever parameters they need, from the simulation parameters that was sent across from the Matlab side.

The master node is now ready to receive the initialization flag from AVDS. Once the flag is received from AVDS, the master node sends appropriate simulation parameters to the AVDS environment, including the task identification number and the total time of simulation. If the task on hand is the AFDIA task, then the master node also sends the parameters that describe the failure, including the time of failure, the surface that failed, the magnitude of failure and the effectiveness of the surface after failure. AVDS receives all this information through its client side socket and then starts the simulation.

The simulation within AVDS is setup so that it allows the "pilot" to "fly" the aircraft to an altitude of 3000 ft, before data transfer takes place. From the time instant tha the aircraft takes off till it reaches 3000 ft, there is no transfer of data from AVDS to the main and compute nodes on the cluster. As a consequence, no NN training takes place. Once the aircraft reaches 3000 ft, AVDS sends out data frames at a rate of 30Hz across the socket to the master node. The master node collects the data from AVDS and distributes it to all the compute nodes in the cluster. Once all the data is distributed to the compute nodes, the master node then proceeds to execute the code for the MNN while the other compute nodes execute the codes for the DNNs, based on their individual ranks. For example, in the case of the SFDIA procedure, the node with rank 1 would execute the code for the P-DNN, the node with rank 2 would execute the code for Q-DNN and the node with rank 3 would execute the code for R-DNN. Once the forward and backward pass of each DNN is complete, they send their respective outputs to the MNN. Within the MNN on the master node, once all the estimates of the rate are received from the DNNs, the MNN proceeds to compute the error metrics and perform the procedure for failure detection and identification. Upon failure detection and/or identification, the MNN then proceeds to set appropriate flags. These flags are then sent across to the DNNs during the next cycle of data transfer.

In the case of AFDIA, the process is similar to that of the SFDIA, except that once the error metrics are computed within the MNN and error flags are set or not set, this data is sent back to the AVDS environment. During the next cycle, AVDS checks the flags to determine if a failure has been declared and takes appropriate actions, based on the status of the flags.

Figure 5-9 shows the detailed working of the communication scheme between the Matlab GUI, the Master node on the cluster and the AVDS environments for the parallel implementation of the SFDIA and AFDIA schemes. Figure 5-10 shows the communication scheme between the Matlab GUI, the Master node on the cluster and the AVDS environments for the sequential training implementation of the NNs for SFDIA purposes.



Linux
Server side socket
Matlab
Client side Socket
AVDS
Client side Socket
BSD Socket communication
BSD Socket communication
Init server side socket Wait for INIT flag from Matlab
Initialize client side socket Send INIT flag to server side socket under Linux Wait for Ack
Recv INIT from Matlab Send Ack
Recv Ack Send Network & Sim parameters Wait for Ack
Recv Network & Sim parameters Send Ack to Matlab
Recv Ack Close client side socket
Initialize client side socket Send INIT flag to server side socket under Linux Wait for Ack
Recv Network & Sim parameters
Recv INIT from AVDS Send Ack
Start simulation Start data transfer
Send Sim & Network parameters
End of simulation Close client side socket
Recv Data from AVDS
MPI communication
Send data to all Slave Nodes
Node #2 Recv data from Master
Node #3 Recv data from Master
End of sim Close server side socket Close MPI communication
Recv data from all Slave Nodes
Node #7 Recv data from Master
AFDIA PROCESS
MPI communication
Node #2 Close MPI communication
Node #3 Close MPI communication
Node #7 Close MPI communication

Figure 5-9 : Details of Matlab-Master Node-AVDS communication scheme

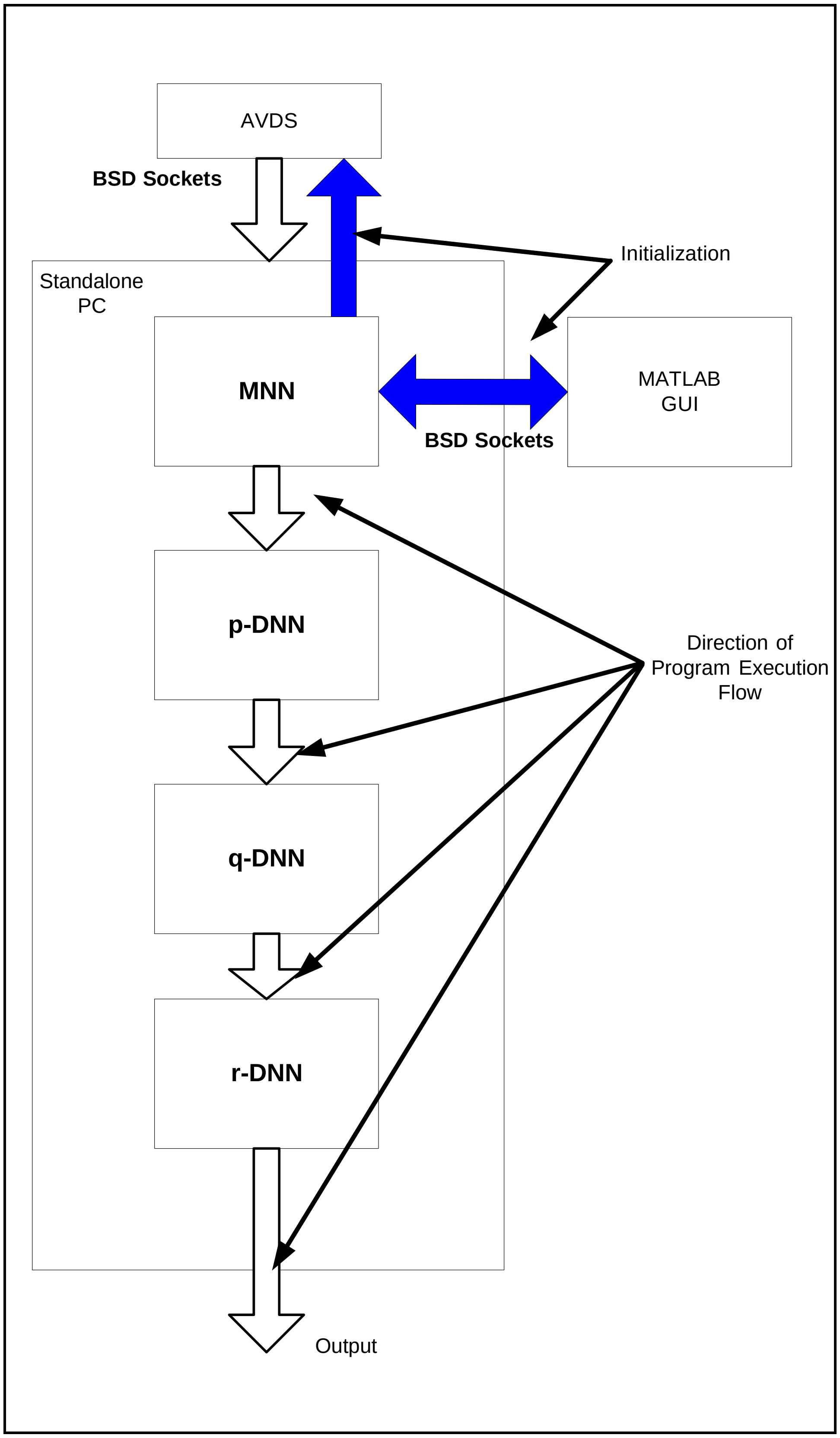


Figure 5-10 : Sequential Implementation of Adaptive Flight Control System Schemes

5.4 Modeling of Parallelization

Parallelization of a sequential algorithm conveys different meanings under different contexts and cannot be generalized to apply for all contexts. The goals of parallelization differ from each application and so, different criteria should be used as yardsticks for interpreting the results of parallelization.

Within this research effort, the goal is to provide a proof of concept implementation of a parallel distributed computation and control environment for the purpose of realizing a fault tolerant flight control system. This fault tolerance is geared towards detecting, identifying and accommodating for failures on systems of sensors without physical redundancy and failures on the control surfaces of an aircraft. Towards this goal, a Beowulf cluster based parallel computing environment was built from COTS components. Figure 5-10 gives the logical execution order of a sequential implementation of the fault tolerant flight control scheme on a single node.

The first stage within this sequential implementation is the initialization of the NN structures with the data from the Matlab GUI. Once the initialization parameters are received from the Matlab environment, and the NNs initialized, the node then waits for the AVDS simulation environment to start and initialize its client side sockets. The AVDS environment is setup in such a way that it does not transmit data back to the master node until the aircraft is above an altitude of 3000 feet. Once the aircraft breaches the 3000 ft barrier, it starts to send frames of data back to the master node.

If T_{MNN}, T_{PDNN}, T_{QDNN} and T_{RDNN} are the times of execution of the MNN, PDNN, QDNN and the RDNN respectively, then the total time that it takes to go through one cycle of execution of the forward and backward paths of the NNs is the sum of all the individual execution times. If t_x is a time measure of uncertainties and jitter involved in the transfer of data to and from main memory, the time taken for startup and other operations, and t_1 is the time taken for the socket communication between AVDS and the master node, then the total time taken for one cycle of the sequential implementation can be given by

$$T_s = t_1 + T_{MNN} + T_{PDNN} + T_{QDNN} + T_{RDNN} + T_{ERR} + t_x \qquad \text{Eq. 5.1}$$

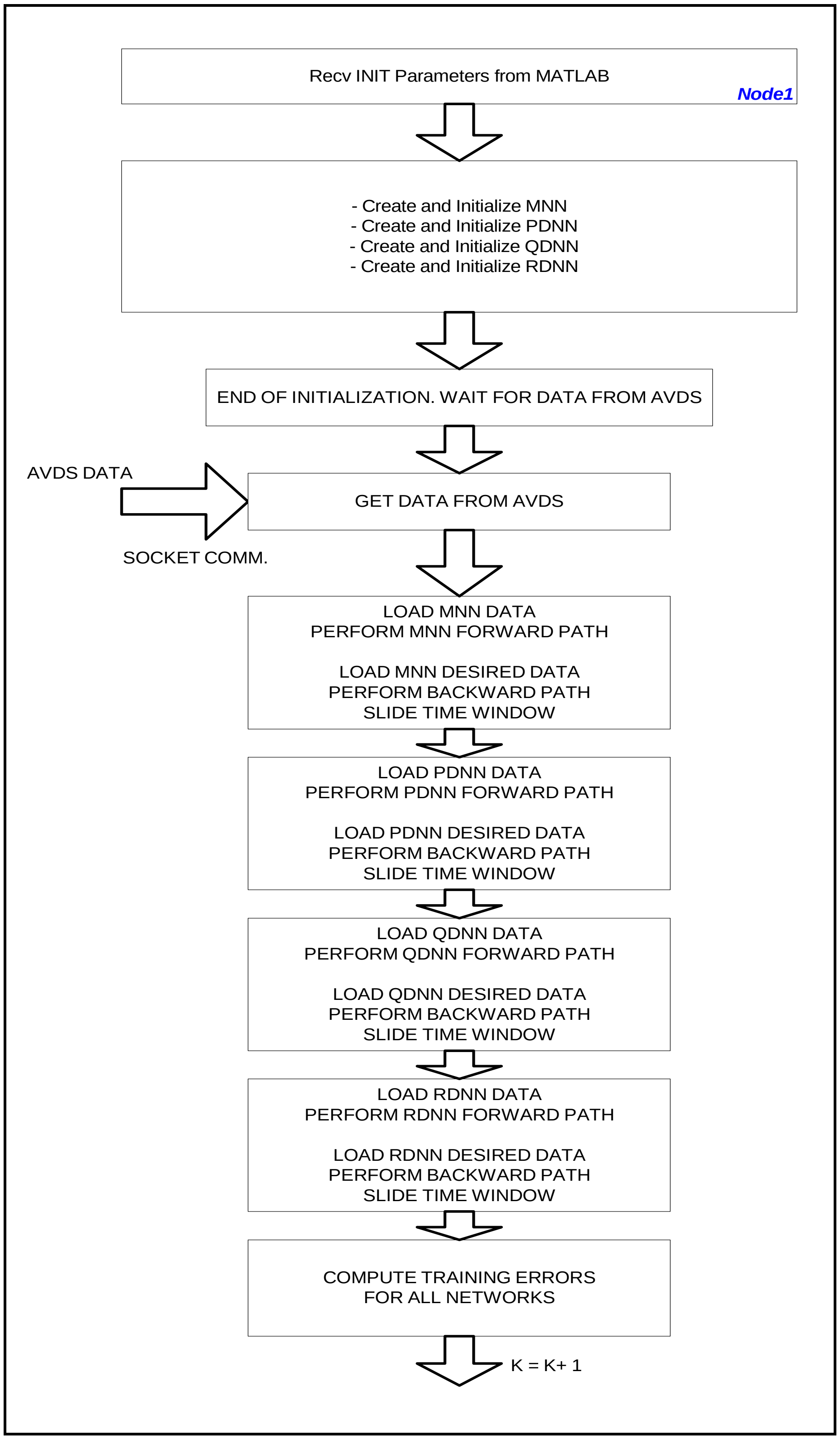


Figure 5-11 : Sequential Algorithm

Consider the parallel implementation of the SFDIA algorithm. The communication sequence and the order of execution of the NN structures MNN, PDNN, QDNN and the RDNN is shown in Figure 5-11.

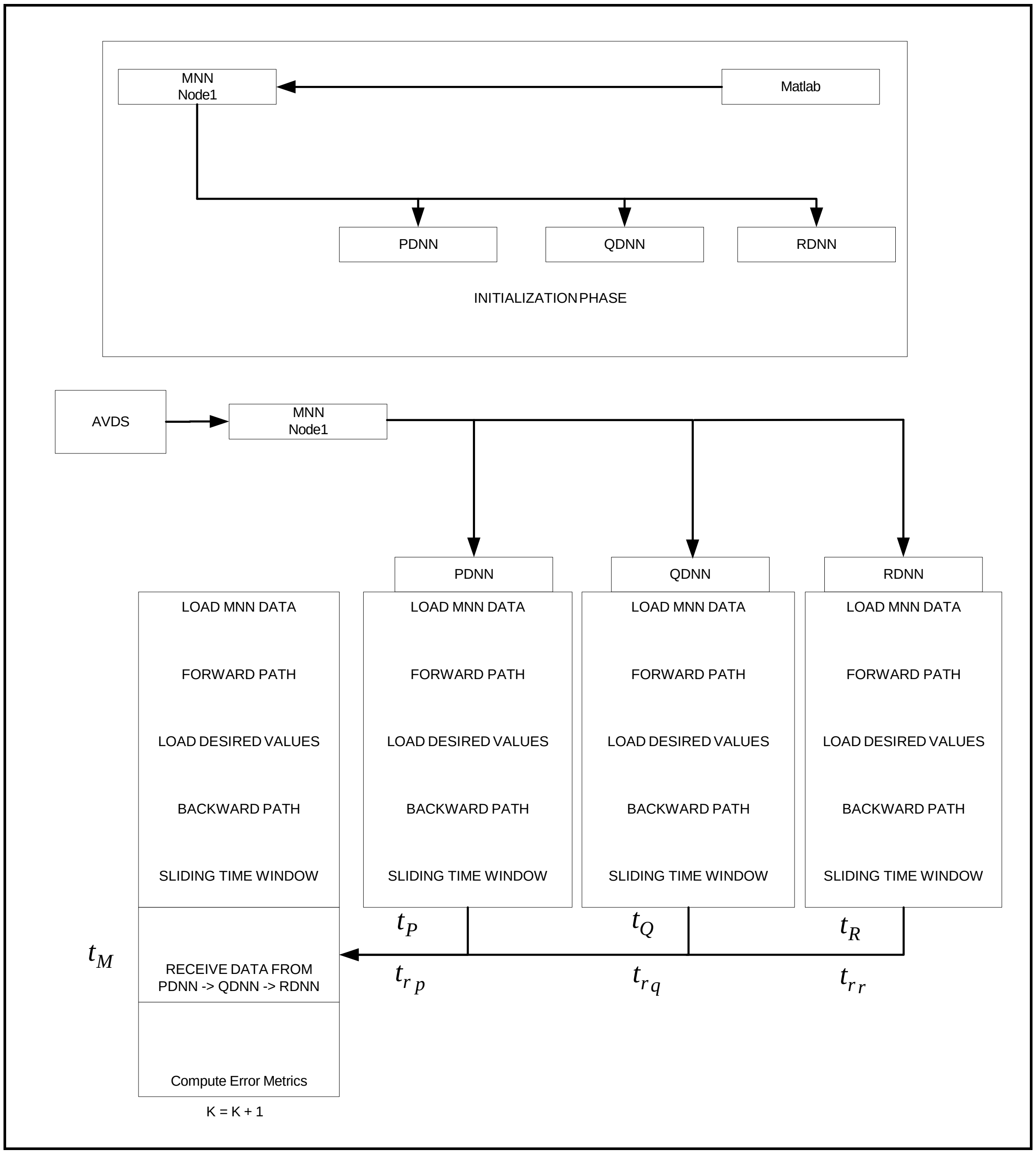


Figure 5-12 : Parallel Implementation Timing Chart

In the following analysis, designating t_1 to be the time taken for socket communication between the AVDS environment and the main node on the Beowulf cluster, t_M, t_P, t_Q, t_R to be the time taken by the MNN, PDNN QDNN and the RDNN to complete one cycle of learning and $t_{r_p}, t_{r_q}, t_{r_r}$ the time taken for the MPI based communication between the master node and the slave nodes on the cluster, we can derive an expression for the maximum time that it would take to complete one cycle of training for all the networks in the cluster. To begin the analysis, consider communication between the MNN and the PDNN only, the MNN will have the roll rate estimate data from PDNN after either $t_M + t_{r_p}$ or $t_P + t_{r_p}$. Depending on whether the MNN finished computation earlier than the PDNN or PDNN finishes its computation earlier, the latest time at which MNN will have the data from PDNN can be given as

$$\max(t_M + t_{r_p}, t_P + t_{r_p}) \qquad \text{Eq. 5.2}$$

Now considering three networks, MNN, PDNN and the QDNN, the MNN will have the roll rate estimate data from PDNN and pitch rate estimate data from the QDNN after either $t_M + t_{r_p} + t_{r_q}$ or $t_P + t_{r_p} + t_{r_q}$ or $t_Q + t_{r_q}$, depending on which of the three processes finish their computation earliest. Thus the latest time that the MNN will have the data from all the PDNN and the QDNN can be given as

$$\max(t_M + t_{r_p} + t_{r_q}, t_P + t_{r_p} + t_{r_q}, t_Q + t_{r_q}) \qquad \text{Eq. 5.3}$$

Similarly, considering all the four networks MNN, PDNN< QDNN and the RDNN, the MNN will have the roll rate estimate data from PDNN, pitch rate estimate data from the QDNN and the yaw rate estimate data from the RDNN after either $t_M + t_{r_p} + t_{r_q} + t_{r_r}$ or $t_P + t_{r_p} + t_{r_q} + t_{r_r}$ or $t_Q + t_{r_q} + t_{r_r}$ or $t_R + t_{r_r}$. Thus the latest time that the MNN will have the data from all the three DNNs can be given as

$$\max(t_M + t_{r_p} + t_{r_q} + t_{r_r},\ t_P + t_{r_p} + t_{r_q} + t_{r_r},\ t_Q + t_{r_q} + t_{r_r},\ t_R + t_{r_r}) \qquad \text{Eq. 5.4}$$

And the total time for the parallel implementation to complete one cycle of learning can be given as

$$\begin{aligned} T_P = t_1 + \max(t_M + t_{r_p} + t_{r_q} + t_{r_r}, t_P + t_{r_p} + t_{r_q} + t_{r_r}, t_Q + t_{r_q} + t_{r_r}, t_R + t_{r_r}) \\ + t_{x_{MNN}} + t_{x_{PDNN}} + t_{x_{QDNN}} + t_{x_{RDNN}} \end{aligned} \qquad \text{Eq. 5.5}$$

Since both the sequential and parallel implementations of the algorithms wait for the data to arrive from AVDS, we can safely remove that wait time from the total time taken by both implementations.

$$T_s = T_{MNN} + T_{PDNN} + T_{QDNN} + T_{RDNN} + T_{ERR} + t_x \qquad \text{Eq. 5.6}$$

$$\begin{aligned} T_P &= \max(t_M + t_{r_p} + t_{r_q} + t_{r_r}, t_P + t_{r_p} + t_{r_q} + t_{r_r}, t_Q + t_{r_q} + t_{r_r}, t_R + t_{r_r}) \\ &+ t_{x_{MNN}} + t_{x_{PDNN}} + t_{x_{QDNN}} + t_{x_{RDNN}} \end{aligned} \qquad \text{Eq. 5.7}$$

Thus, the speedup that can be achieved by the parallel implementation of the sequential algorithm can be expressed as

$$S = \frac{T_s}{T_P} \qquad \text{Eq. 5.6}$$

Since the parallel implementation of the fault tolerant flight control schemes for SFDIA and AFDIA purposes are set to run a predefined frequency of 30 Hz and since data synchronization is absolutely important, blocking calls for Send and Receive operations are used.

From experimental data, the average time taken for the completion of one cycle of learning for the sequential implementation is 0.0258 s. the average time for the completion of one cycle of learning for the parallel implementation is also 0.0258 s. Thus, the speedup is given by S = 1. Figure 5-12 gives a comparison of the execution times for the sequential and parallel implementations of the training cycle of four NNs, namely the MNN, PDNN, QDNN and the RDNN. Within this research work, the timing directives used to measure the time taken to comple one cycle of NN training for the sequential and parallel implementations included the time that the MNN waits to receive the frame of data from AVDS. This influences the timing profile of both the sequential and parallel implementations, leading to similar execution times for the sequential and parallel implementations.

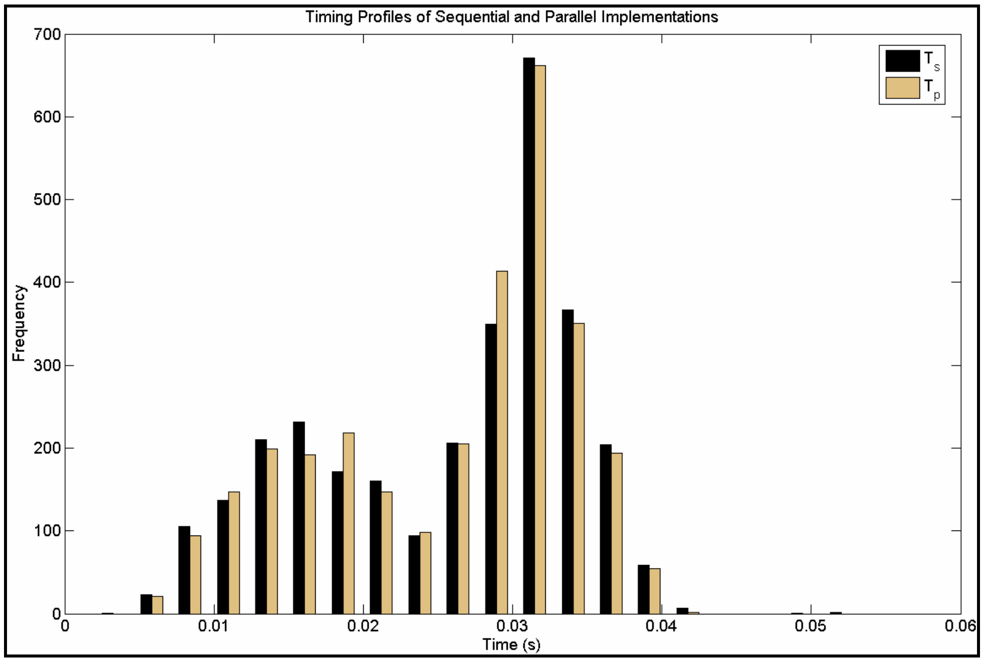


Figure 5-13 : Execution times for Sequential and Parallel Implementations

Chapter 6 - Sensor and Actuator Failure Detection Identification and Accommodation

The main objectives of this research are the design and simulation of adaptive flight control schemes for the purposes of *Sensor Failure Detection Identification and Accommodation* (SFDIA) and *Actuator Failure Detection Identification and Accommodation* (AFDIA) on a distributed computational architecture. As previously mentioned in Chapter 1, this research effort is divided into different phases, with incremental goals in each phase, leading to the implementations of the SFDIA and AFDIA schemes on the distributed computational platform.

This chapter discusses the structure of the SFDIA and AFDIA algorithms, their implementations and the results from the on-line simulation tests.

As mentioned in Chapter 4, the MLP neural network architecture is used along with the EBPA as the training algorithm as the adaptive component of the fault tolerant flight control scheme. The SFDIA scheme consists of four distinct neural networks, namely the *Main Neural Network* (MNN), and three *De-Centralized Neural Networks* (DNNs) P-DNN, Q-DNN and R-DNN, representing the *roll*, *pitch* and *yaw* gyros.

The functionalities of the networks are designed so that the *Main Neural Network* will learn the time histories of *pitch*, *roll* and *yaw* sensors, based on sensor outputs from time instant '*k-1*' till '*k-l*', from sensor systems onboard the aircraft. The time interval *l* is the training pattern time window and exerts a huge influence on the time needed for training the network. The overall outputs of the network are the estimates of the *roll*, *pitch* and *yaw* gyros at current time instant '*k*'. On the other hand, the *De-Centralized Neural Networks* would estimate the outputs of their respective sensors at the current time instant '*k*', based on inputs from other sensors onboard the aircraft. The most important distinction between the MNN and the DNNs is that the MNN uses the values of the *roll*, *pitch* and *yaw* gyros, albeit from previous time instances, among others, to estimate the current values of the roll, pitch and yaw rates. The DNNs on the other hand, DO NOT use values from the sensor they are representing, to estimate the current outputs. For example, the Q-DNN can have any measurements from any sensor onboard the aircraft, except the *pitch* rate gyro. Similarly for the roll and yaw gyros, the inputs to the DNNs could be the measurements from any sensor onboard the aircraft, except the roll rate and the yaw rate respectively. This ensures that spurious inputs from a failed sensor cannot pass into the network that is trying to estimate the nominal values of the corresponding sensor and corrupt the training process, keeping it trustworthy. Failure detection and identification is achieved by monitoring the errors in the estimates of the rate gyro outputs, generated by the MNN and the DNNs.

6.1 Types of Sensor Failures

This scope of this research in terms of the sensors whose behavior is to be emulated by the networks is limited to only the rate gyroscopes, namely the *pitch*, *roll* and *yaw* gyros that are assumed to be without physical redundancy. The output from a typical rate gyro is given as

$$Rate = K_g (V_r(t) - K_0)$$

where:

$Rate = Angular\ Velocity\ (\deg/\sec)$

K_g *is the rate gyro gain* $((\deg/\sec)/volt)$

K_0 *is the rate gyro offset* $(volt)$

V *is the voltage offset from rate gyro* $(Volt)$

The accuracy of rate gyroscopes is characterized by two parameters, namely the drift and the scale factor. Gyroscope drift is the ability of the gyro to reference all rate measurements to the nominal zero rate measurement [48] and appears as an additive term in the gyro output. As a consequence, any drift from the actual value would cause the angular error to accumulate, resulting in faulty output from the gyroscope. The scale factor describes the ability of the gyro to accurately sense the angular velocity at different angular rates and describes the sensitivity of the sensor in deg/sec/volts. The accuracy of this parameter directly translates into the accuracy of the output as it is used to convert the voltage output from the gyro into angular rate. If system imperfections are present, it can result in variations of the scale factor and these variations would appear as errors in the calibration of the output voltage versus the angular rate.

From the above descriptions, the generic sensor failures can be modeled as

1. *Additive* sensor failure: In this type of failure, a constant value appears as a bias in the nominal sensor output and case can be modeled as $X_{failure,i} = X_{nom,i} + \rho n_i$, where, n_i is the direction vector of the faulty sensor and ρ is the magnitude of the failure, positive or negative.

2. *Multiplicative* sensor failure: This failure can be modeled by multiplying the sensor's nominal value by a constant factor and is given as $X_{failure,i} = (1 + kn_i)X_{nom,i}$, where k is the multiplying factor for the $i-th$ sensor. This multiplicative failure can be further divided into two, based on the direction vector n_i as
    a. Step type sensor failure, where $n_i = 1$
    b. Ramp type failures, where

$$n_i = \begin{cases} \dfrac{t - t_{f1}}{t_{f2} - t_{f1}} & t_{f1} \le t \le t_{f2} \\ 1 & t >= t_{f2} \end{cases} \qquad \text{Eq 6.1}$$

and t_{f1}, t_{f2} are the initial and final time instants of ramp type failures.

6.2 Sensor Failure Detection Identification and Accommodation

The Sensor Failure Detection Identification and Accommodation process is described in the following sections. During the training process, the networks produce outputs that estimate the *pitch*, *roll* and *yaw* gyro measurements at the current time instant and the interconnection weights and thresholds within the network are adjusted, as described in Chapter 4, in such a manner as to reduce the estimation error. In the original failure detection scheme, a quadratic parameter $MQEE(k)$, the sum of the squares of the difference between the outputs from the MNN and the corresponding sensor values, is used for sensor failure detection (*SFD*). The $MQEE(k)$ is given as

$$MQEE(k) = \frac{1}{2} \sum_{i=1}^{\#ofDNNs} (Y_i(k) - O_{i,MNN}(k))^2 \qquad \text{Eq 6.2}$$

$$= \frac{1}{2}[(p(k) - \hat{p}_{MNN}(k))^2 + (q(k) - \hat{q}_{MNN}(k))^2 + (r(k) - \hat{r}_{MNN}(k))^2] \qquad \text{Eq 6.3}$$

Under nominal operating conditions, when there is no sensor failure and the networks are trained well, *MQEE*, which is a function of time, is close to zero, but when a sensor failure occurs, the outputs from the networks differs significantly from the sensor values, resulting in values of *MQEE* is significantly different from zero. Also, under nominal conditions, the learning process is active on all the networks, including the *MNN* and the three *DNNs*. However, when the *MQEE* value exceeds a certain pre-defined threshold, a failure is declared, the learning processes on all the networks is halted and subsequently, the sensor values are replaced by the outputs from the *DNNs*, wherever the sensor measurements are used. This would constitute the failure detection phase of the adaptive flight control scheme, where the occurrence of a generic sensor failure is detected, but the actual failed sensor is not identified. The next step is the identification of the failed sensor and can be achieved by monitoring the absolute value of the estimation error at each *DNN*. If this value, at any of the *DNNs*, exceeds a pre-defined threshold, the sensor failure is then identified on the corresponding sensor. The estimation error of the *DNNs* is given by the following expression

$$DQEE_x(k) = \frac{1}{2}(x(k) - \hat{x}(k))^2 \qquad \text{Eq 6.4}$$

where $x = p, q, r$

Once a sensor failure is identified, the next step involves the accommodation process for the failed sensor, where the output from the x-DNN is used to replace the measurement from the faulty sensor. This output is also used as an input to the *MNN* and also to any other *DNN* using the sensor measurement as an input. This approach of using error statistics of both the *MNN* and the *DNNs* would reduce the rate of false alarms in the failure detection and identification process.

Along with the original scheme for fault detection, a modified scheme is proposed to improve the detection capabilities of the system for soft failures [49]. In this modified scheme, the quadratic estimation error *MQEE* used in the original scheme is replaced by a different quadratic estimation error, given by

$$OQEE(k) = \frac{1}{2}\sum_{i=1}^{\#ofDNNs}(O_{i,MNN}(k) - O_{i,DNN}(k))^2 \qquad \text{Eq 6.5}$$

$$= \frac{1}{2}[(\hat{p}_{MNN}(k) - \hat{p}_{DNN}(k))^2 + (\hat{q}_{MNN}(k) - \hat{q}_{DNN}(k))^2 + (\hat{r}_{MNN}(k) - \hat{r}_{DNN}(k))^2] \qquad \text{Eq 6.6}$$

The need for an error statistic different from the MQEE can be justified by an analysis of a typical ramp type sensor failure, for example, when one of the gyros fail, the output $\hat{x}_{MNN}$ would tend to track the corrupted sensor value since the *MNN* has the outputs from the failed sensor as one of its input. As a result, the parameter MQEE does not provide accurate failure detection, as the difference between $x(k)$ and $\hat{x}_{MNN}$ is not significant enough to exceed the predefined threshold. On the other hand, the output of the *x-DNN* would try to track the “nominal” value of the x-gyro, as it does not receive any measurement from the faulty sensor as an input. This difference between the outputs of the *MNN* and the *x-DNN* would cause the parameter *OQEE* to spike at the instant of failure and so can be used to detect a failure more reliable than when using the *MQEE* parameter. A block diagram of the scheme is shown in Figure 6-1.

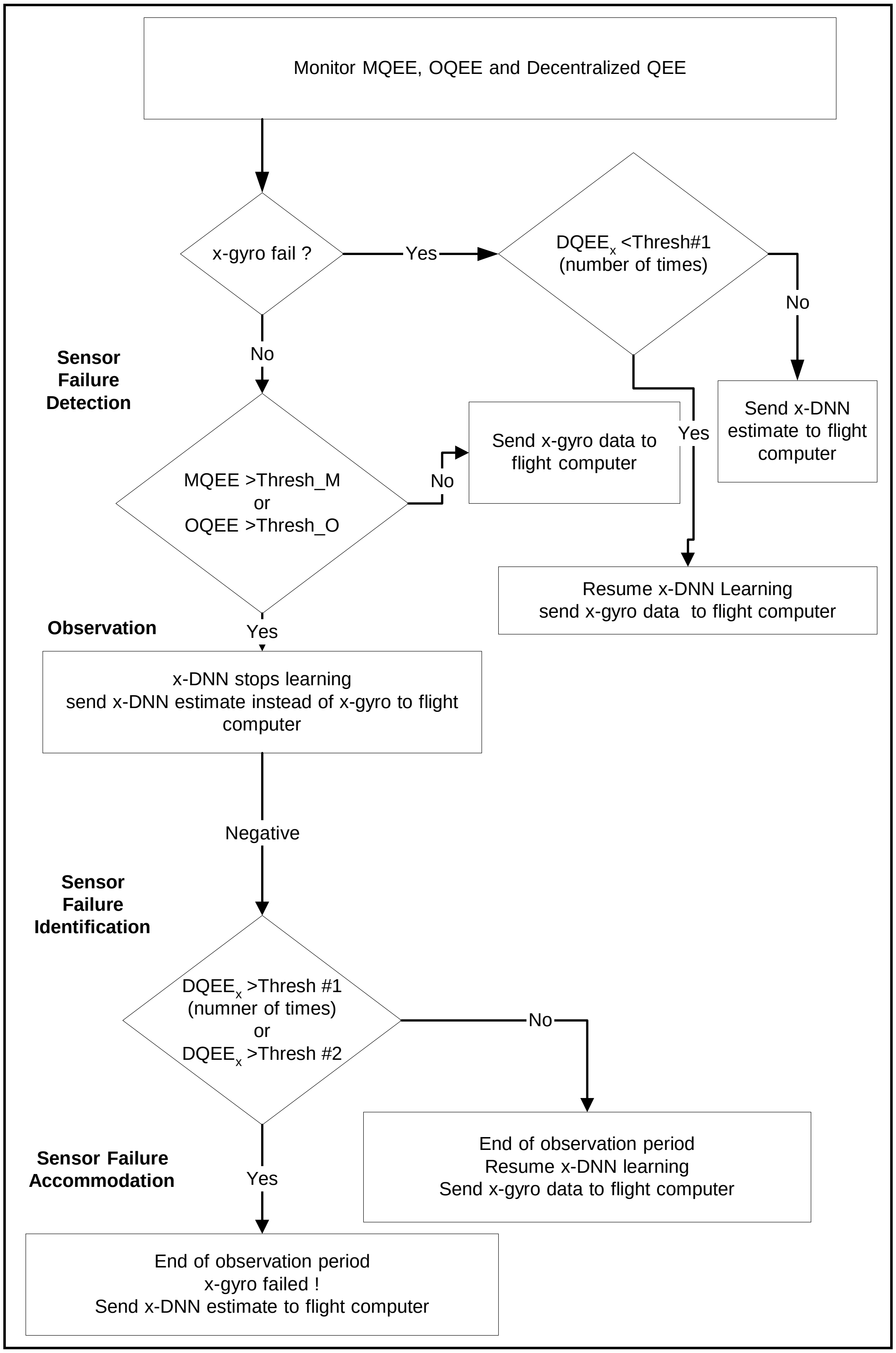

Monitor MQEE, OQEE and Decentralized QEE
x-gyro fail ?
Yes
DQEE$_x$ <Thresh#1 (number of times)
No
Sensor Failure Detection
No
Send x-gyro data to flight computer
Yes
Send x-DNN estimate to flight computer
MQEE >Thresh_M or OQEE >Thresh_O
No
Resume x-DNN Learning send x-gyro data to flight computer
Observation
Yes
x-DNN stops learning send x-DNN estimate instead of x-gyro to flight computer
Negative
Sensor Failure Identification
DQEE$_x$ >Thresh #1 (numner of times) or DQEE$_x$ >Thresh #2
No
Sensor Failure Accommodation
Yes
End of observation period Resume x-DNN learning Send x-gyro data to flight computer
End of observation period x-gyro failed ! Send x-DNN estimate to flight computer

Figure 6-1 : Block Diagram of SFDIA Scheme

6.3 Results of Offline Training for SFDIA purposes

The NNs MNN, P-DNN, Q-DNN and R-DNN were trained offline for 100 epochs, using flight data with 5000 data points, generated at 30 Hz, from the AVDS environment. These trained networks were then tested every 10 epochs, starting from the 50th epoch, with data that it had not seen during its training cycle. During the training cycle, the weights and thresholds of the networks were saved after every 10 epochs. These pre-trained network structures were then used within the on-line Sensor Failure Detection, Identification and Accommodation scheme running on the cluster. The following figures Figure 6-2 through Figure 6-19 are representative of the training and testing results for the MNN. Once the training was completed, the last set of weights and thresholds from each network was saved as the starting point for the on-line learning during the SFDIA and AFDIA tasks.

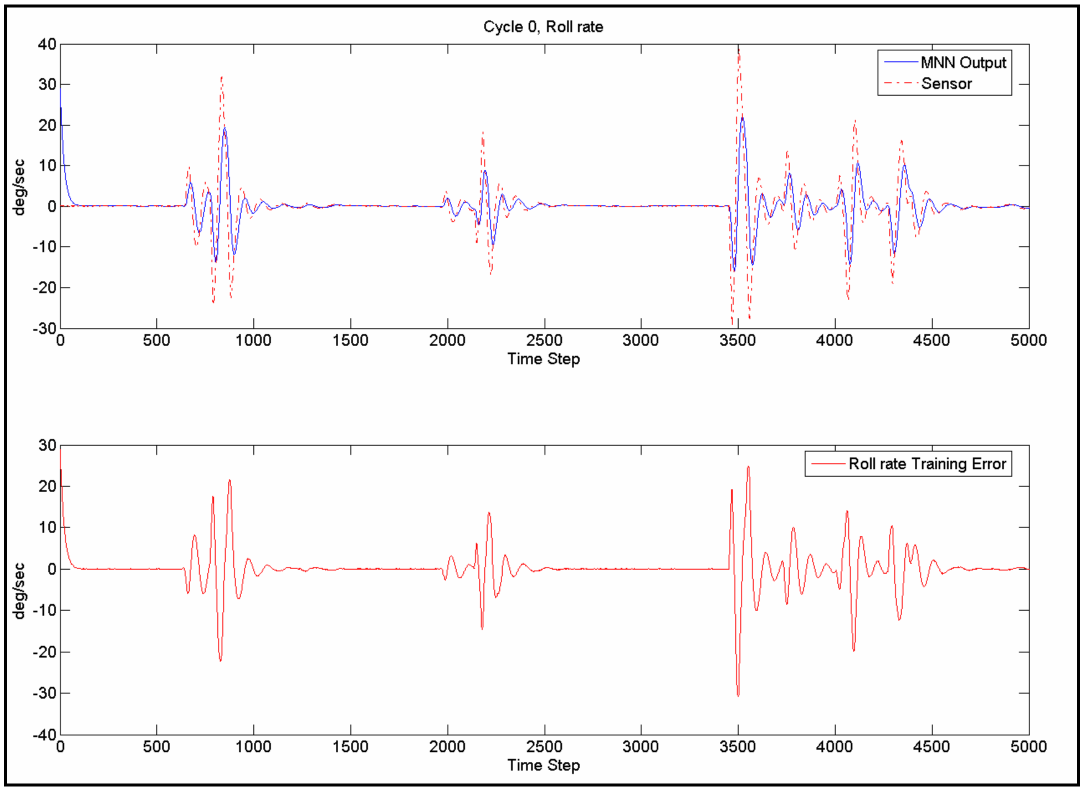


Figure 6-2 : MNN Output, Cycle 0, Roll Rate

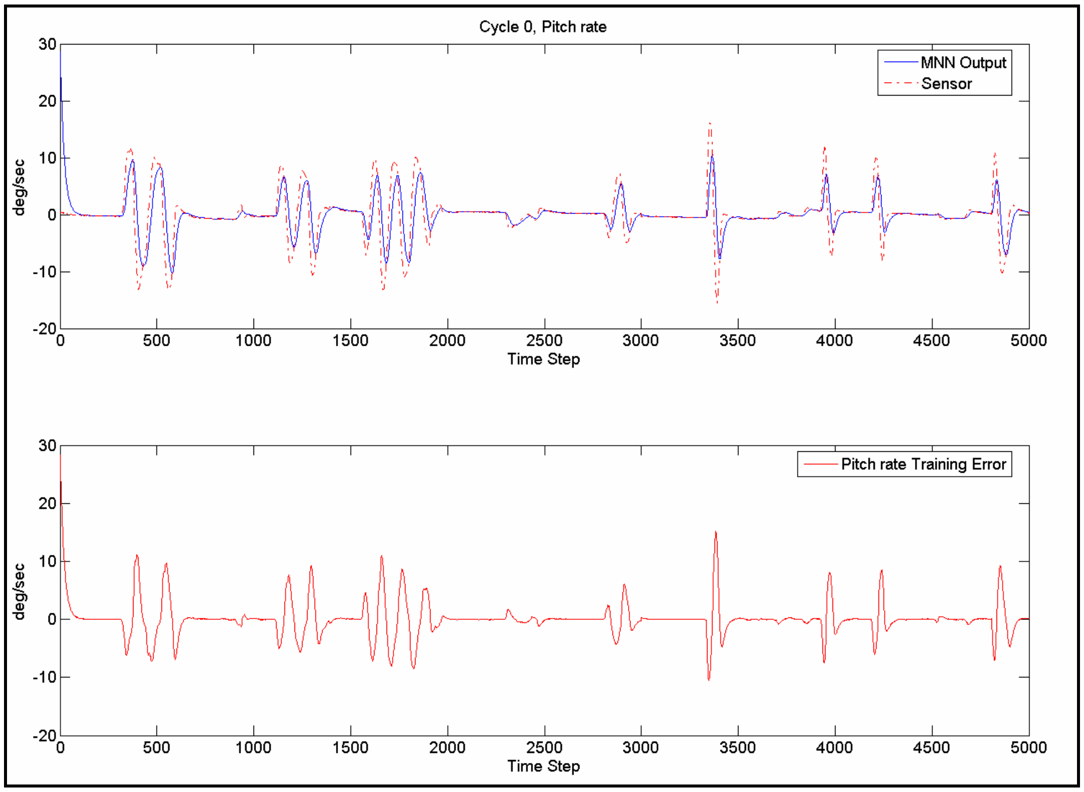


Figure 6-3 : MNN Output, Cycle 0, Pitch Rate

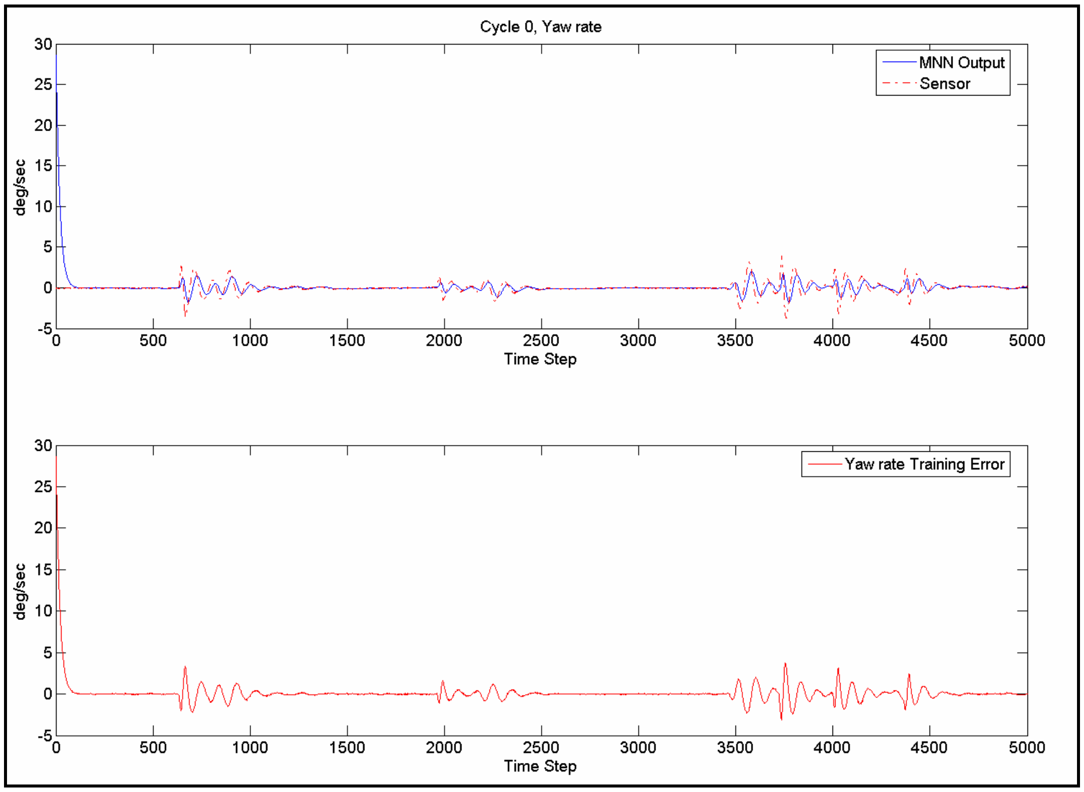


Figure 6-4 : MNN Output, Cycle 0, Yaw Rate

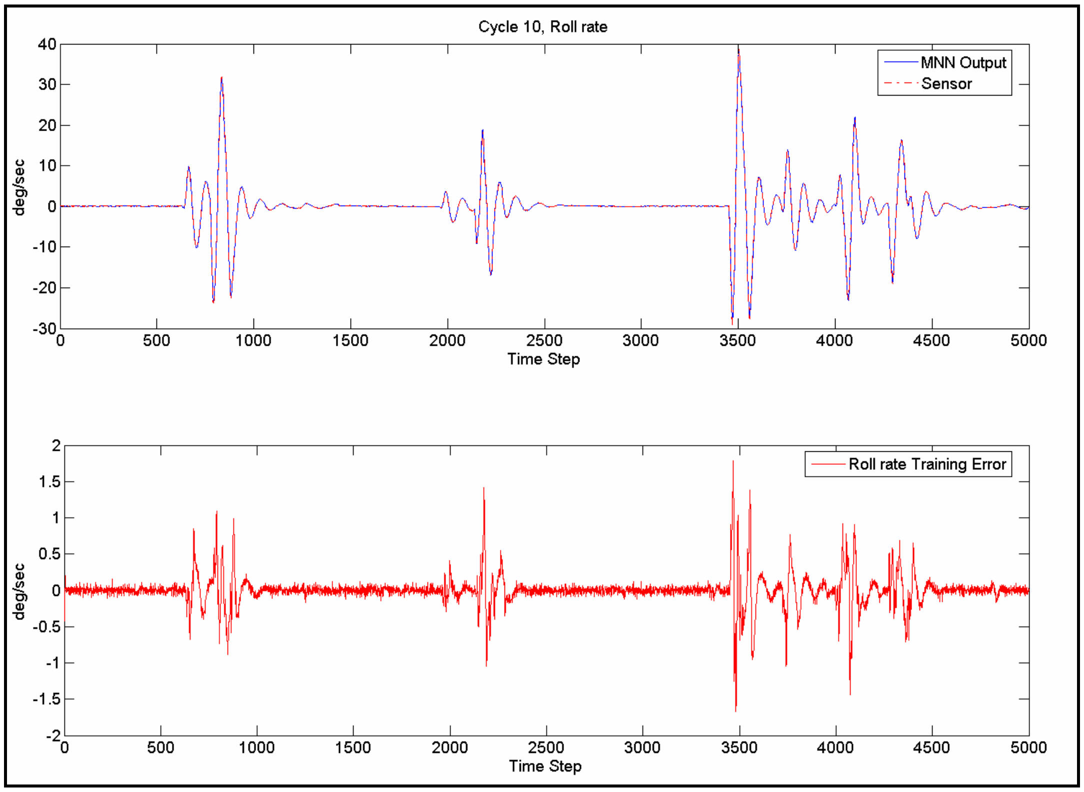


Figure 6-5 : MNN Output, Cycle 10, Roll Rate

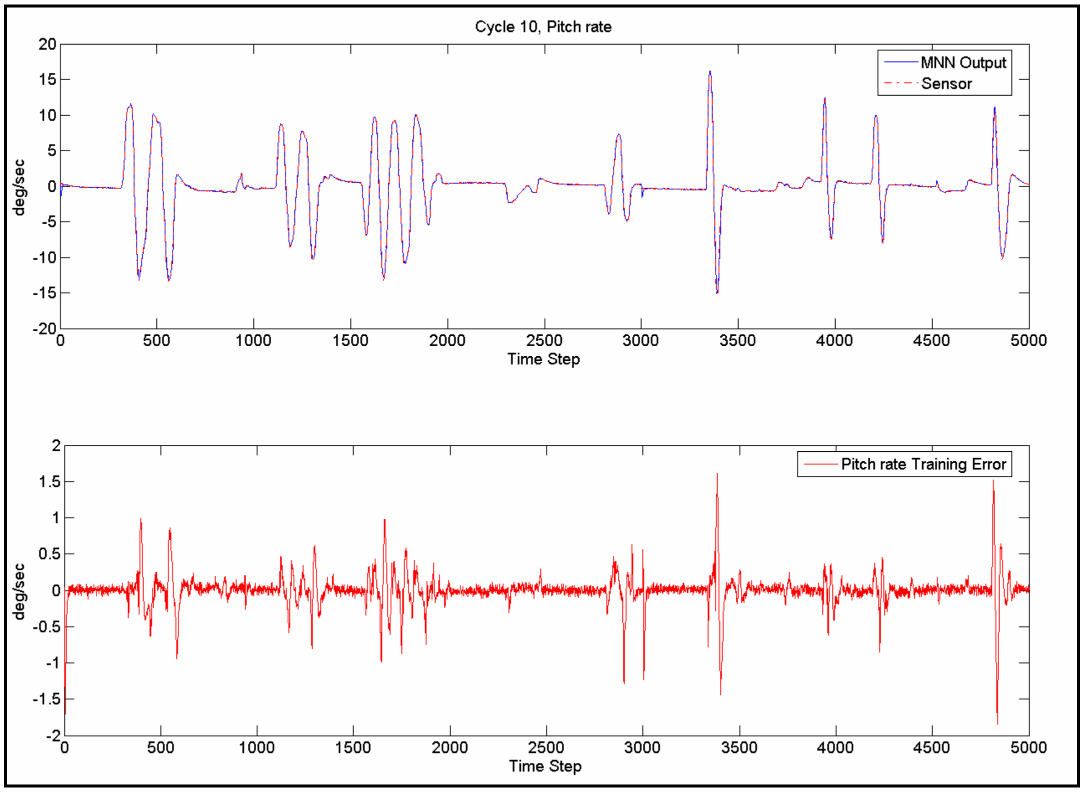


Figure 6-6 : MNN Output, Cycle 10, Pitch Rate

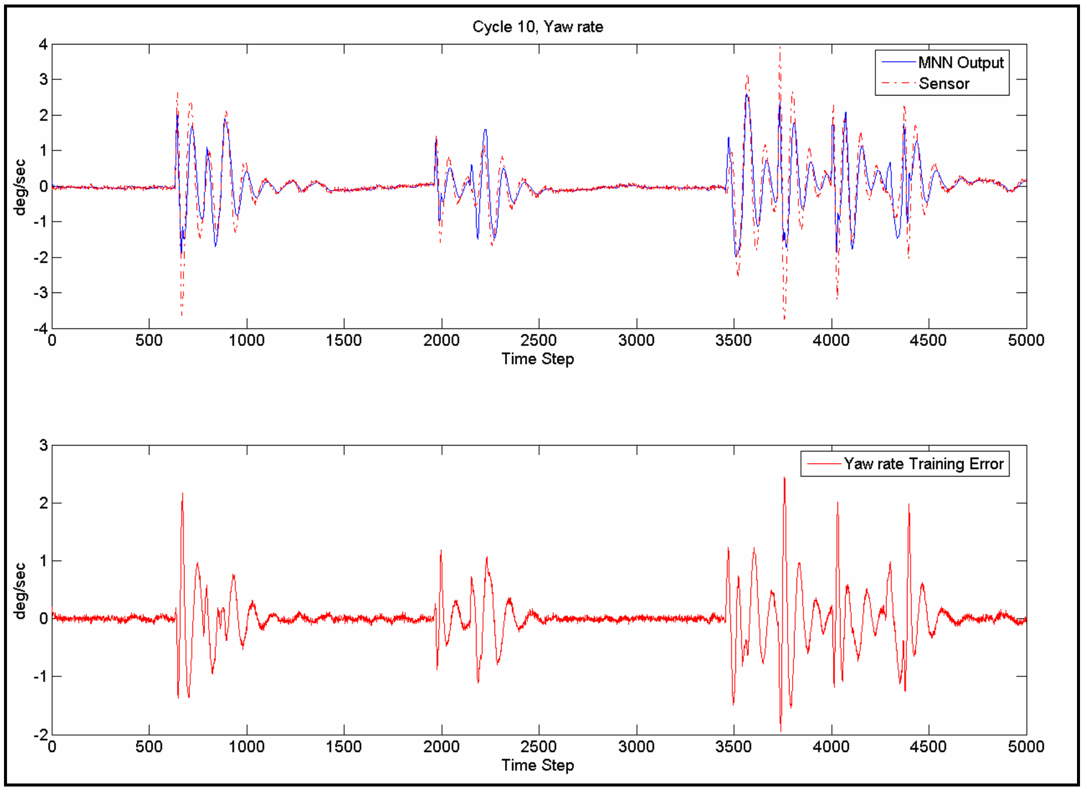


Figure 6-7 : MNN Output, Cycle 10, Yaw Rate

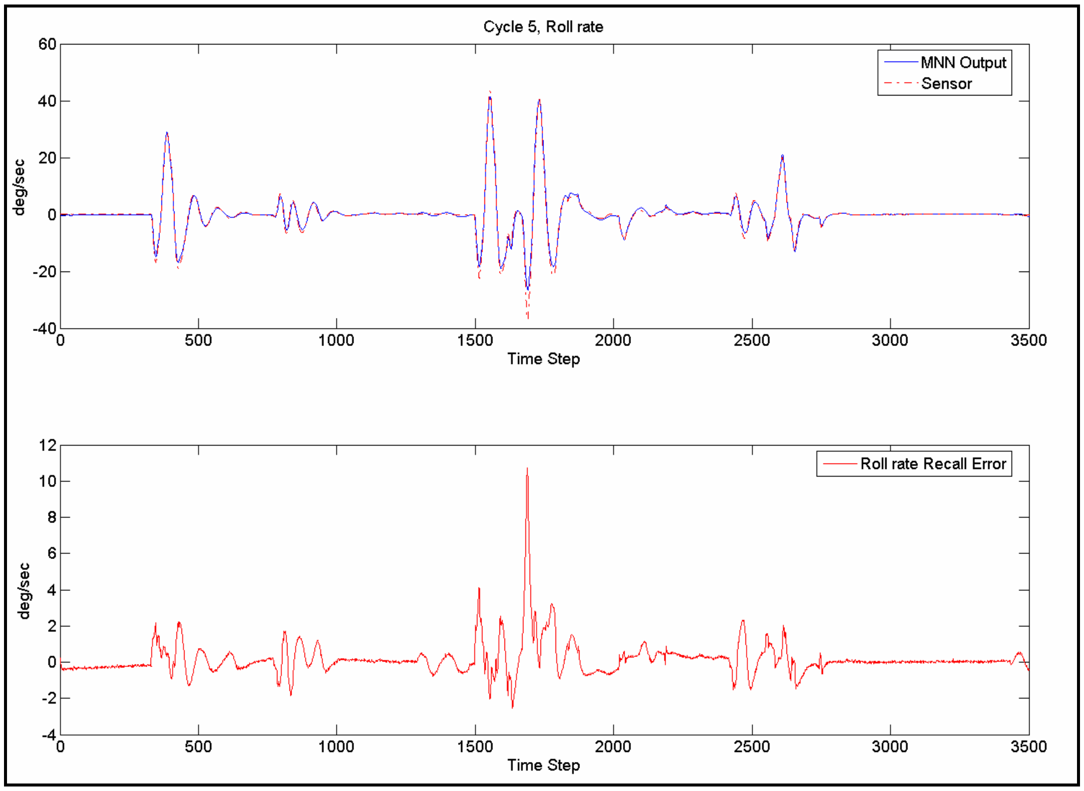


Figure 6-8 : MNN Output, Cycle 5, Roll Rate (Testing)

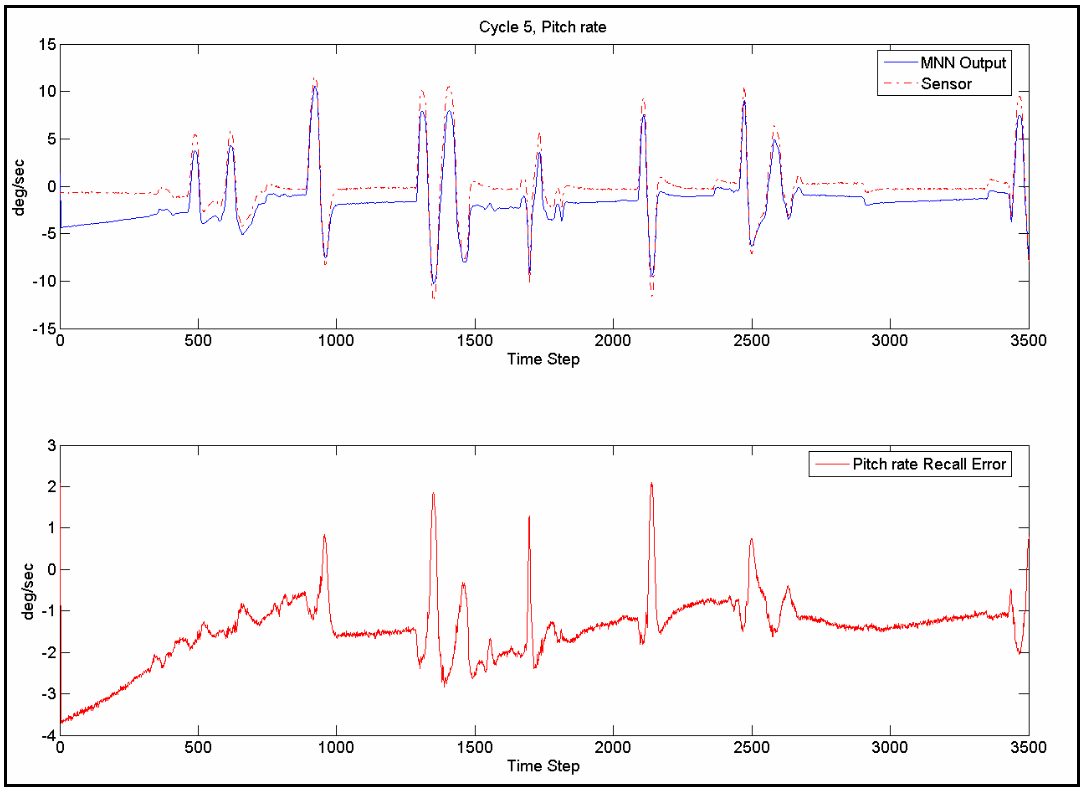


Figure 6-9 : MNN Output, Cycle 5, Pitch Rate (Testing)

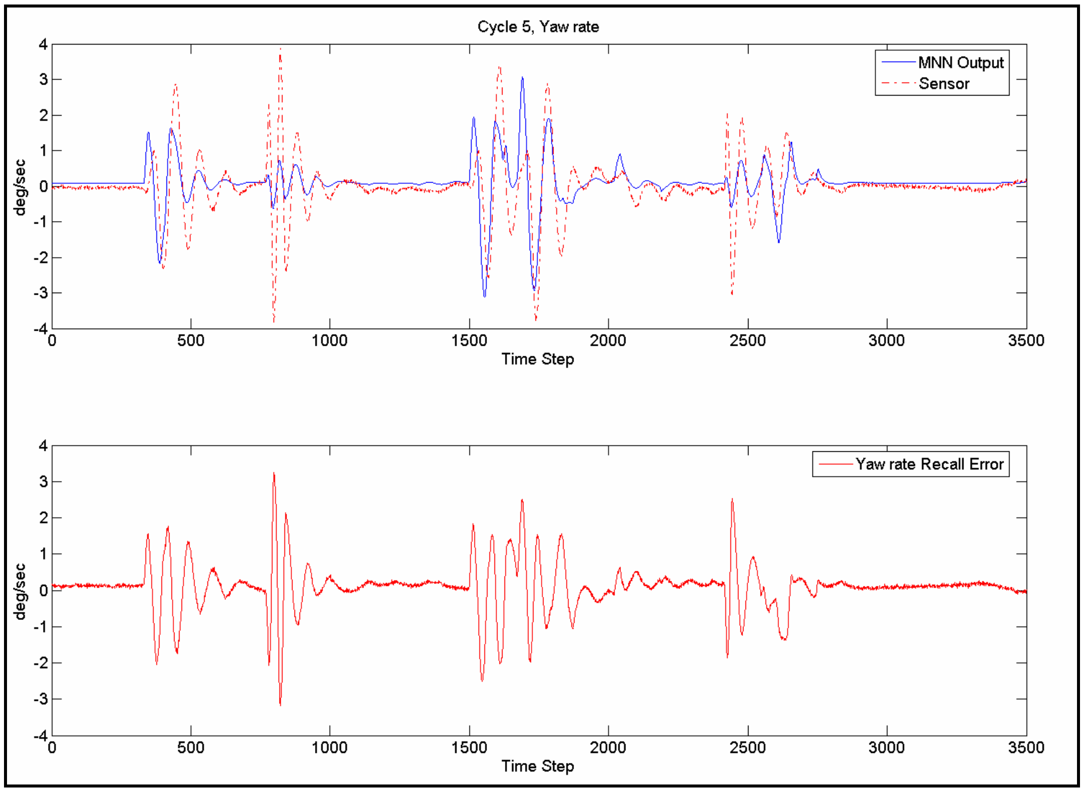


Figure 6-10 : MNN Output, Cycle 5, Yaw Rate (Testing)

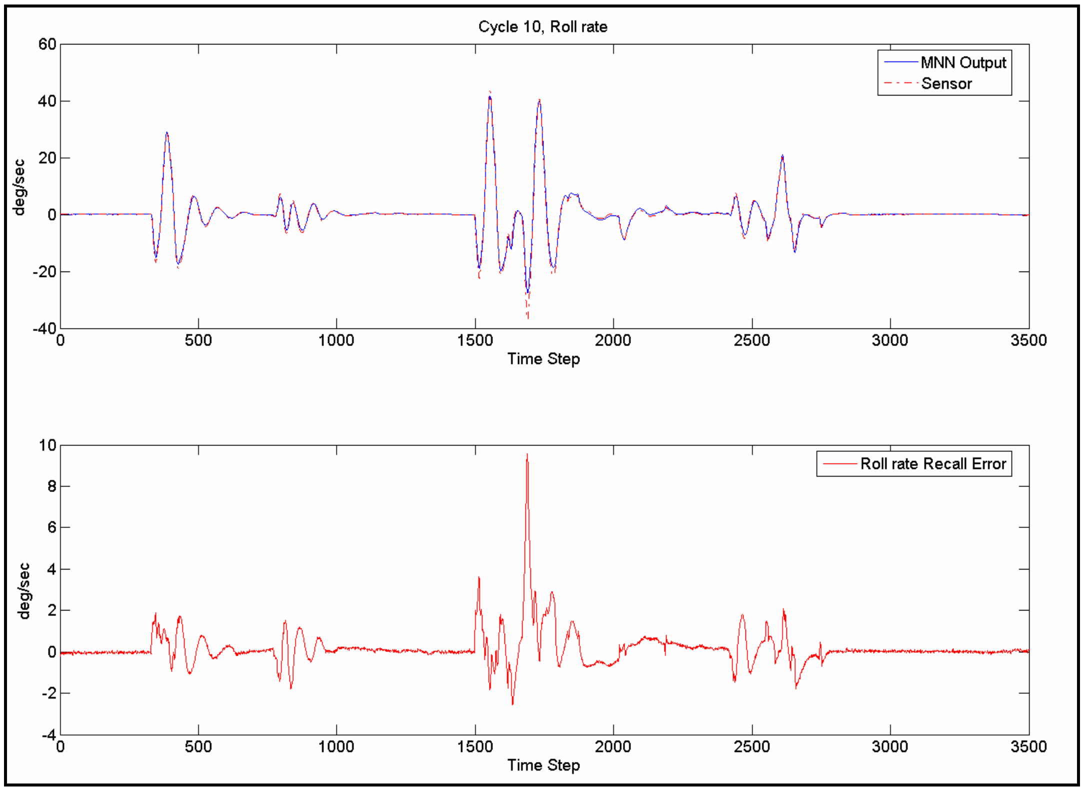


Figure 6-11 : MNN Output, Cycle 10, Roll Rate (Testing)

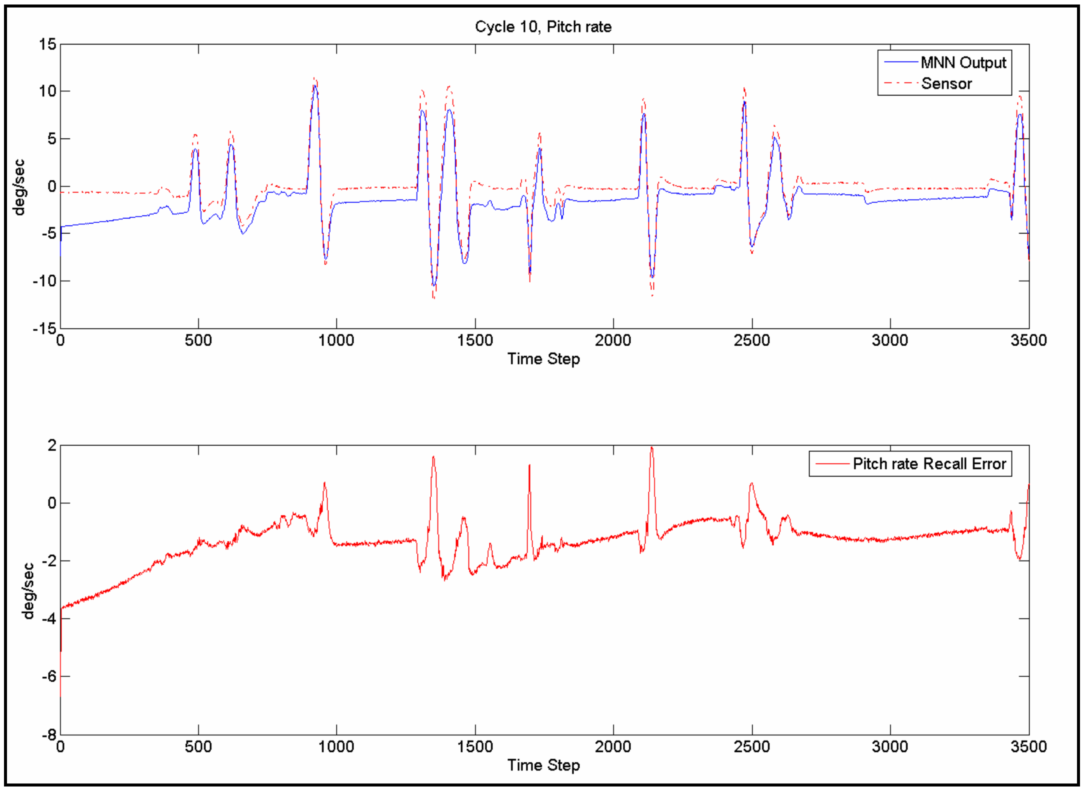


Figure 6-12 : MNN Output, Cycle 10, Pitch Rate (Testing)

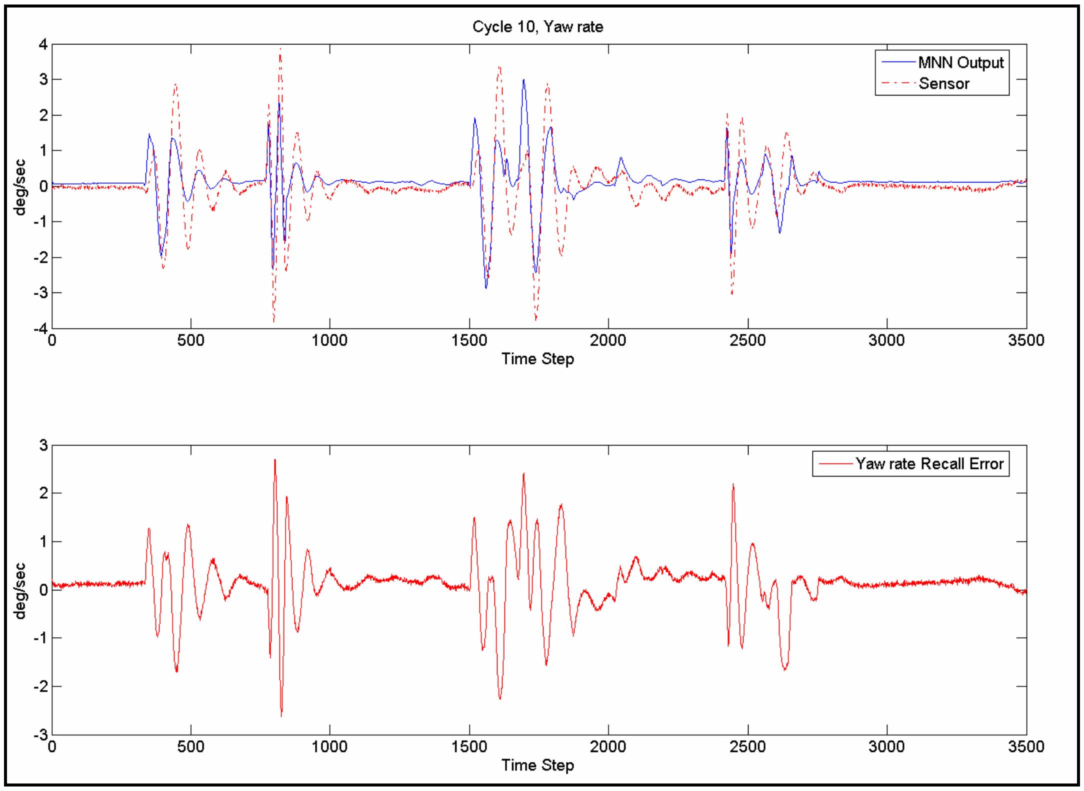


Figure 6-13 : MNN Output, Cycle 10, Yaw Rate (Testing)

The Decentralized Neural networks are trained and tested in a similar fashion to that of the MNN with the same set of training data and testing data. The following figures represent the training and testing cycle of the P-DNN, Q-DNN and the R-DNNs.

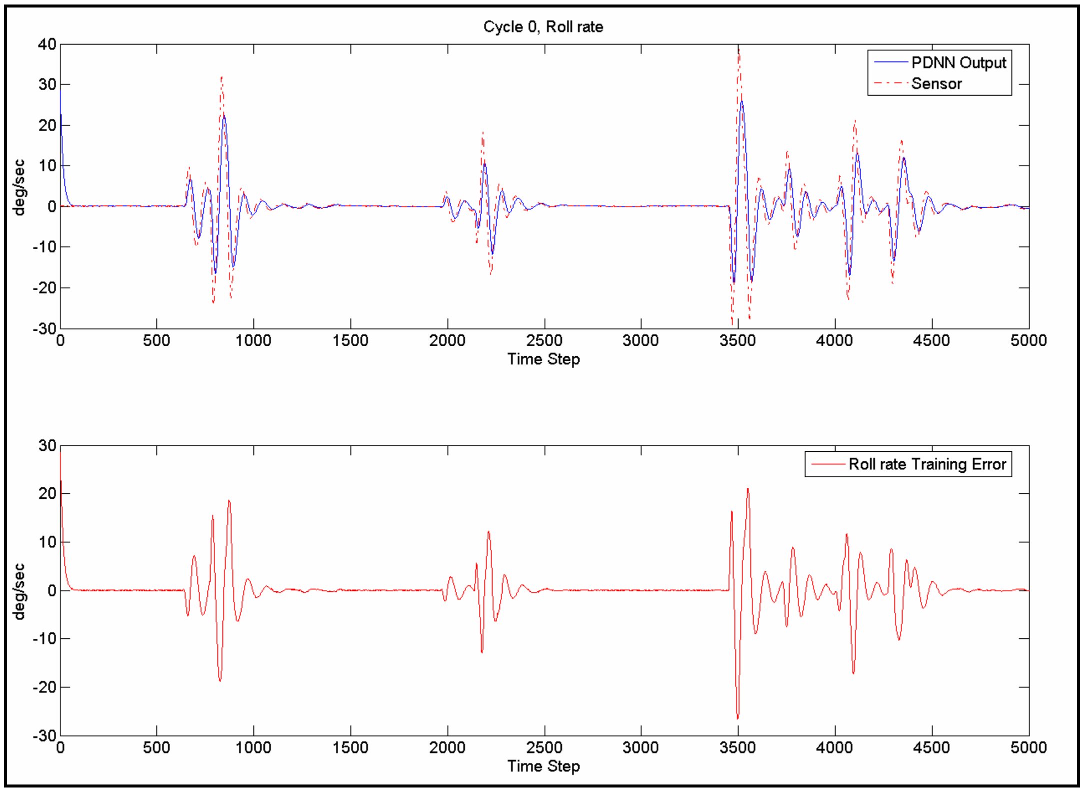


Figure 6-14 : P-DNN Output, Cycle 0

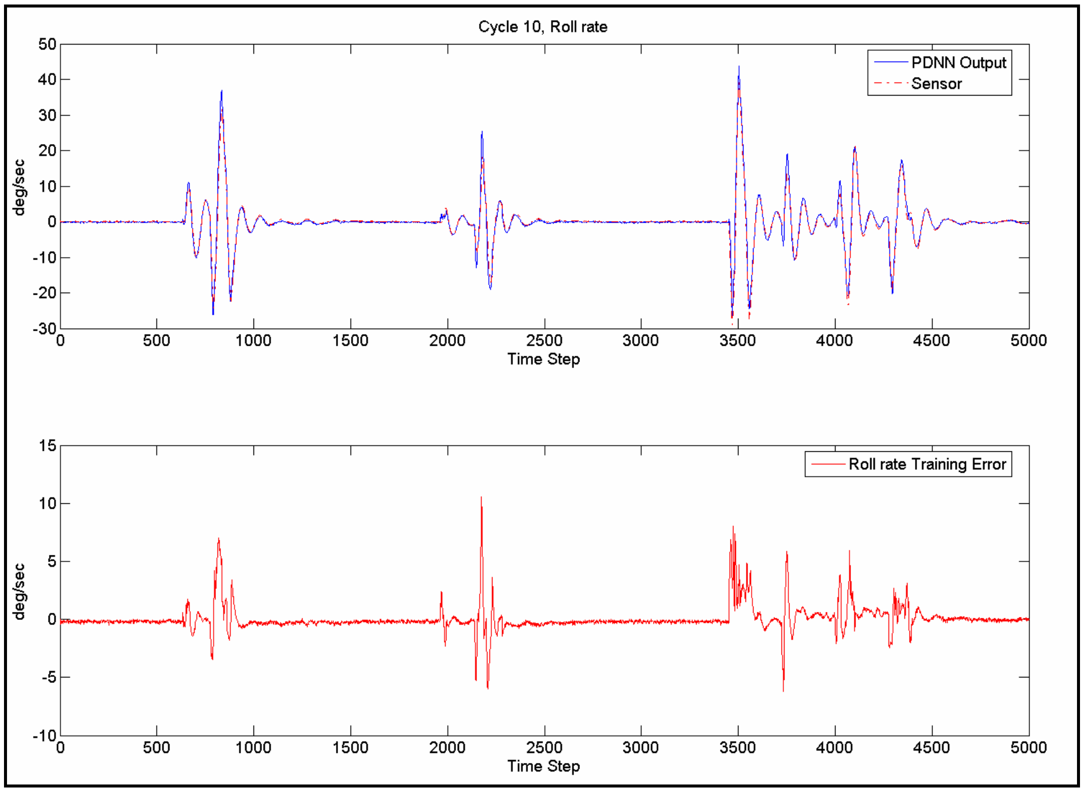


Figure 6-15 : P-DNN Output, Cycle 10

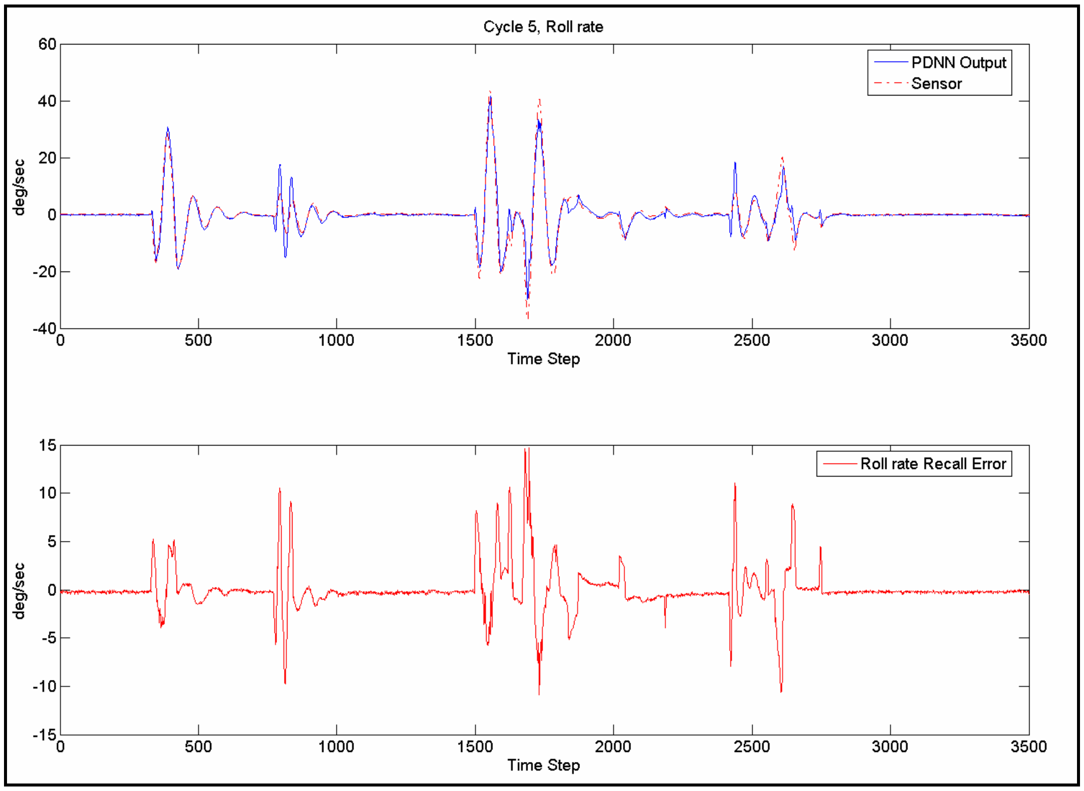


Figure 6-16 : P-DNN Output, Cycle 5 (Testing)

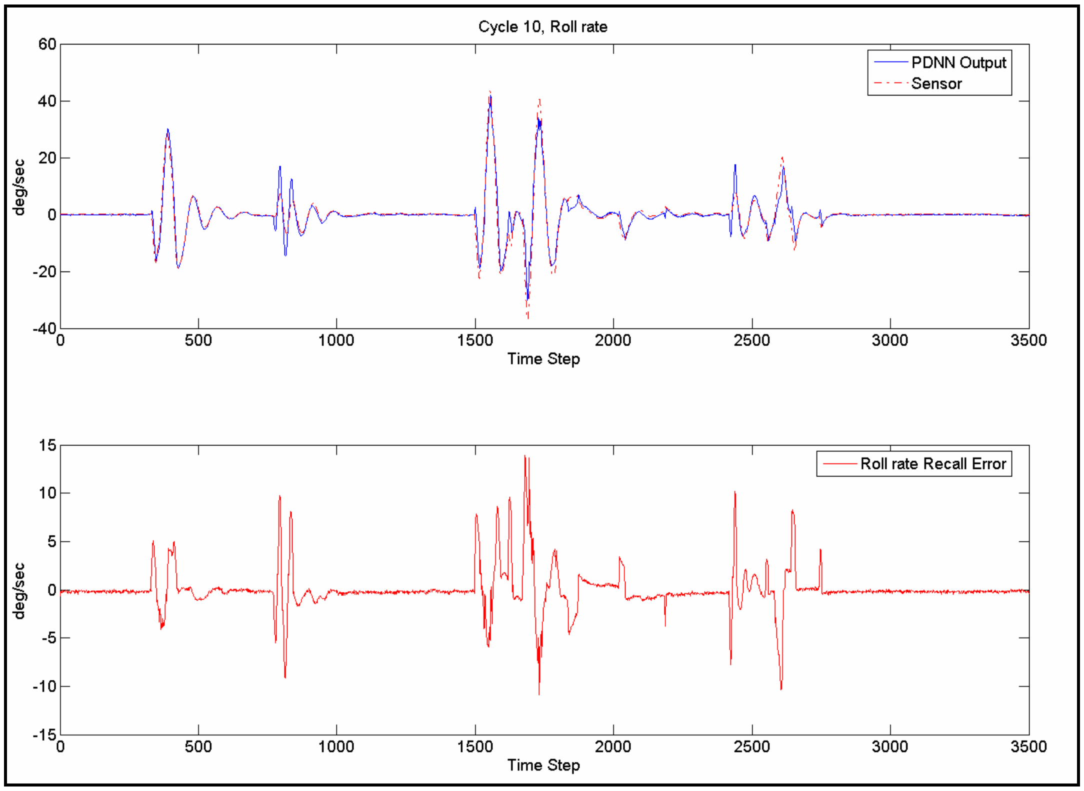


Figure 6-17 : P-DNN Output, Cycle 10 (Testing)

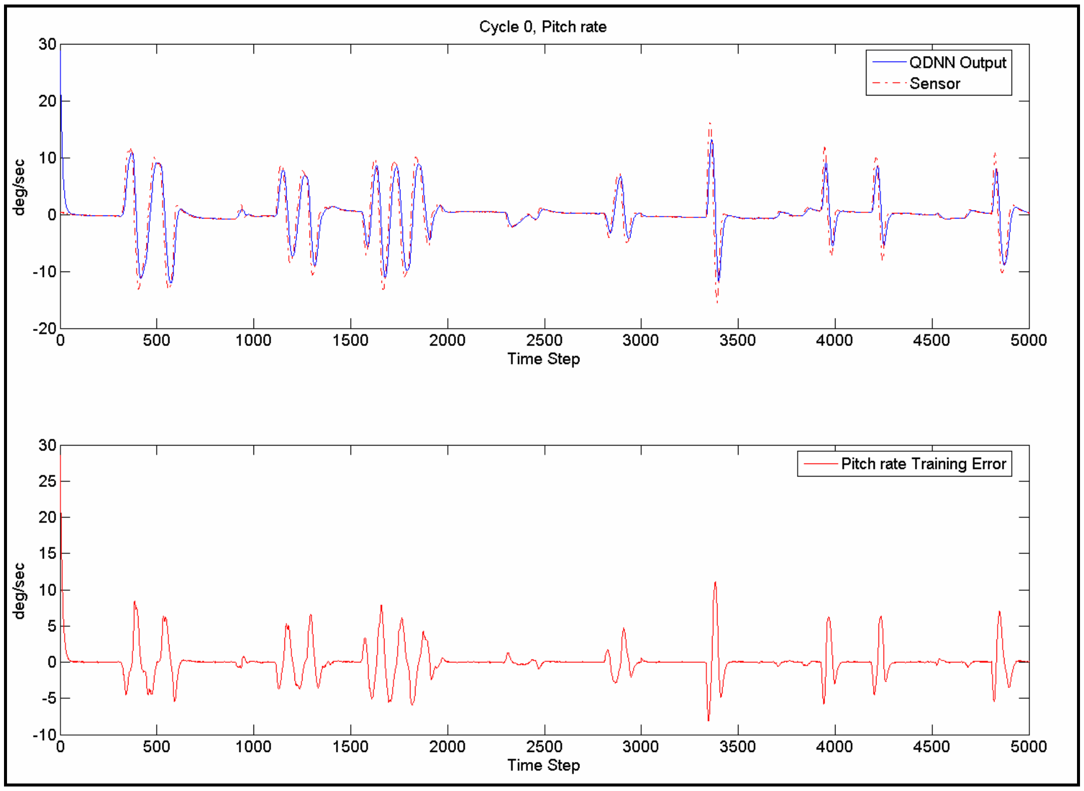


Figure 6-18 : Q-DNN Output, Cycle 0

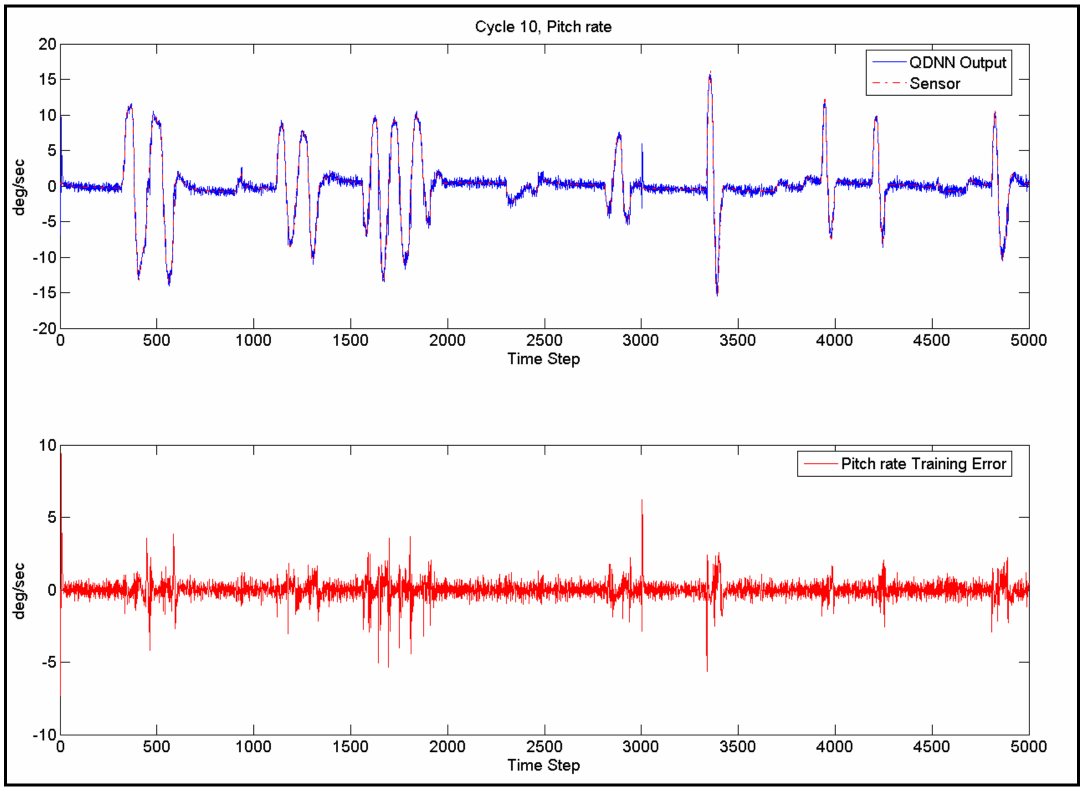


Figure 6-19 : Q-DNN Output, Cycle 10

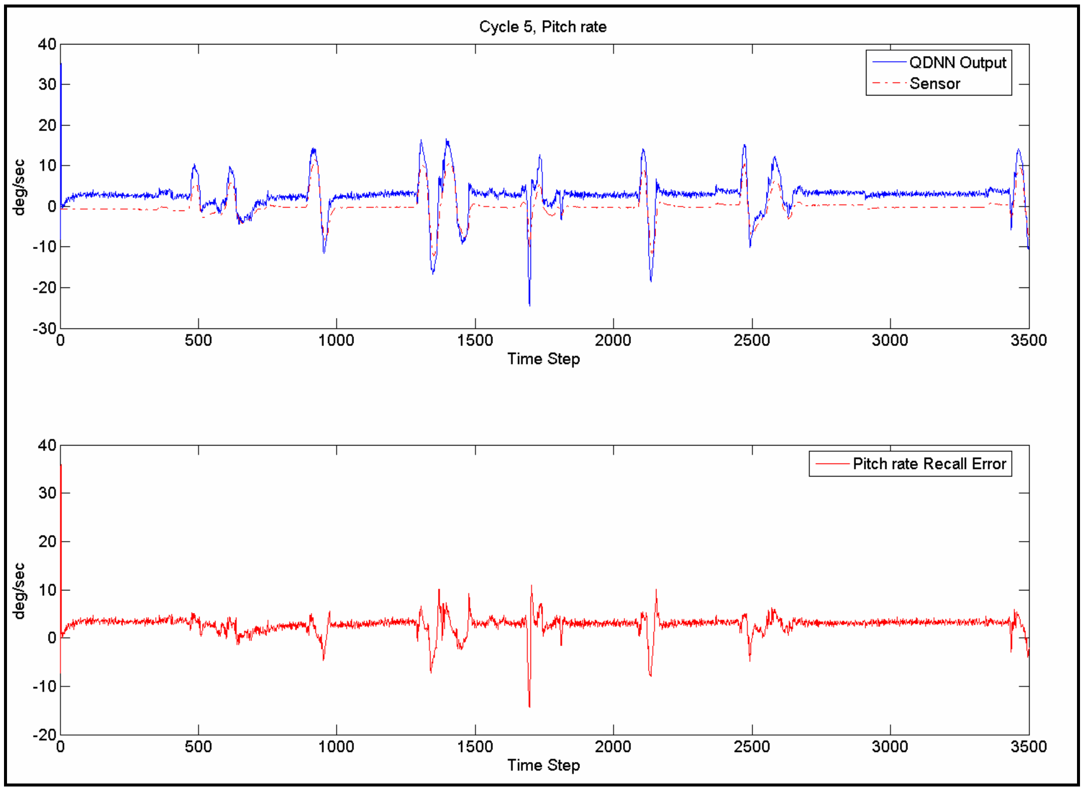


Figure 6-20 : Q-DNN Output, Cycle 5 (Testing)

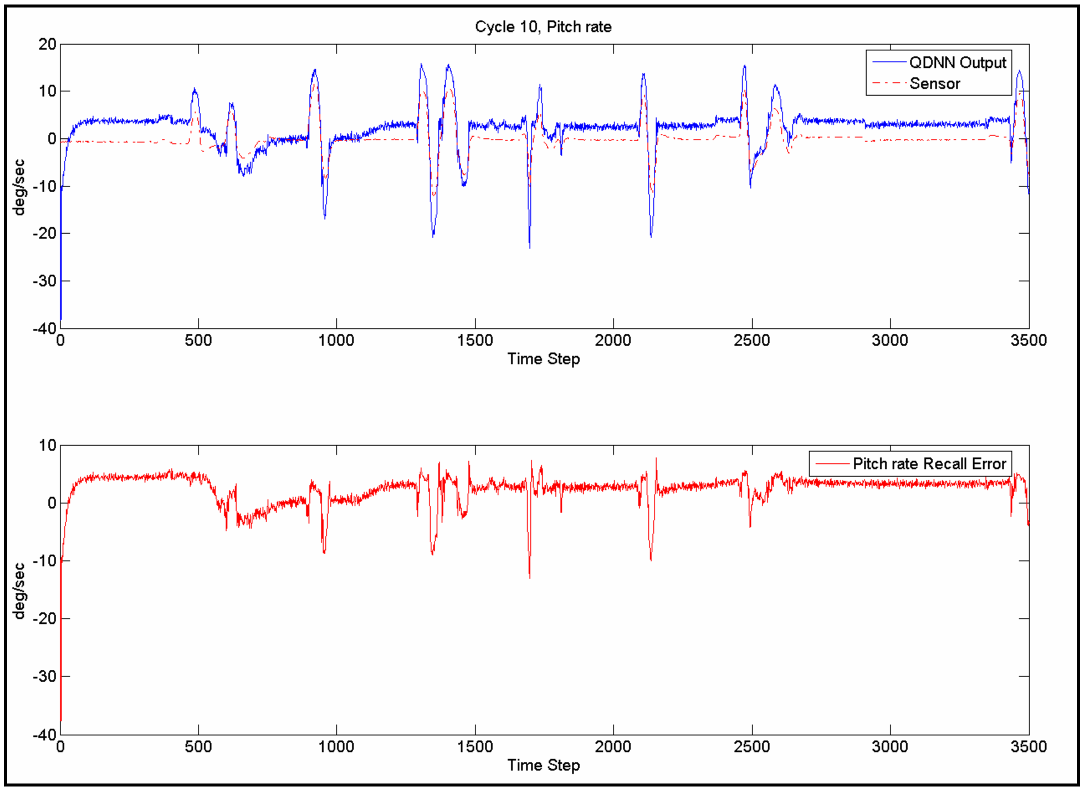


Figure 6-21 : Q-DNN Output, Cycle 10 (Testing)

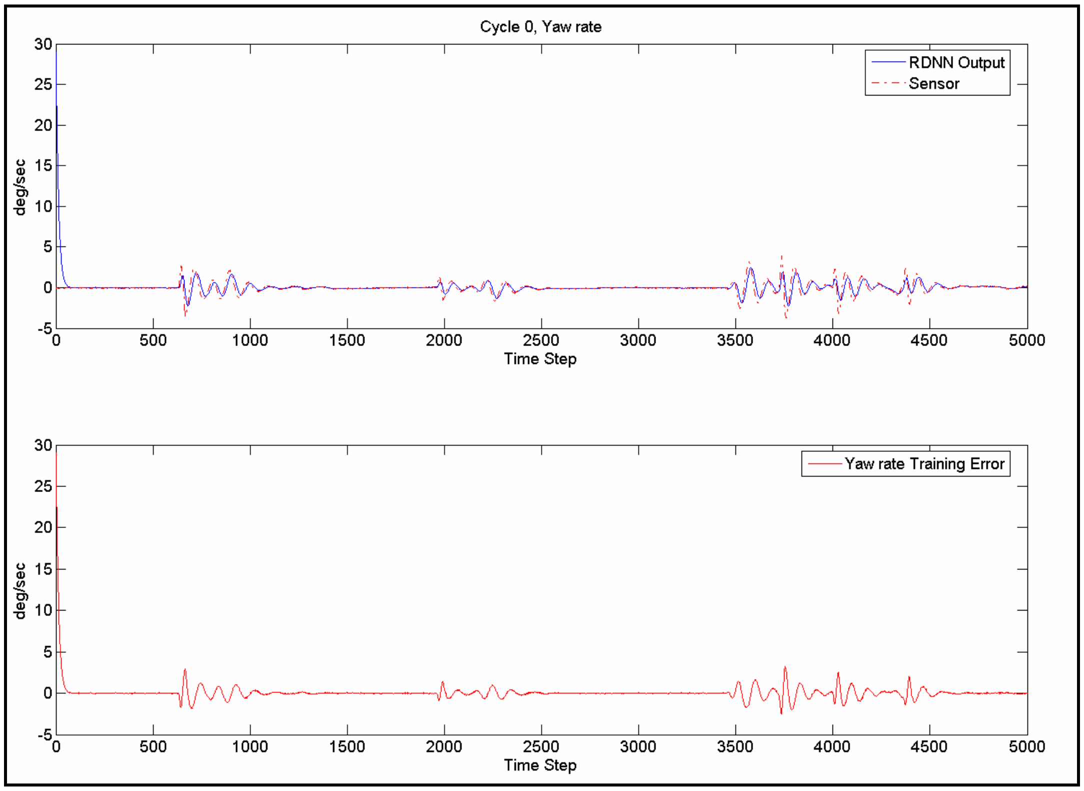


Figure 6-22 : R-DNN Output, Cycle 0

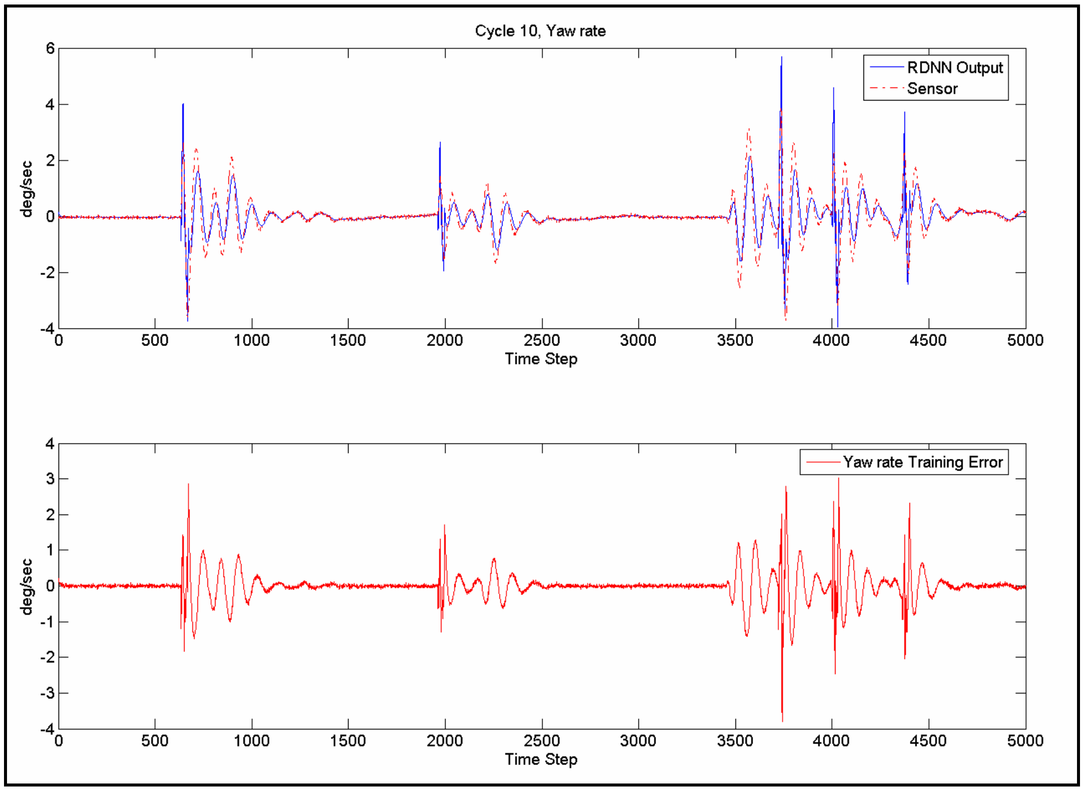


Figure 6-23 : R-DNN Output, Cycle 10

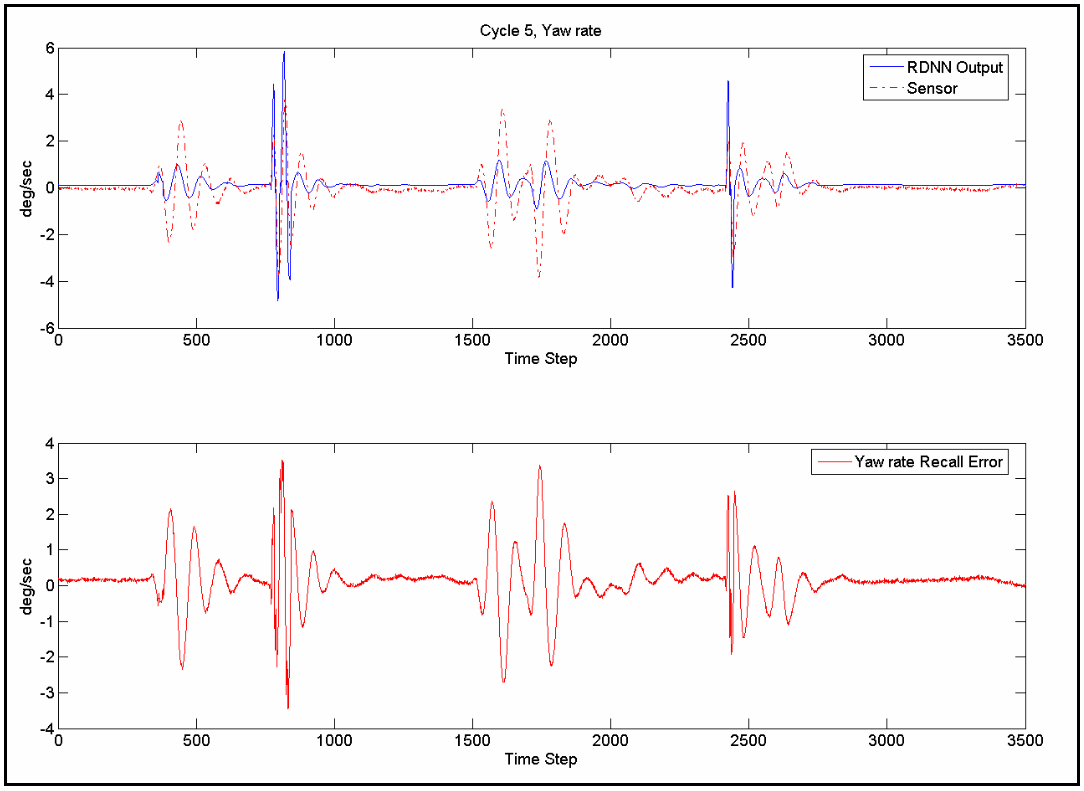


Figure 6-24 : R-DNN Output, Cycle 5 (Testing)

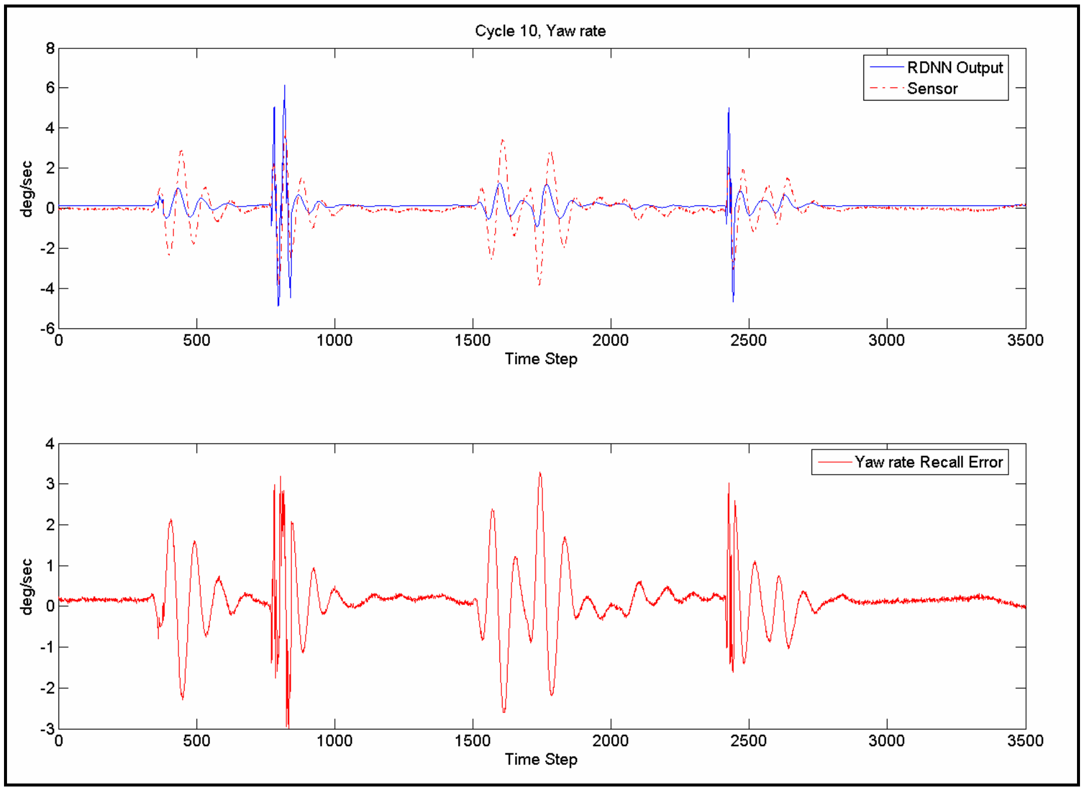


Figure 6-25 : R-DNN Output, Cycle 10 (Testing)

The RMS value of the training error for the MNN outputs and the P, Q and R DNN outputs are presented in the following tables.

Table 6-1 : Mean of Training and Testing Errors on MNN Outputs

Trial 001	Cycle 0		Cycle 5		Cycle 10	
	Training	Testing	Training	Testing	Training	Testing
MNN P Output (deg/s)	0.081970	no data	0.004492	0.013770	0.003696	0.012894
MNN Q Output (deg/s)	0.053757	no data	0.004127	0.024169	0.003774	0.023323
MNN R Output (deg/s)	0.025208	no data	0.008697	0.009485	0.000000	0.008345

Table 6-2 : Mean of Training and Testing Errors on P-DNN Output

Trial 001	Cycle 0		Cycle 5		Cycle 10	
	Training	Testing	Training	Testing	Training	Testing
P-DNN P Output (deg/s)	0.070769	no data	0.007452	0.032378	0.019894	0.031356

Table 6-3 : Mean of Training and Testing Errors on Q-DNN Output

Trial 001	Cycle 0		Cycle 5		Cycle 10	
	Training	Testing	Training	Testing	Training	Testing
Q-DNN Q Output (deg/s)	0.038386	no data	0.11570	0.050728	0.11491	0.043032

Table 6-4 : Mean of Training and Testing Error on R-DNN Output

Trial 001	Cycle 0		Cycle 5		Cycle 10	
	Training	Testing	Training	Testing	Training	Testing
R-DNN R Output (deg/s)	0.021021	no data	0.007625	0.010511	0.007132	0.010215

6.4 SFDIA On-line Simulation and Implementation Results

For the purpose of Sensor Failure Detection, Identification and Accommodation, the MNN, P-DNN, Q-DNN and R-DNN were trained with offline data from the AVDS environment followed by the on-line simulation. The structures of the NNs used in this task are listed in the table. As described earlier, the initial settings for the neural network structures are derived from the offline training process. Using the Matlab GUI, parameters describing the failed gyro, the time of failure and the type of failure were set. Within this study, the failures on the gyros were restricted to step type failures of a small (4 deg/s) and large (8 deg/s) bias amplitude. The following table gives a listing of the structure of the NNs used for this study.

Table 6-5 : Structures of NN for SFDIA

Network	No. of Inputs	TW	ILN	HLN	Outputs	LR	MR
MNN	8	5	40	25	3	0.05	0.00
P-DNN	6	3	18	18	1	0.05	0.005
Q-DNN	4	5	20	20	1	0.1	0.00
R-DNN	5	4	20	25	1	0.075	0.005

Once the parameters of the SFDIA procedure were set using the Matlab GUI, they were sent across the socket communication channel to the main node on the Beowulf cluster. When the main node receives the simulation parameters data, it distributes the same to all the compute nodes on the cluster. As described in section 5.3, once this process is complete, the main node waits for the initialization string from AVDS. Once initialization string from AVDS is received, the main node sends the task identification number and the total time of simulation across to AVDS and then waits for the “pilot” to fly the aircraft. Within AVDS, once the task identification number and the total time of simulation are received, the “pilot” takes off and climbs to an altitude over 3000 ft. once the aircraft breaches the 3000 ft barrier, AVDS starts its one-way communication with the main node and sends sensor data at a rate of 30 Hz. Within the main node, once the data is received from AVDS, it is distributed across the compute nodes to generate the estimates of the *roll*, *pitch* and *yaw* rates. These estimates are then sent back to the main node where the error metrics are computed and stored. Once the time of failure is reached, these error metrics are then used to check for failures on roll rate gyro, pitch rate gyro and the yaw rate gyros. If a failure is detected on any of the gyros, then a flag indicating such an event is set and sent as a part of the data stream to the DNNs at the next time step, indicating that a failure had occurred at the previous step. Within the DNNs, a code segment checks for the status of these specific flags to determine if the MNN had declared a failure on their respective gyro, at the previous time step. If the flag is set to indicate a failure, then the DNN suspends it learning process and begins the recall process. Thus from this time forward, the output from the DNN representing the failed gyro is used as an estimate of the actual sensor measurement. This trend can be clearly seen in the plots of the MNN output for the channel representing the failed gyro.

Figures 6-26 to 6-31 show the results from the SFDIA procedure, with a step failure of 8 deg/sec bias occurring on the Q-gyro occurring at 50 sec into the flight. The error threshold on the MNN is exceeded at 50.066s, indicating a failure on the sensor subsystems. Subsequently, the error on the Q-DNN exceeds its thresholds at 50.066s, identifying the failure.

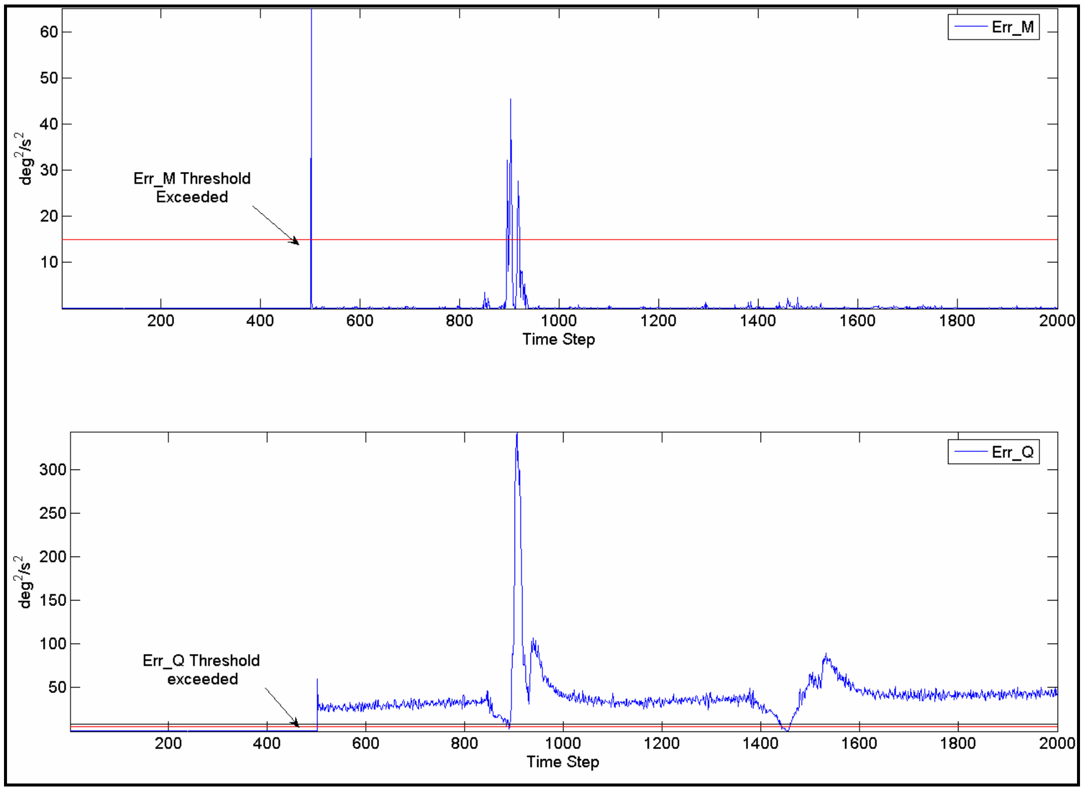


Figure 6-26 : Error Thresholds for Failure on Q (8 deg/s Bias)

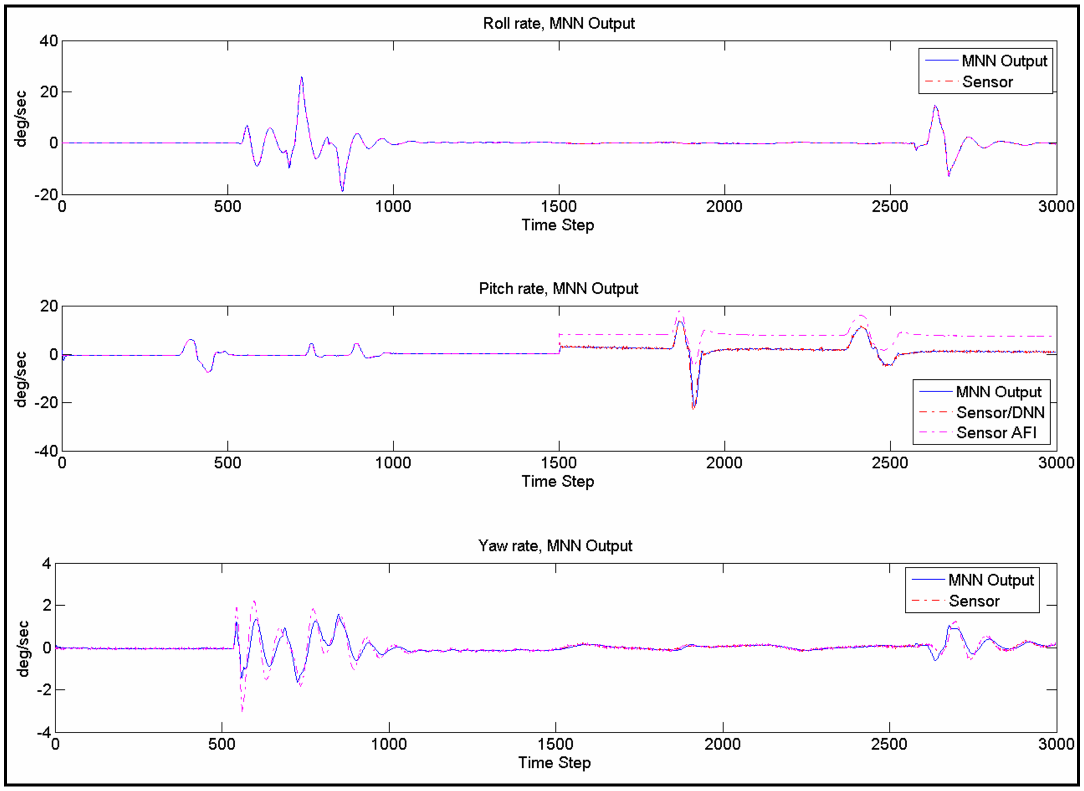


Figure 6-27 : MNN Outputs for Failure on Q (8 deg/s Bias)

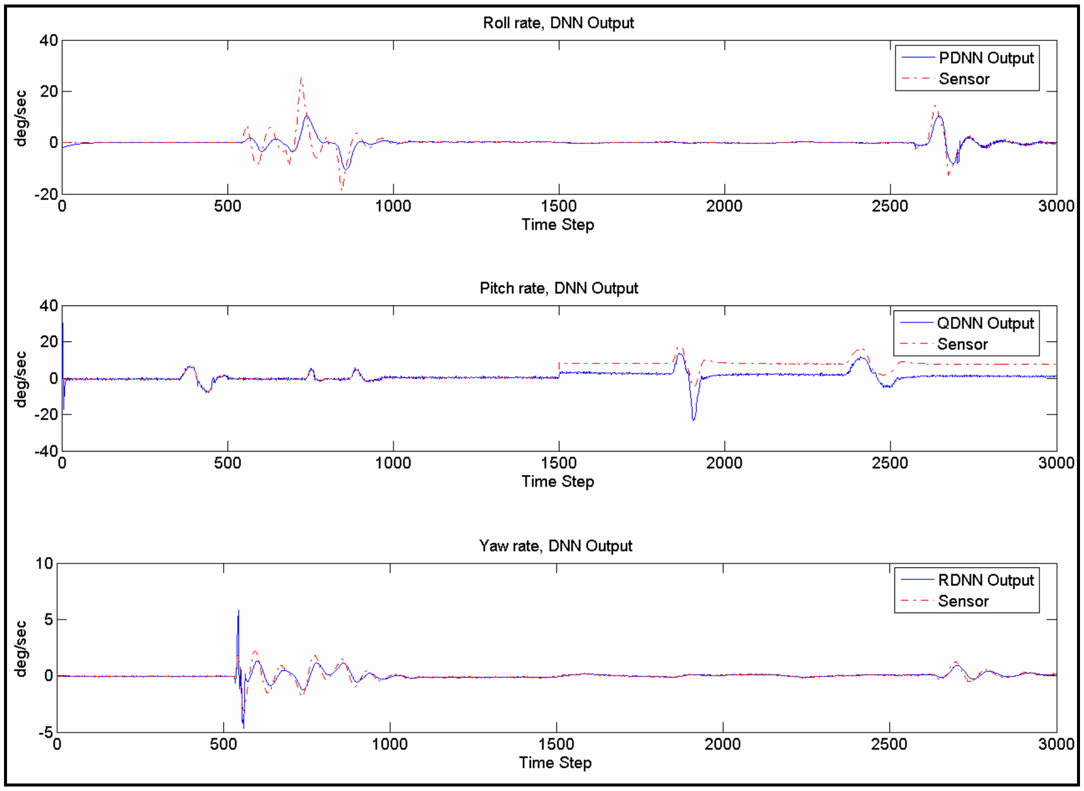


Figure 6-28 : DNN Outputs for Failure on Q (8 deg/s Bias)

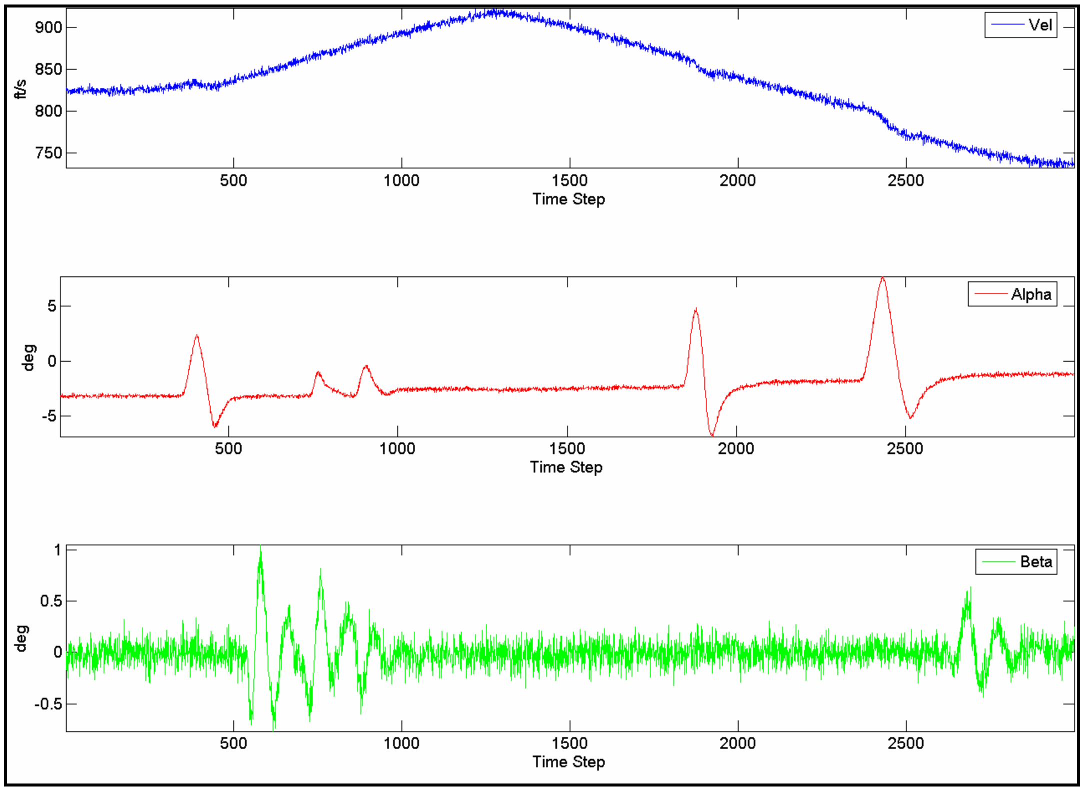


Figure 6-29 : Velocity, Alpha, and Beta

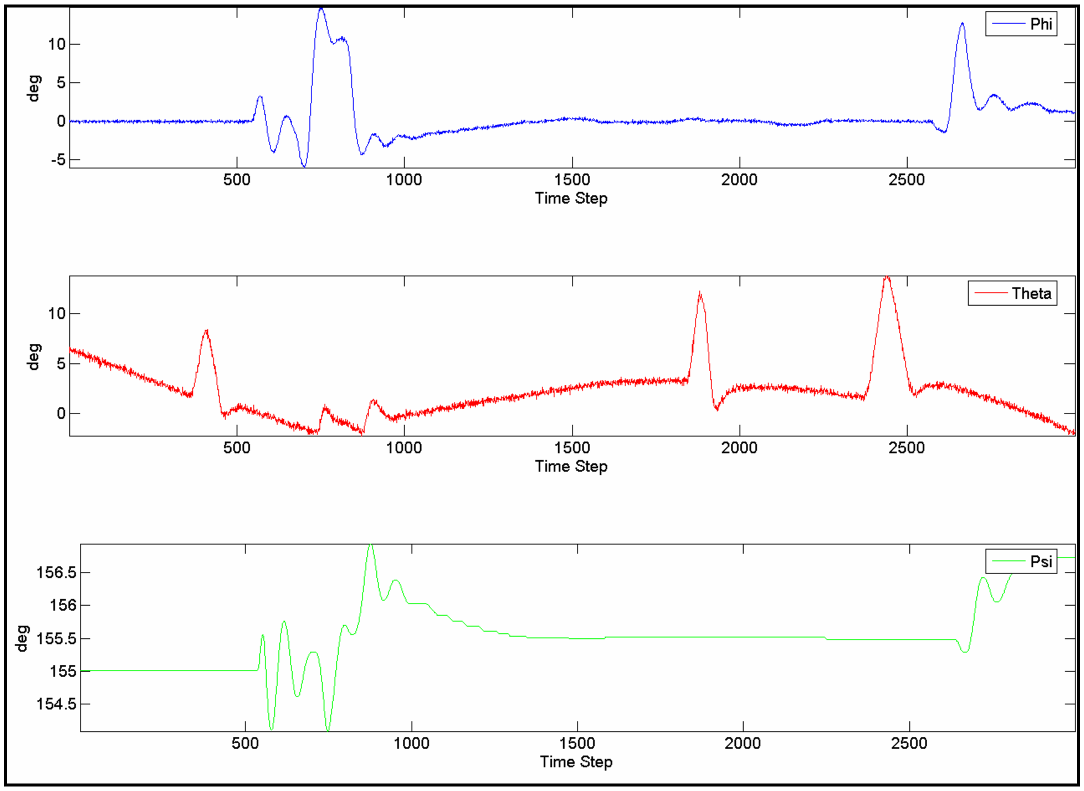


Figure 6-30 : Euler Angles

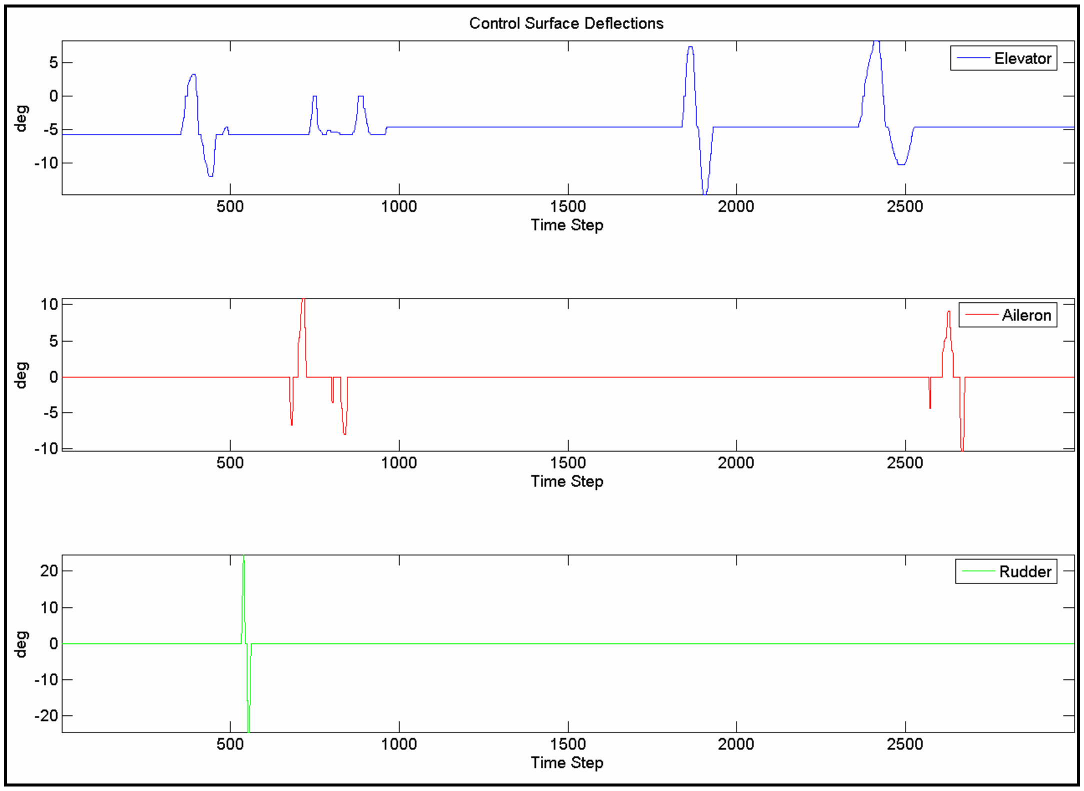


Figure 6-31 : Control Surface Deflections

Figures 6-32 to 6-37 show the results from the SFDIA procedure, with a step failure of 8 deg/sec bias occurring on the R-gyro occurring at 50 sec into the flight. The error threshold on the MNN is exceeded at 50.066s, indicating a failure on the sensor subsystems. Subsequently, the error on the R-DNN exceeds its thresholds at 50.066s, identifying the failure.

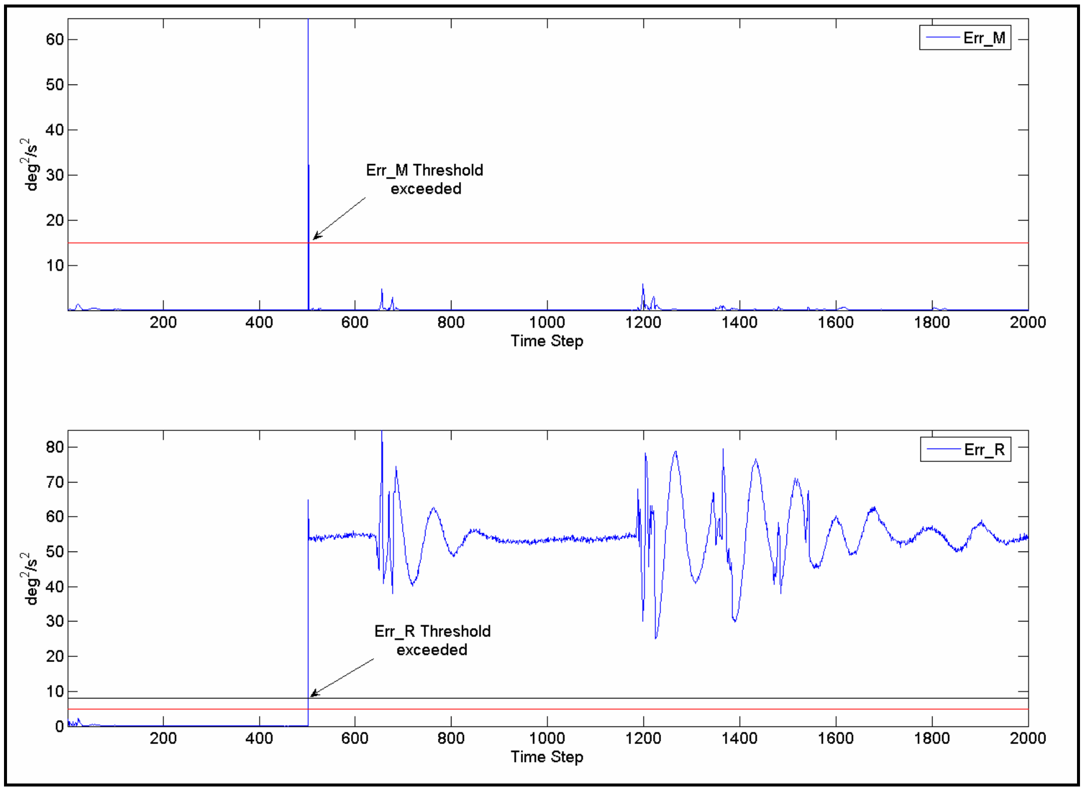


Figure 6-32 : Error Thresholds for Failure on R (8 deg/s Bias)

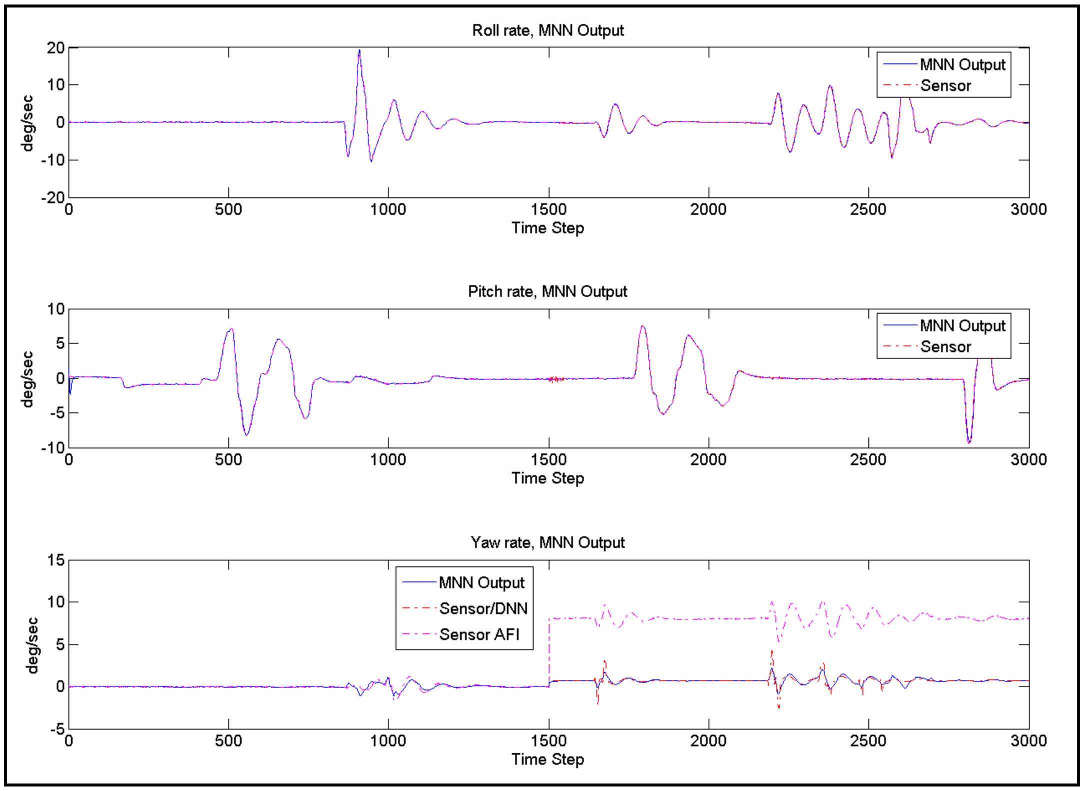


Figure 6-33 : MNN Outputs for Failure on R (8 deg/s Bias)

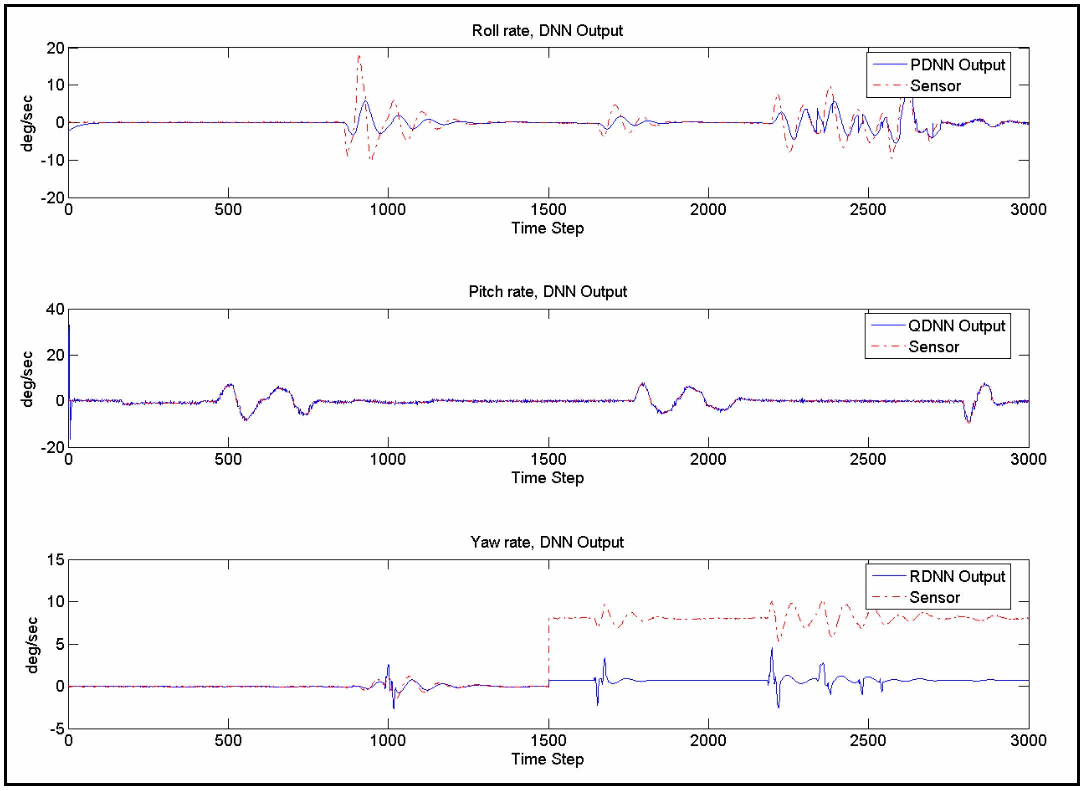


Figure 6-34 : DNN Outputs for Failure on R (8 deg/s Bias)

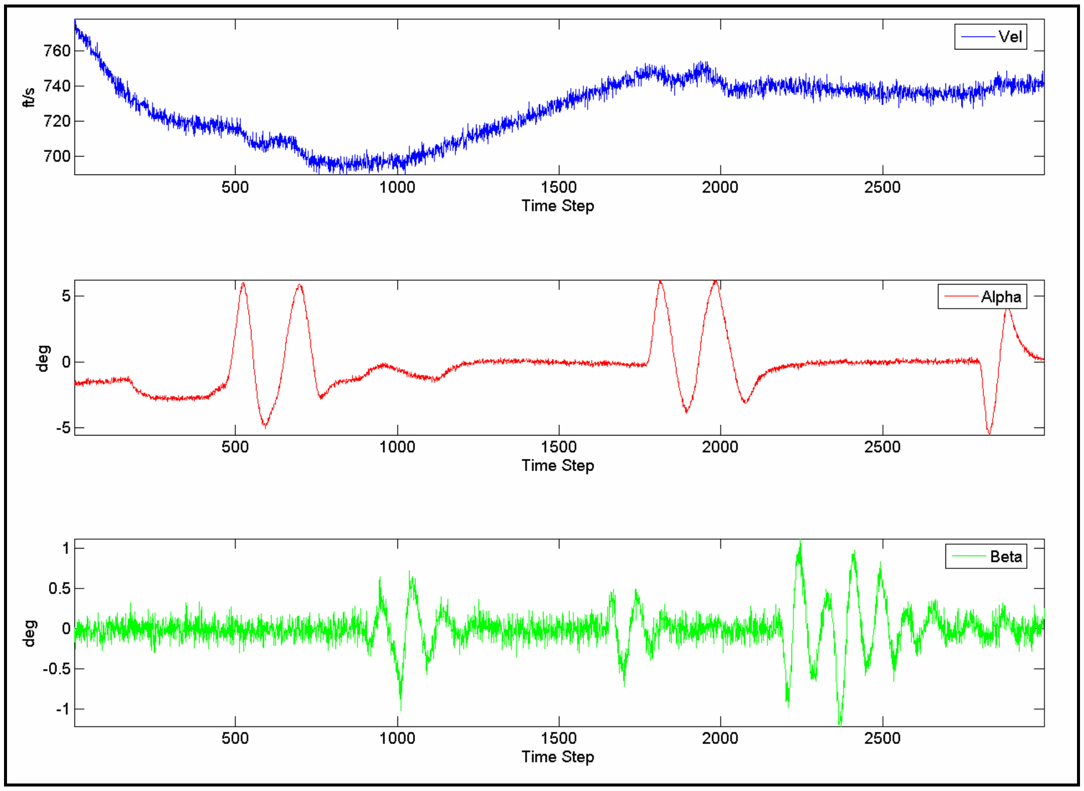


Figure 6-35 : Velocity, Alpha and Beta

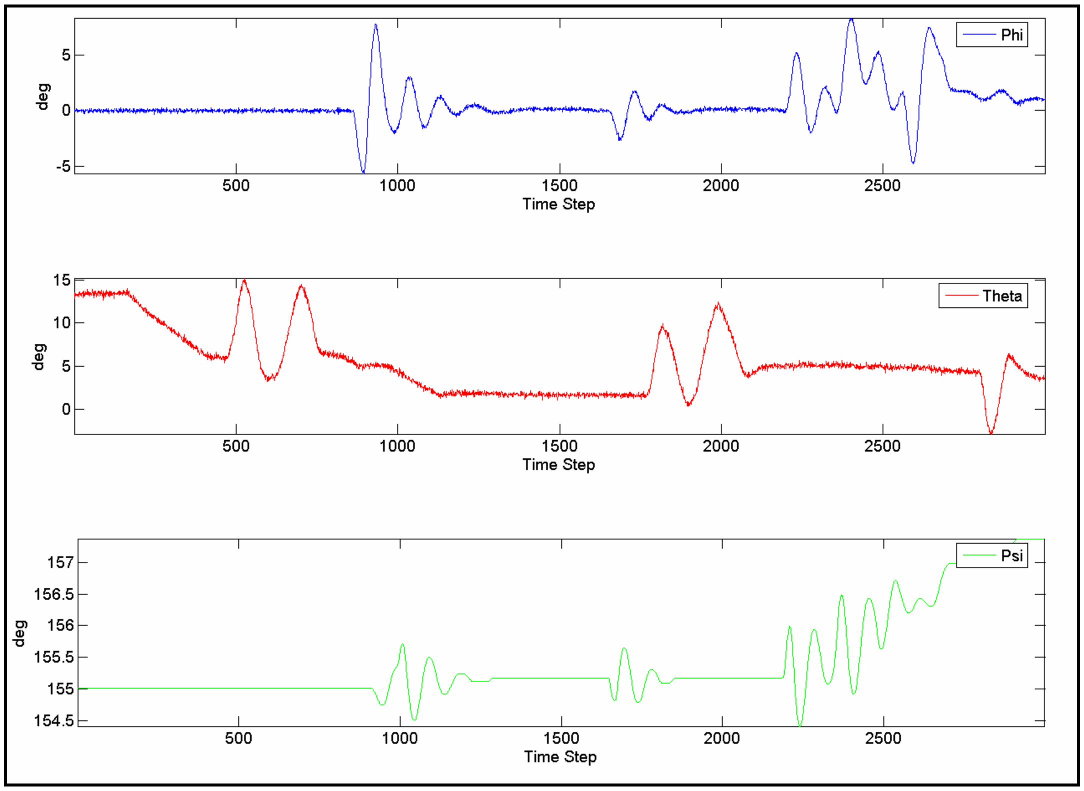


Figure 6-36 : Euler Angles

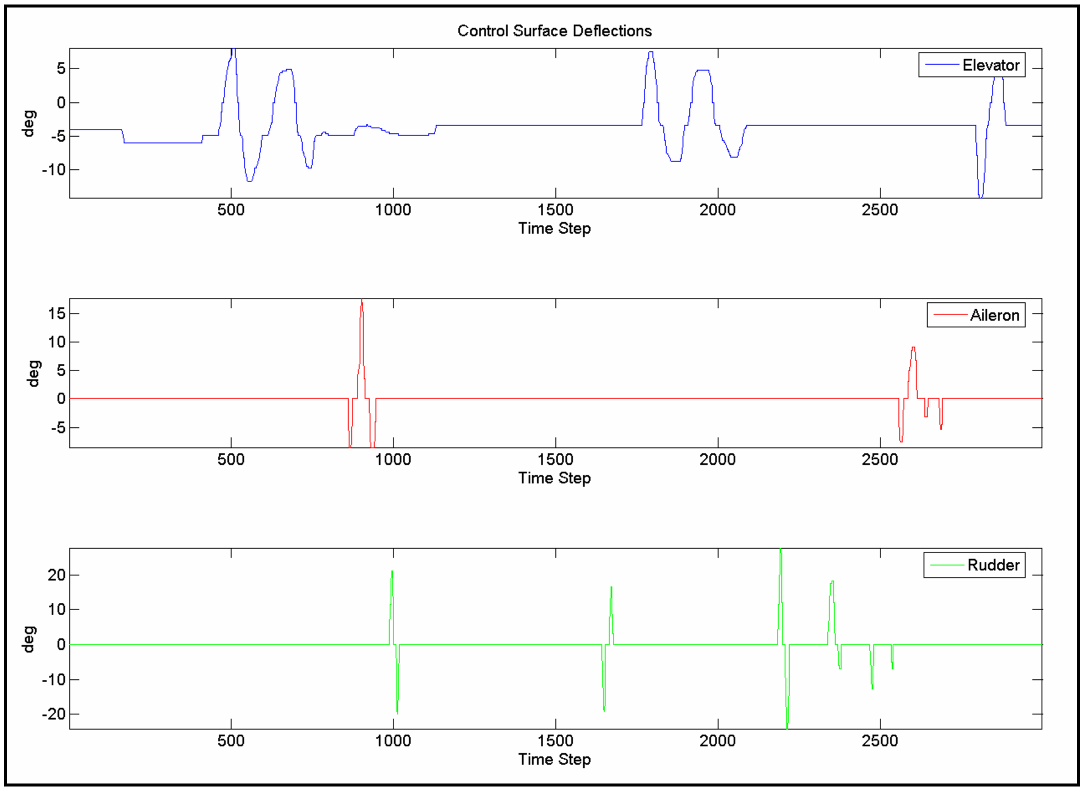


Figure 6-37 : Control Surface Deflections

Figures 6-38 to 6-43 show the results from the SFDIA procedure, with a step failure of 8 deg/sec bias occurring on the P-gyro occurring at 50 sec into the flight. The error threshold on the MNN is exceeded at 50.066s, indicating a failure on the sensor subsystems. Subsequently, the error on the P-DNN exceeds its thresholds at 50.066s, identifying the failure.

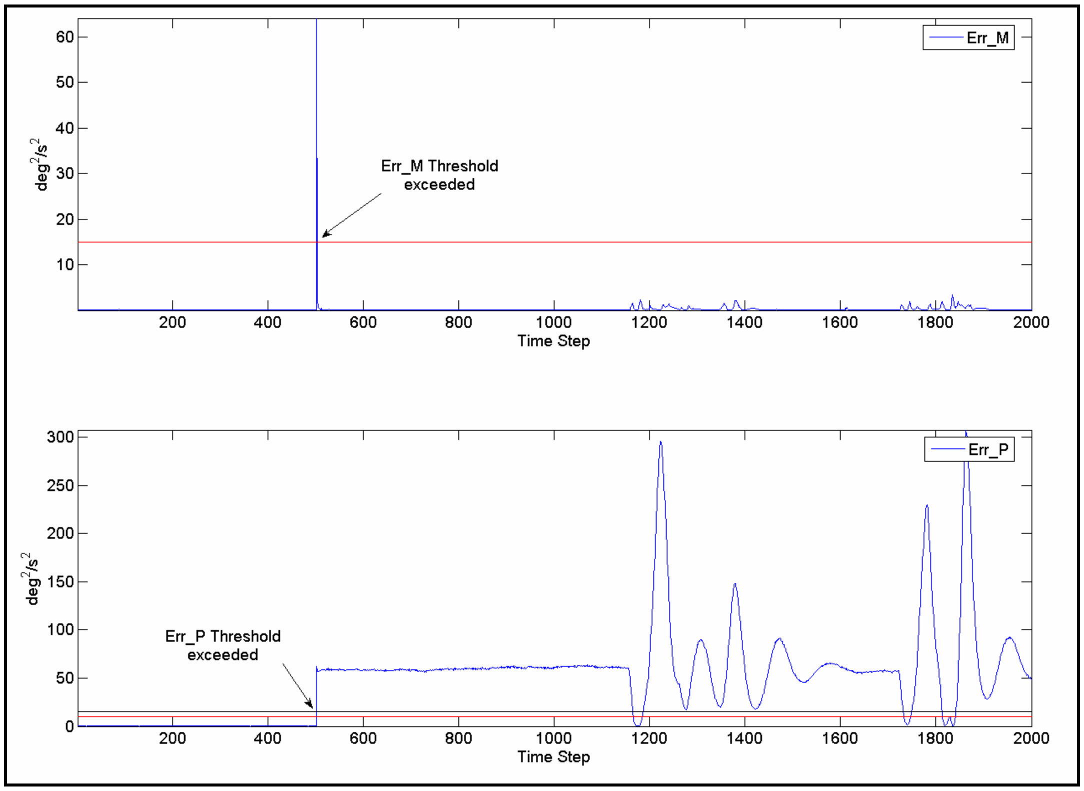


Figure 6-38 : Error Thresholds for Failure on P (8 deg/s Bias)

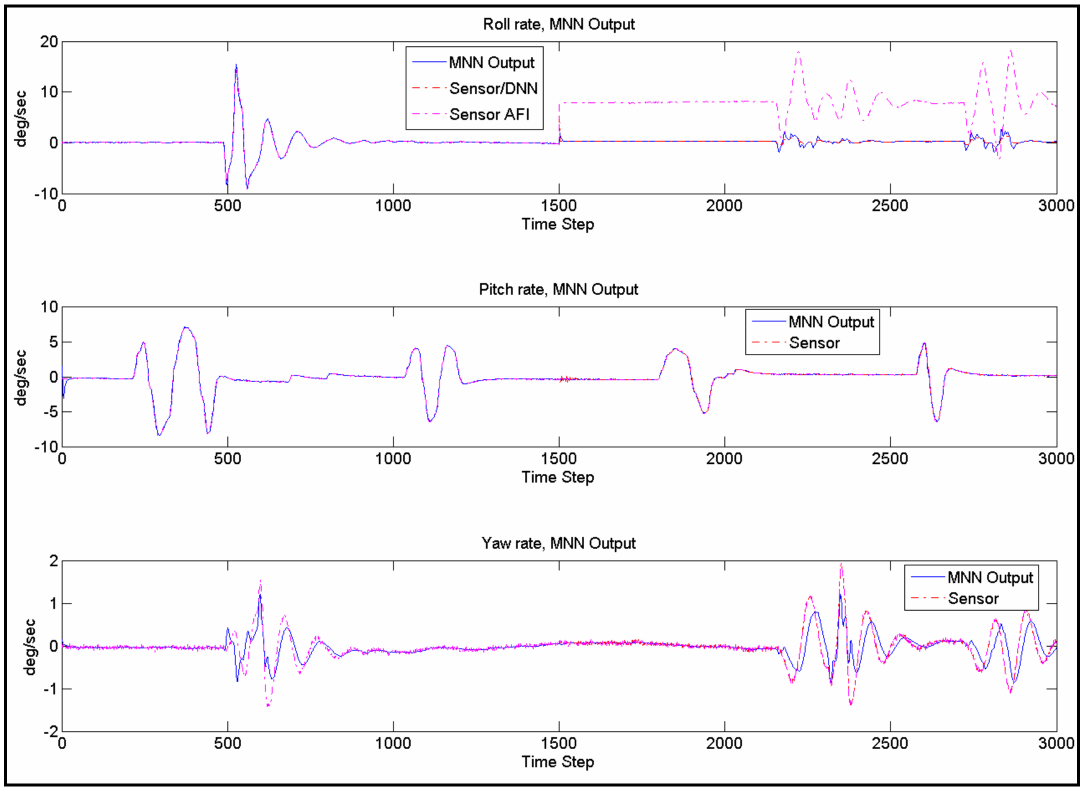


Figure 6-39 : MNN Outputs for Failure on P (8 deg/s Bias)

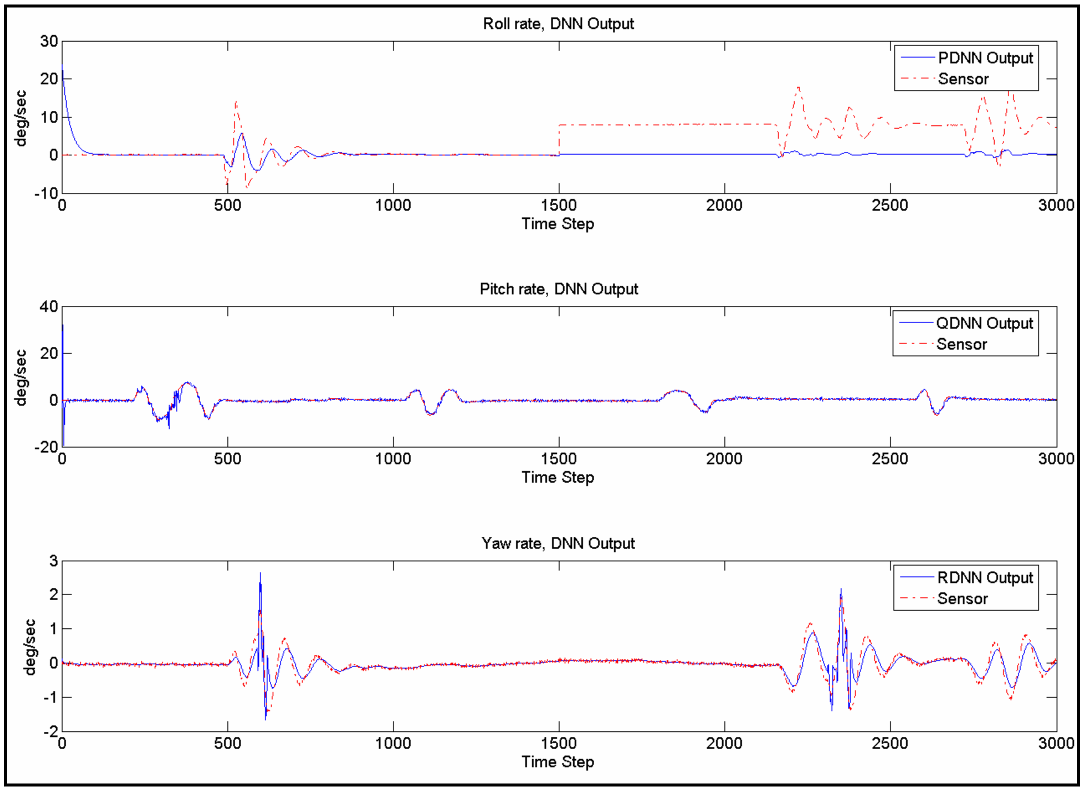


Figure 6-40 : DNN Outputs for Failure on P (8 deg/s Bias)

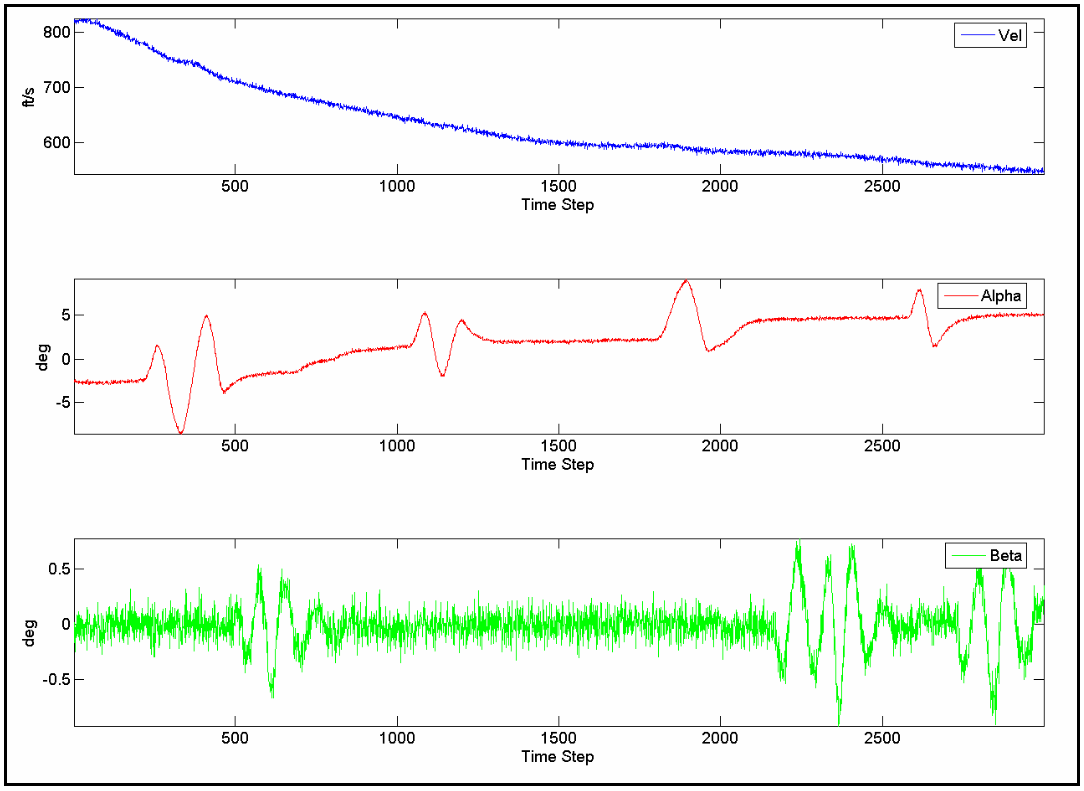


Figure 6-41 : Velocity, Alpha and Beta

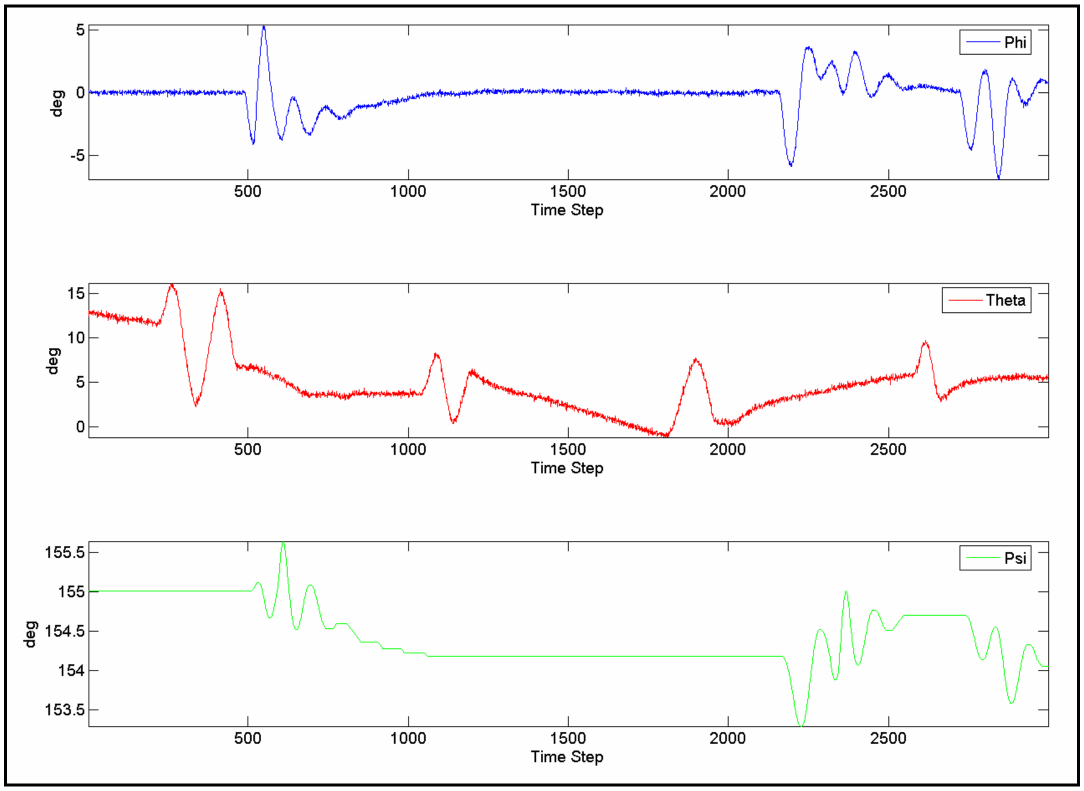


Figure 6-42 : Euler Angles

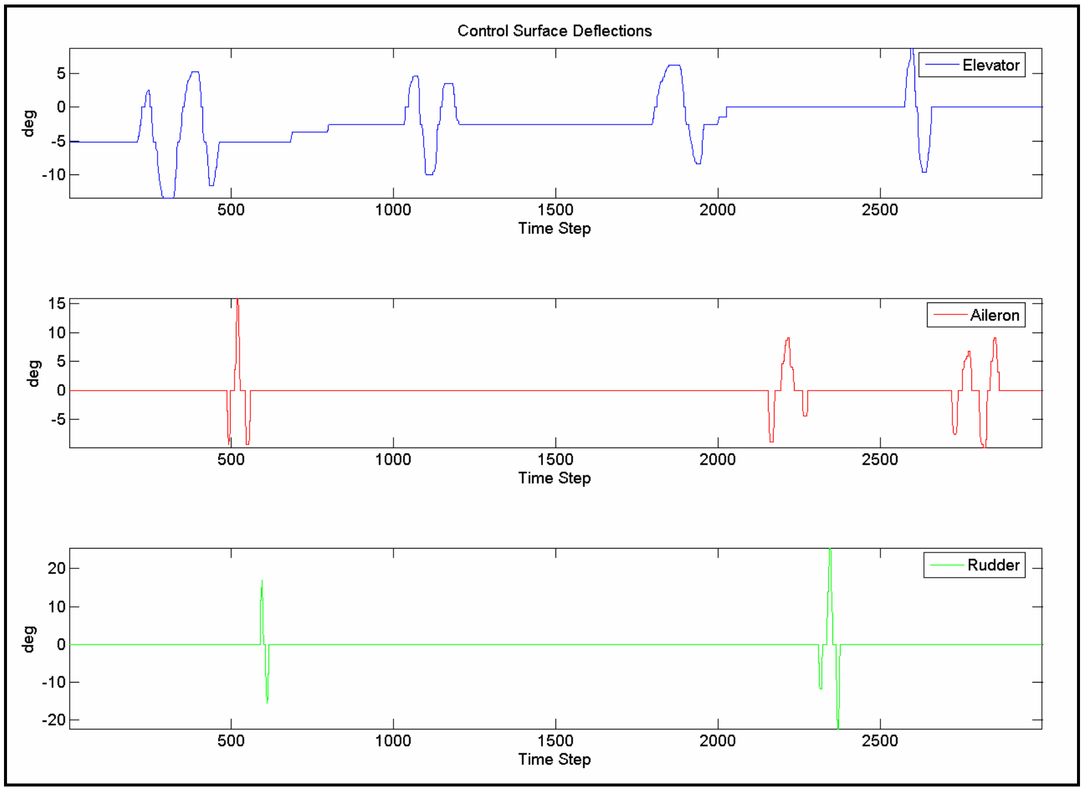


Figure 6-43 : Control Surface Deflections

The following table presents the results of post failure analysis of the estimation errors of the MNN as well as the DNNs for small and large failures on the three gyros. The results are tabulated from three runs of the failure scenarios.

Table 6-6 : Post Failure Estimation Error Analysis for MNN and DNNs

Failed Gyro	**Magnitude**													
			MNN (P)		**MNN (Q)**		**MNN (R)**		**P -DNN**		**Q - DNN**		**R – DNN**	
			Mean (deg/s)	**Std (deg/s)**	**Mean (deg/s)**	**Std (deg/s)**	**Mean (deg/s)**	**Std (deg/s)**	**Mean (deg/s)**	**Std (deg/s)**	**Mean (deg/s)**	**Std (deg/s)**	**Mean (deg/s)**	**Std (deg/s)**
P	**Small**	1	0.0025	0.1485					-0.2996	2.4389				
		2	0.0031	0.2408					-0.3938	3.9541				
		3	0.0030	0.2710					-0.0820	3.0787				
	Large	1	0.005	0.2851					-0.2491	2.5425				
		2	0.0052	0.3011					0.4324	5.3915				
		3	0.0054	0.1741					-0.5300	2.8449				
Q	**Small**	1			0	0.4840					-1.0572	1.9829		
		2			0	0.6407					-1.3475	1.1470		
		3			-0.0020	0.4670					-1.4440	0.8234		
	Large	1			0.0058	0.6535					-1.8091	1.5893		
		2			-0.0011	0.4177					-2.8651	1.5451		
		3			-0.0027	0.3587					-3.6574	1.8354		
R	**Small**	1					-0.0021	0.2452					-0.4275	0.3643
		2					0	0.2476					-0.5766	0.4842
		3					-0.0011	0.1565					-0.2128	0.2905
	Large	1					-0.0037	0.2911					-0.6409	0.5094
		2					-0.0019	0.1124					-0.5555	0.1796
		3					-0.0028	0.1621					-0.7719	0.3424

The following section describes the AFDIA procedures, the setup of the simulation runs and results from the parallel implementation of the scheme on the Beowulf cluster.

6.5 Actuator Failure Detection Identification and Accommodation

The need for an adaptive flight control scheme is perhaps more acute in the case of actuator failure as compared to sensor failure, as it is possible to have physical redundancy for sensors onboard the aircraft, it is not possible to have physical redundancy for the control surfaces and the lift generating sections of an airplane. The use of neural networks for state estimation provides an attractive option to develop an analytical control scheme that would reconfigure the control structure of the aircraft, in the event of an actuator failure. The goals of this section of the research effort is the detection, identification and accommodation of failures on the control surfaces of an aircraft, using NN based controllers to drive the remaining “healthy” control surfaces. Multiple failures on control surfaces, catastrophic failures such the loss of an entire wing or a portion of the aircraft fuselage on the aircraft structures, and failures on the propulsion systems onboard an aircraft are not considered within this research effort.

Within the scope of this research effort, actuator failures are defined and simulated as a drop in surface efficiency, resulting from partial loss of surface, such as a missing part of a control surface or stuck control surfaces, or a combination of the two and are divided into four basic configurations as:

1. Stuck Left Elevator at trim/non-trim deflection, without missing surface;
2. Stuck Left Elevator at trim/non-trim deflection, with missing surface;
3. Stuck Left Aileron at trim/non-trim deflection, without missing surface;
4. Stuck Left Aileron at trim/non-trim deflection, with missing surface.

The mathematical modeling of actuator failures in the presence of control surface loss is derived through a set of closed form expressions for the non-dimensional stability and control derivatives and is a function of the normal force coefficients of the control surface with the failed actuator. These equations for longitudinal stability derivatives given below:

$$c_{L_\alpha} = K_{1,1} + K_{1,2}c_{L_{\delta S_R}} + K_{1,3}c_{L_{\delta S_L}} \qquad \text{Eq 6.7}$$

$$c_{m_\alpha} = K_{2,1} + K_{2,2}c_{L_{\delta S_R}} + K_{2,3}c_{L_{\delta S_L}} \qquad \text{Eq 6.8}$$

$$c_{L_{\dot\alpha}} = K_{3,1} + K_{3,2}c_{L_{\delta S_R}} + K_{3,3}c_{L_{\delta S_L}} \qquad \text{Eq 6.9}$$

$$c_{m_{\dot\alpha}} = K_{4,1} + K_{4,2}c_{L_{\delta S_R}} + K_{4,3}c_{L_{\delta S_L}} \qquad \text{Eq 6.10}$$

$$c_{L_q} = K_{5,1} + K_{5,2}c_{L_{\delta S_R}} + K_{5,3}c_{L_{\delta S_L}} \qquad \text{Eq 6.11}$$

$$c_{m_q} = K_{6,1} + K_{6,2}c_{L_{\delta S_R}} + K_{6,3}c_{L_{\delta S_L}} \qquad \text{Eq 6.12}$$

The implementation of the AFDIA scheme is similar to the SFDIA scheme and has four distinct neural networks, namely the *Main Neural Network* (MNN), the *pitch*, *roll* and *yaw Neural Network Controllers* (NNCs). The outputs of the MNN are the same as they are for the SFDIA scheme and are the estimates of the outputs of the *pitch*, *roll* and *yaw* gyros, at a time instant *k*. The outputs of the NNCs are the estimates of the control surface deflections at a time instant *k*, computed from the time histories of measurements from the onboard sensors. Under nominal conditions, the controller emulates the nominal deflections of the control surfaces and does not assume any physical meaning. However, training under nominal conditions would help improve the transient response of the network, following an actuator failure and the subsequent actuator failure accommodation process. The error cost function that is minimized during the training process is given as

$$J_{x_{nom}} = (\delta_{y_{L,R}} - \hat{\delta}_{y_{L,R}}), \text{ where } x = \{pitch, roll, yaw\},\ y = \{Elev, Ail, Rud\} \quad \text{Eq 6.13}$$

When an actuator failure occurs, it results in a change in the aircraft trim conditions due to changes in the aerodynamic forces and moments. Failures on the control surfaces could also lead to a reduction in their effectiveness, especially in cases where there is a loss of surface. With the occurrence of such failures on the aircraft control surfaces, there results a change in the angular velocities of the aircraft, consequently inducing a noticeable change in the value of the quadratic error parameter *MQEE* in the MNN. If this value exceeds a certain predefined threshold, a failure is implied. This part constitutes the Actuator Failure Detection (AFD) component of the fault tolerant flight control scheme. Following AFD, actuator failure identification is achieved by observing the auto and cross correlation coefficients between the angular rates *p*, *q* and *r*. In general, for two random processes, the cross correlation coefficient is defined by

$$R_{YX}(n) = E[Y(k)X(k+n)] \quad \text{Eq 6.14}$$

The MNN monitors the error between the estimates of the *roll*, *pitch* and *yaw* rate gyros and the actual measurements from the sensors themselves. If this error metric exceeds a predefined threshold, then a flag is raised, indicating the occurrence of a possible failure. Then MNN then examines the auto and cross correlation coefficients $R_{pq}, R_{qr}, R_{pr}, R_{tt}, R_{ff}$.

The most significant effect of an actuator failure is the loss of symmetry on the aircraft, which in turn would introduce a dynamic coupling between the longitudinal and lateral directional aircraft dynamics. This dynamic coupling would result in significant changes in the cross-correlation functions R_{pq}, R_{pr}, R_{qr} and the auto correlation coefficients R_{tt}, R_{ff}, which can be then used to identify the failed actuator. Once AFI is successfully determined, failure accommodation is achieved by generating compensating control surface deflections as generated by the individual NNCs.

In the case of the pitch NNC, an error cost function defined as:

$$J_{Pitch_{AFA}} = k_1(q - q_{ref}) + k_2(\theta - \theta_{ref}) + k_3(\dot{q} - \dot{q}_{ref}) \qquad \text{Eq 6.15}$$

where, $q_{ref} = 0; \theta_{ref} = \theta_{trim}; \dot{q}_{ref} = 0$

is minimized in the longitudinal direction to generate the compensating Elevator deflections for Actuator Failure Accommodation (AFA) in the longitudinal direction. At the same time, compensating control surface deflections are generated from the roll and yaw NNC so that the following cost functions are minimized in the lateral direction. These cost functions are defined as:

$$J_{Roll_{AFA}} = k_4(p - p_{ref}) + k_5(\phi - \phi_{ref}) \qquad \text{Eq 6.16}$$

$$J_{Yaw_{AFA}} = k_6(r - r_{ref}) \qquad \text{Eq 6.17}$$

In this research effort, the NNCs are executed in parallel on individual compute nodes on the cluster and the compensating control surface deflections that are generated by the controllers are sent to the MNN using the MPI directive MPI_Send. Once all the control surface deflections are received within the MNN, it sends the same across the other socket interface to the AVDS environment. The MNN also sends a flag indicating the detection and identification of actuator failure along with the compensating control surface deflections. Code segments within the AVDS environment read the status of these flags to determine if the control surface deflections that are used in the dynamic simulation are to be from the pilot stick inputs or from the controllers.

Within this research effort, the structure of the AFDIA scheme is built in such a way that once actuator failures are detected and identified, the outputs from the controller takes over the control of the aircraft and the pilot input is completely cut off. Improvements to this setup could be to make use of a combination of the surface deflection commands from the pilot as well as the commands from controller outputs.

The block diagram of the AFDIA scheme is shown in Figure 5.2

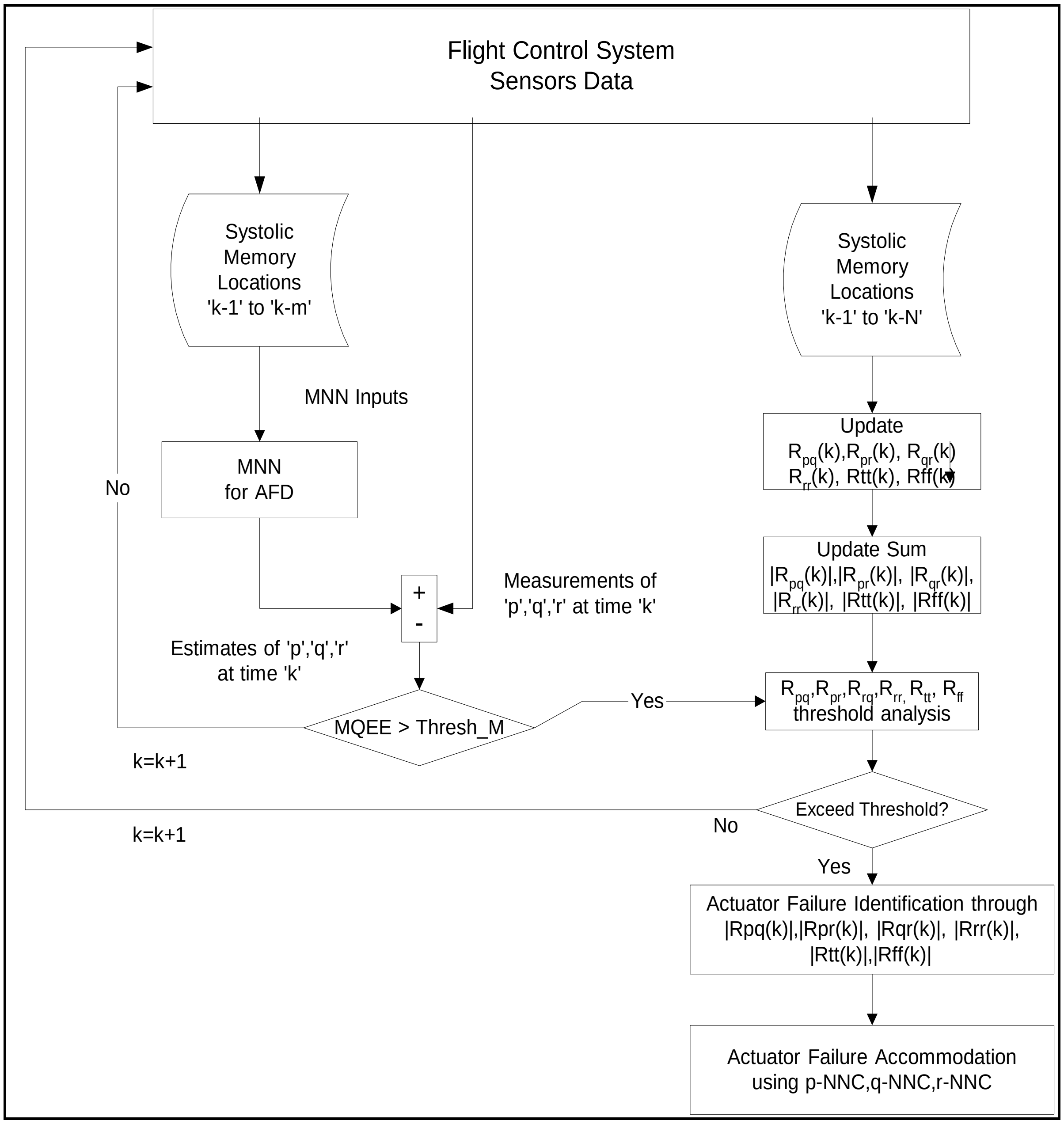

Flight Control System Sensors Data
Systolic Memory Locations 'k-1' to 'k-m'
Systolic Memory Locations 'k-1' to 'k-N'
MNN Inputs
MNN for AFD
No
Update $R_{pq}(k)$,$R_{pr}(k)$, $R_{qr}(k)$ $R_{rr}(k)$, Rtt(k), Rff(k)
Update Sum $|R_{pq}(k)|$,$|R_{pr}(k)|$, $|R_{qr}(k)|$, $|R_{rr}(k)|$, |Rtt(k)|, |Rff(k)|
+
-
Measurements of 'p','q','r' at time 'k'
Estimates of 'p','q','r' at time 'k'
MQEE > Thresh_M
Yes
R_{pq},R_{pr},R_{rq},R_{rr}, R_{tt}, R_{ff} threshold analysis
k=k+1
Exceed Threshold?
No
k=k+1
Yes
Actuator Failure Identification through |Rpq(k)|,|Rpr(k)|, |Rqr(k)|, |Rrr(k)|, |Rtt(k)|,|Rff(k)|
Actuator Failure Accommodation using p-NNC,q-NNC,r-NNC

Figure 6-44 : Block Diagram of the AFDIA Scheme

6.6 AFDIA On-line Simulation and Implementation Results

Similar to the procedure followed for the SFDIA purposes, the networks representing the controllers were trained offline. The NNs ROLL-NNC, PITCH-NNC and YAW-NNC were trained offline for 100 epochs, using flight data with 5000 data points, generated at 30 Hz, from the AVDS environment. These trained networks were then tested every 10 epochs, starting from the 50th epoch, with data that it had not seen during its training cycle. During the training cycle, the weights and thresholds of the networks were saved after every 10 epochs. The offline training for the neural network controllers was primarily used to narrow down the structure of the networks, particularly with regard to the inputs, the history time window, the number of hidden layer neurons, and the learning and momentum rates. Unlike the SFDIA procedure, these pre-trained networks were not used within the AFDIA scheme. With respect to the MNN, the previously trained network structure for the SFDIA procedure was used as the baseline network architecture for the on-line implementation of the AFDIA process. The following table gives a listing of the structure of the NNs used for the purpose of actuator failure detection, identification, and accommodation.

Table 6-7 : Structures of NN for AFDIA

Network	No. of Inputs	TW	ILN	HLN	Outputs	LR	MR
MNN	8	5	40	25	3	0.05	0.005
Roll NNC	5	3	15	12	1	0.1	0.005
Pitch-NNC	3	2	6	20	1	0.1	0.005
Yaw-NNC	5	3	15	20	1	0.1	0.005

Similar to the setup procedure for SFDIA, once the parameters of the AFDIA procedure were set using the Matlab GUI, they were sent across the socket communication channel to the main node on the Beowulf cluster. The parameters include the surface that is scheduled to fail, the time at which the failure is supposed to occur, the magnitude of failure and the resulting effectiveness of the surface that failed. When the main node receives the simulation parameters data, it distributes the same to all the compute nodes on the cluster. As described in section 5.3, once this process is complete, the main node waits for the initialization string from AVDS. Once initialization string from AVDS is received, the main node sends the task identification number, the total time of simulation, and the failure specifications across to AVDS and then waits for the "pilot" to start flying the aircraft. Within AVDS, once the task identification number and the total time of simulation are received, the "pilot" takes off and climbs to an altitude over 3000 ft. once the aircraft breaches the 3000 ft barrier, AVDS starts its two-way communication with the main node and sends sensor data at a rate of 30 Hz. Within the main node, once the data is received from AVDS, it is distributed across the compute nodes to generate the estimates of the *roll*, *pitch* and *yaw* rates. These estimates are then sent back to the main node where the error metrics are computed and stored. Once the time of failure is reached, within the AVDS environment, failure is injected on the selected surface. After this instant, the failed stops receiving its input from the pilot stick, and stays fixed at the set position. On the other hand, all the other surfaces continue to receive their inputs form the pilot stick. The aircraft dynamics are simulated under this configuration until a failure is declared by the MNN, on the main node. On the main node, once the failure time is reached, the MNN begins checking these error metrics for failures.

When an actuator failure occurs, it generates noticeable changes in the aircraft trim conditions a well as the aerodynamic forces and moments acting on the aircraft. This typically leads to coupling between the lateral directional dynamics and the longitudinal directional dynamics. This coupling can be seen in the output generated by the MNN. After the time of failure is reached, the MNN begins checking the error metrics. When a predefined threshold on the MNN error metrics exceeds a certain value, a failure condition is declared. Once a failure condition is declared, the next step is the identification, which is done by monitoring the auto correlation coefficients R_{tt} and R_{ff}. if the thresholds on these coefficients are exceeded, and then the appropriate failure is identified as either failure on the Elevators or Failures on the Ailerons. Following failure detection and identification, a flag indicating such an event is set and sent as a part of the data stream to the NNCs at the next time step, indicating that a failure had occurred at the previous step. Once the MNN completes this section of its task, it sends a vector of 6 elements, ordered as the Pitch-NNC outputs for the elevator deflection in the first two elements, the Roll-NNC outputs for the aileron deflection in the next two elements, the Yaw-NNC output for the rudder deflection as the fifth element and a flag indicating the detection and identification of an actuator failure as the sixth element across the socket to the AVDS environment. At each time step, within the NNCs, a code segment checks for the status of the failure flag to determine if the MNN had declared a failure on any control surface, at the previous time step. If the flag is set to indicate a failure, then the NNCs switch their failure mode error generation configuration. Unlike the case of SFDIA, the learning of the NNCs is not suspended and continues to generate the compensating control surface deflections according to the error formulation. Now, within the AVDS environment, another segment of code checks the vector of data that it received from the

MNN at the previous time step to see if the MNN declared a failure. If the MNN had declared a failure, then the code segment accordingly switches to the outputs of the controller as the control surface deflection inputs to the aircraft dynamics.

Once the failure on the control surface is declared and identified, controller takes total authority of the aircraft and attempts to stabilize the aircraft with the help of the remainder of the healthy control surfaces. In this setup for this research study, the pilot is totally removed from the loop.

Within this research effort, as mentioned earlier, failures on only the Elevators and the Ailerons were considered. Simulations were run on the cluster and the results from the simulations are tabulated below. The following failure scenarios on the left elevator scenarios were considered.

- 5 deg failure on Left Elevator, with 50% reduction in efficiency, at 50 sec.
- 5 deg failure on Left Elevator, with 100% reduction in efficiency, at 50 sec.
- 15 deg failure on Left Elevator, with 50% reduction in efficiency, at 50 sec.
- 15 deg failure on Left Elevator, with 100% reduction in efficiency, at 50 sec.
- 0 deg failure on Left Elevator, with 50% reduction in efficiency, at 50 sec.
- 0 deg failure on Left Elevator, with 100% reduction in efficiency, at 50 sec.

On the Left Aileron, the following failures were considered

- 5 deg failure on Left Aileron, with 50% reduction in efficiency, at 50 sec.
- 5 deg failure on Left Aileron, with 0% reduction in efficiency, at 50 sec.
- 15 deg failure on Left Aileron, with 50% reduction in efficiency, at 50 sec.
- 15 deg failure on Left Aileron, with 0% reduction in efficiency, at 50 sec.
- 0 deg failure on Left Aileron, with 50% reduction in efficiency, at 50 sec.
- 0 deg failure on Left Aileron, with 0% reduction in efficiency, at 50 sec.

The results of parallel AFDIA implementation on the cluster can be tabulated as follows.

Table 6-8 : Results of Failure Detection and Identification on Left Elevator and Aileron

Scenario		Main error threshold exceeded at	Autocorrelation threshold exceeded at	Failure Identification
Elev (L)	Ail (L)			
5 deg, 50%		50.133 s	51.866 s	Yes
5 deg, 100%		50.125 s	50.00 s	-
15 deg, 50%		50.172 s	54.910 s	Yes
15 deg, 100%		50.1 s	51.3	Yes
0 deg, 50%		50.133 s	59.133 s	Yes
0 deg, 100%				
	5 deg, 50%	50.066 s	53.7 s	Yes
	5 deg, 0%	50.066 s	51.5 s	Yes
	10 deg, 50%	50.066 s	51.766 s	Yes
	10 deg, 0%	No data	No data	No data
	0 deg, 50%	50.078 s	-	No
	0 deg, 0%	50.133 s	-	No

Statistical analysis was performed on the flight data sets from the simulation environment to determine the Receiver Operating Characteristic (ROC) curve, to achieve a good trade off between true positive rates and false positive rates while detecting the presence of actuator failures. A 2x2 table representing the true and false positives and negatives, used to determine the ROC curves is shown below in Table 6-9.

Table 6-9 : 2 by 2 Table for Determining Failure Detection Rates

	Failure Present	Failure Absent
Failure Detected	True Positive	False Positive
Failure Not Detected	False Negative	True Negative

The corresponding ROC curves for failure detection on the Left Aileron and the Left Elevator are shown in the following figures.

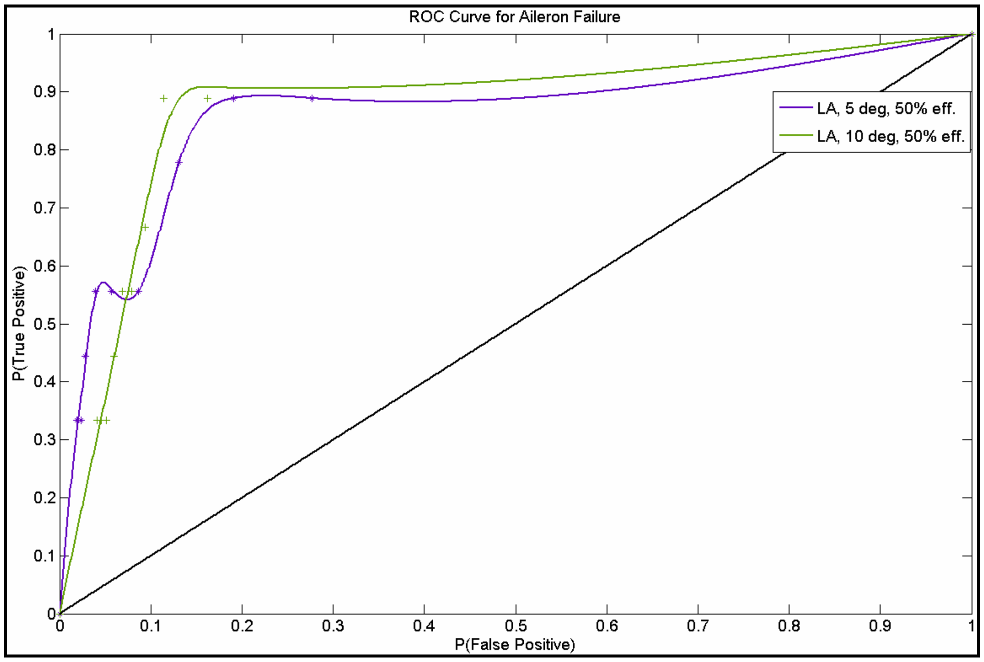


Figure 6-45 : ROC Curve for Aileron Failure

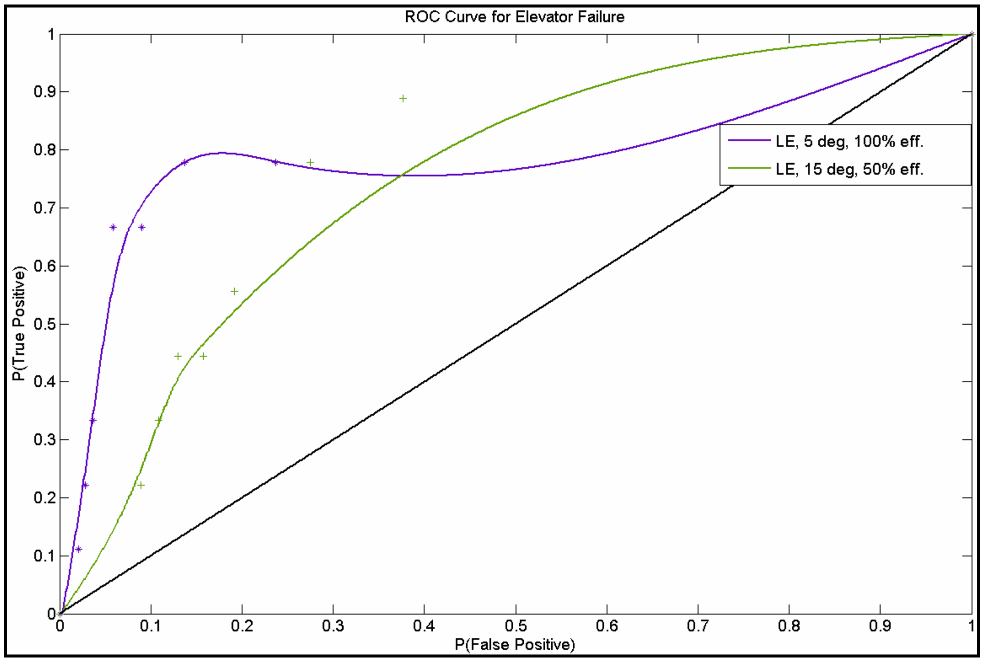


Figure 6-46 : ROC Curve for Elevator Failure

The following table presents the results of post failure analysis of the tracking errors of the controller for failures on the Ailerons and Elevator. The results are tabulated from three runs of the failure scenarios.

Table 6-10 : Post Failure Tracking Error Analysis for Failures on Left Elevator and Aileron

Failed Surface	Failure Magnitude (deg)							
			P (deg/s)		Q (deg/s)		R (deg/s)	
		Eff.	Mean	Std	Mean	Std	Mean	Std
Left Elevator	0	50%	-0.0425	0.2623	0.0359	0.4536	-0.0621	0.1295
		100%	0.0068	0.2683	0.0208	0.9440	-0.0416	0.1215
	5	50%	-0.0071	0.2053	-0.0173	0.4133	0.0303	0.1328
		100%	NA	NA	NA	NA	NA	NA
	15	50%	-0.0035	0.1763	0.0161	0.3372	-0.0882	0.0925
		100%	0.0039	0.2735	0.0382	0.6991	-0.0393	0.1176
Left Aileron	0	50%	NA	NA	NA	NA	NA	NA
		0%	NA	NA	NA	NA	NA	NA
	5	50%	-0.0336	2.6439	0.3099	0.6814	0.3204	0.6442
		0%	5.1164	10.2927	0.7769	2.0945	0.1052	0.7835
	10	50%	4.9556	10.0559	1.1856	3.4311	0.5455	0.7988
		0%	No data	No data	No data	No data	No data	No data

The following figures 6-45 to 6-50 represent the results of Actuator Failure detection, Identification and Accommodation on the Left Elevator for a 5 deg failure with 50% reduction in the efficiency of the failed surface. The error threshold on the MNN is exceeded at 50.133s, indicating a failure on the sensor subsystems. Subsequently, the error on the R_{tt} exceeds its thresholds at 51.866s, identifying the failure.

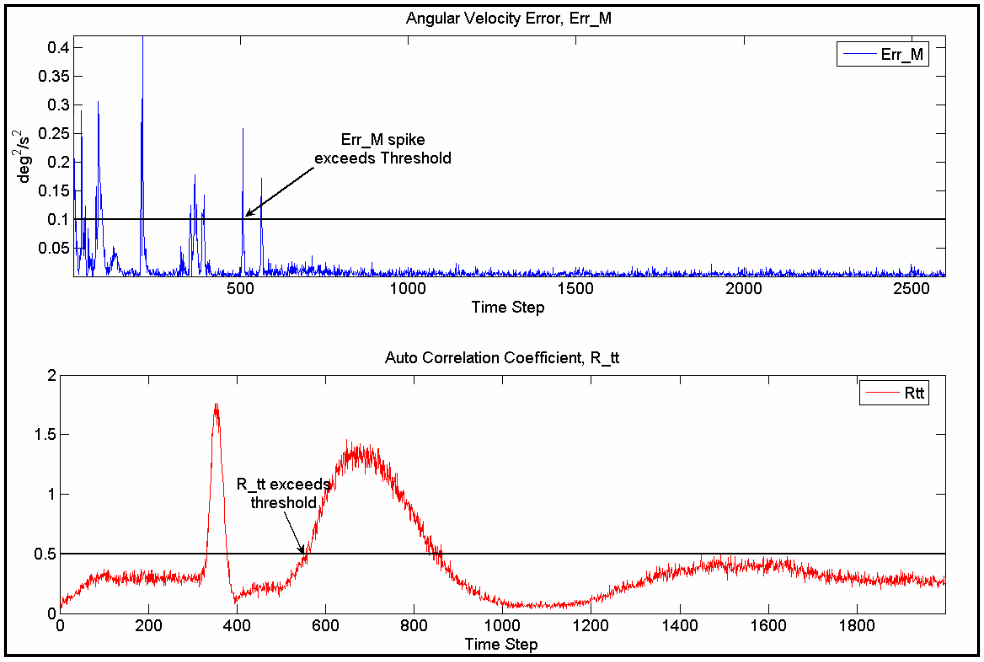


Figure 6-47 : Error Thresholds for Failure on Left Elevator (5 deg, 50% eff.)

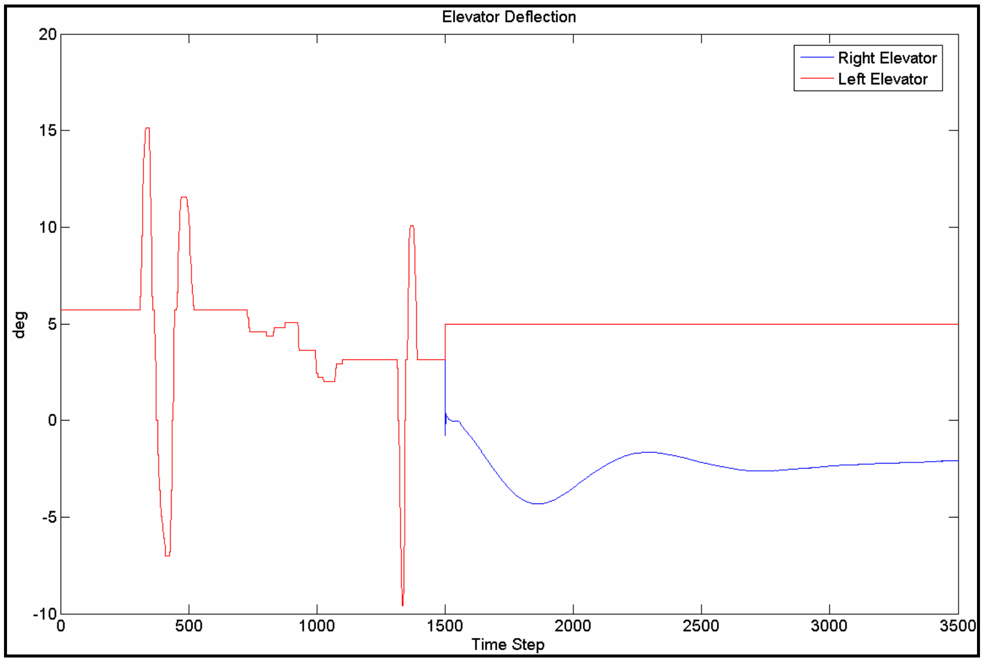


Figure 6-48 : Control Surface Deflections, Failure on Left Elevator (5 deg, 50% eff.)

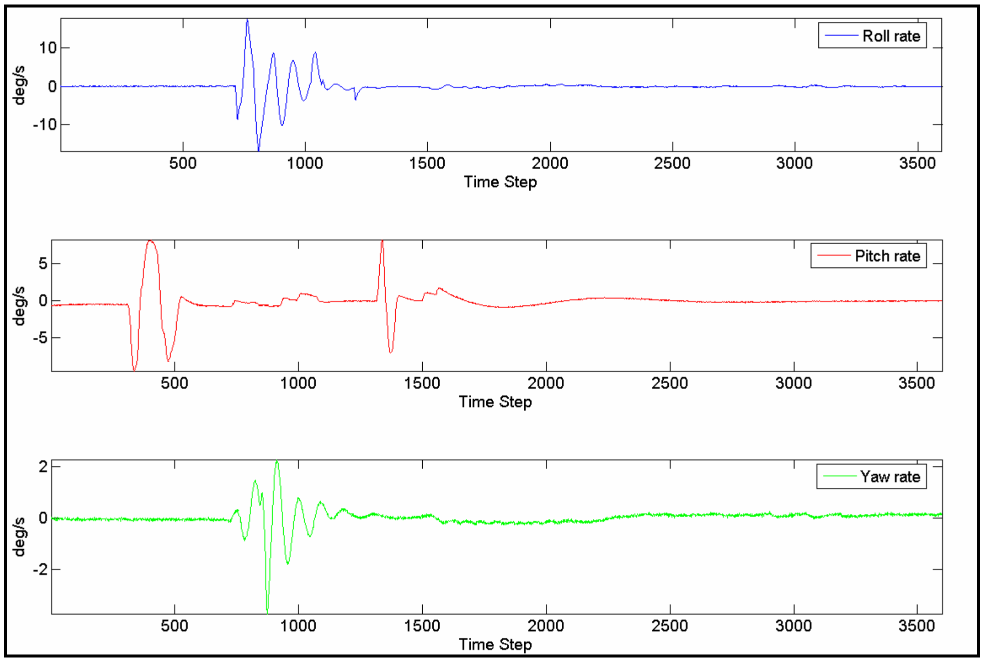


Figure 6-49 : Angular Rates of the Aircraft, for Failure on Left Elevator (5 deg, 50% eff.)

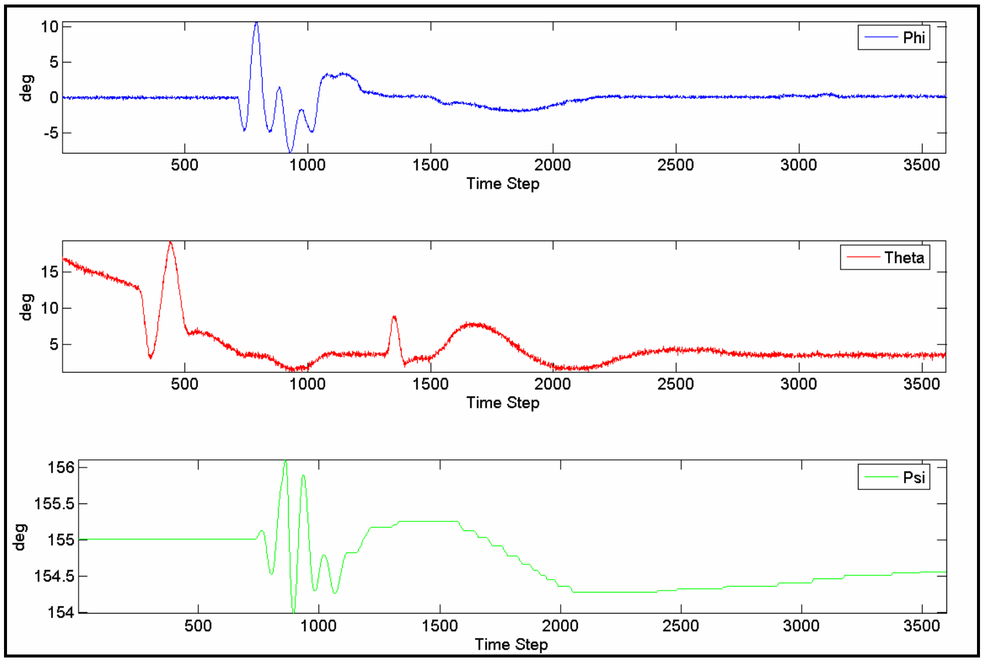


Figure 6-50 : Euler Angles, for Failure on Left Elevator (5 deg, 50% eff.)

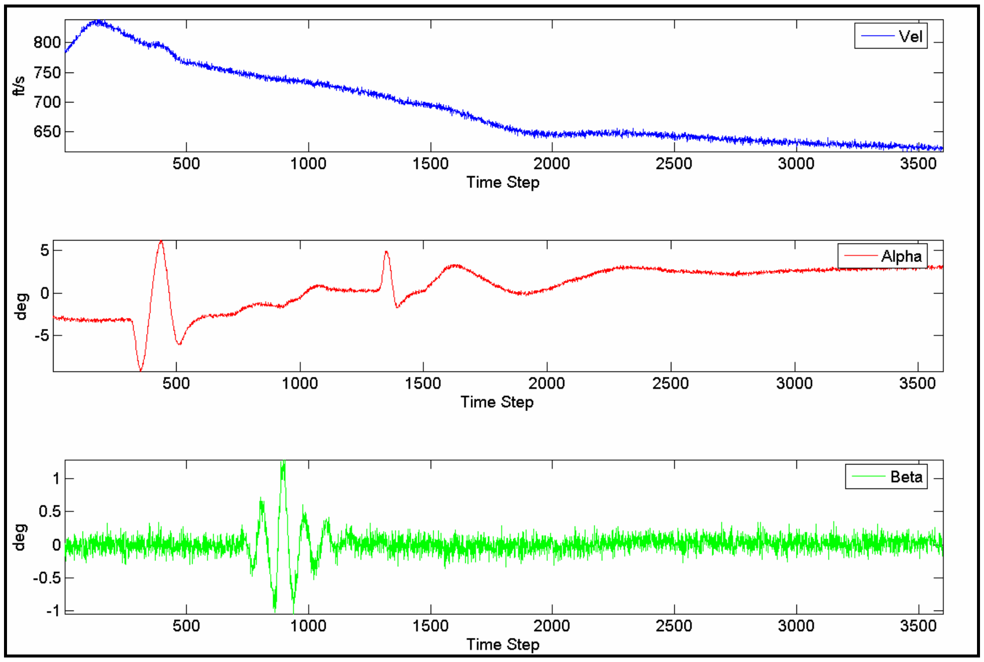


Figure 6-51 : Euler Angles, for Failure on Left Elevator (5 deg, 50% eff.)

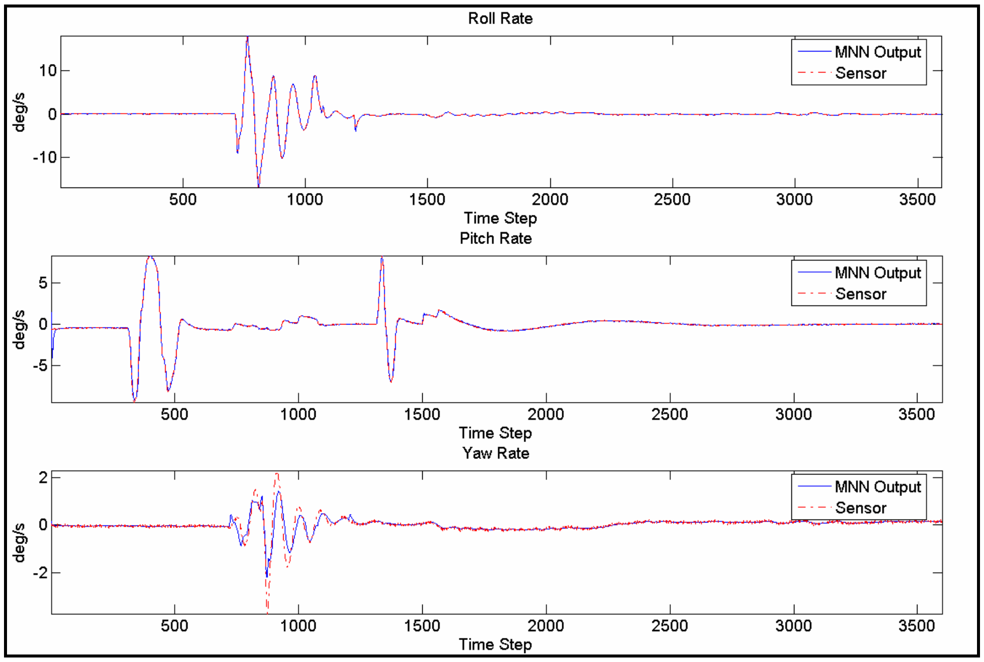


Figure 6-52 : MNN Outputs, for Failure on Left Elevator (5 deg, 50% eff.)

The following figures 6-51 to 6-56 represent the results of Actuator Failure detection, Identification and Accommodation on the Left Elevator for a 0 deg failure with 50% reduction in the efficiency of the failed surface. The error threshold on the MNN is exceeded at 50.133s, indicating a failure on the actuators. Subsequently, the error on the R_{tt} exceeds its thresholds at 59.133s, identifying the nature of the failure in terms of the failed surface.

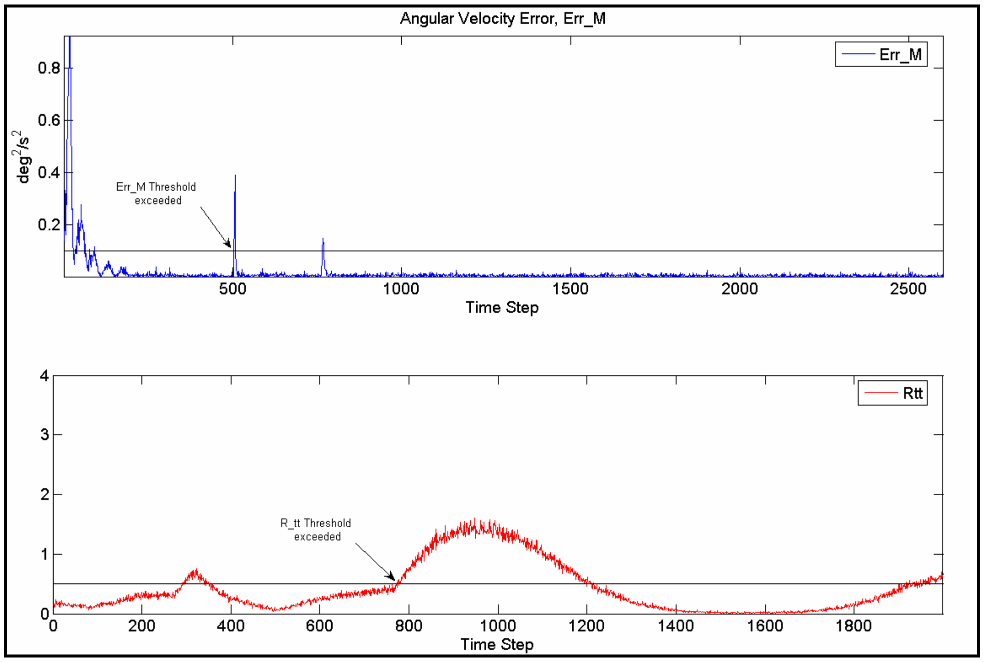


Figure 6-53 : Error Thresholds for Failure on Left Elevator (0 deg, 50 % eff.)

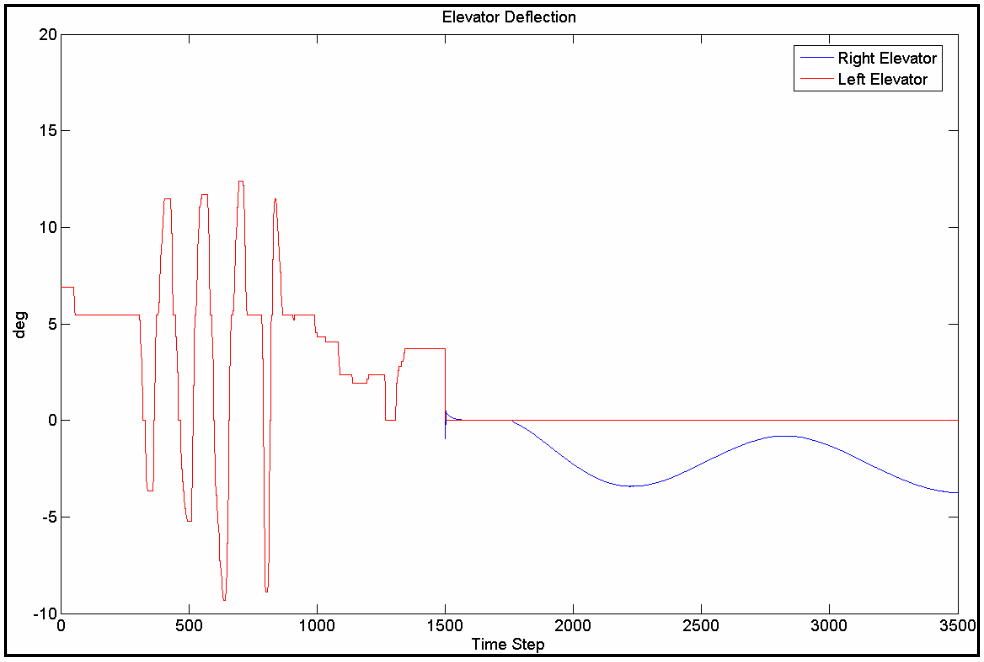


Figure 6-54 : Control Surface Deflections, for Failure on Left Elevator (0 deg, 50 % eff.)

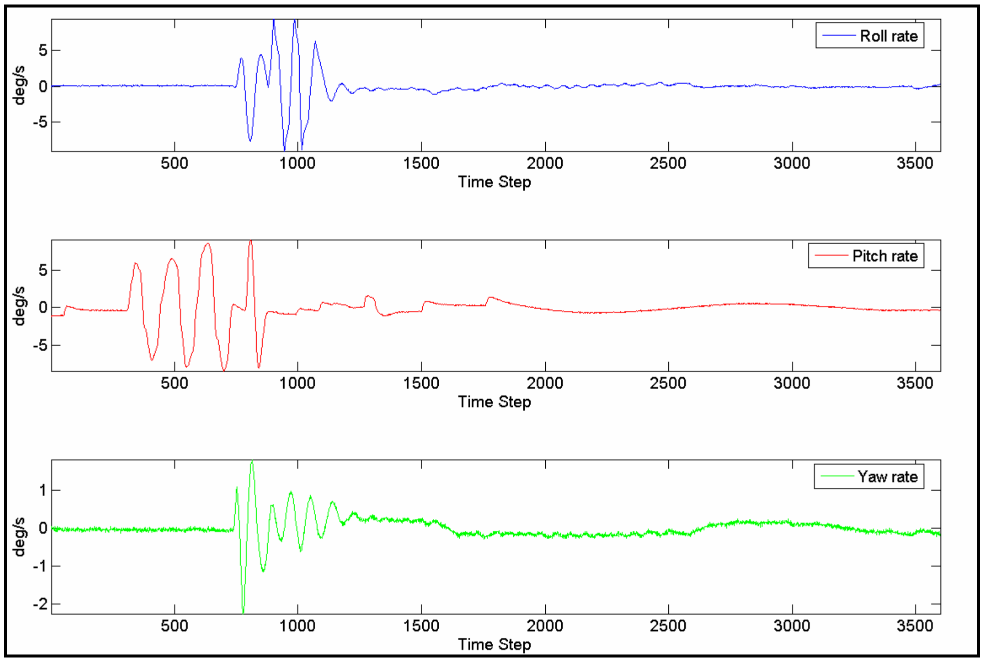


Figure 6-55 : Aircraft Angular rates, for Failure on Left Elevator (0 deg, 50 % eff.)

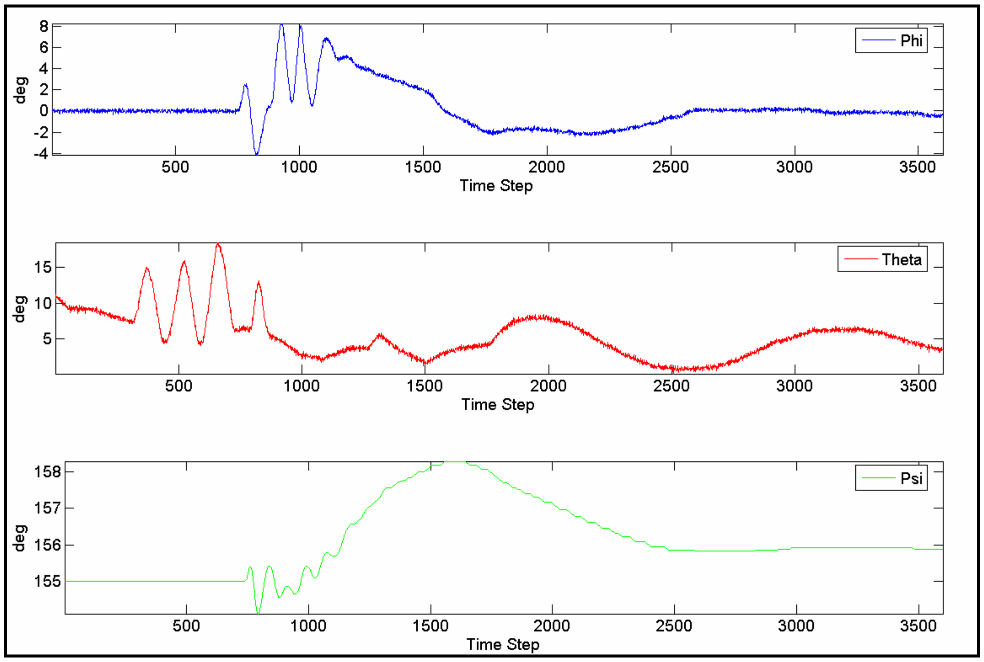


Figure 6-56 : Euler Angles, for Failure on Left Elevator (0 deg, 50 % eff.)

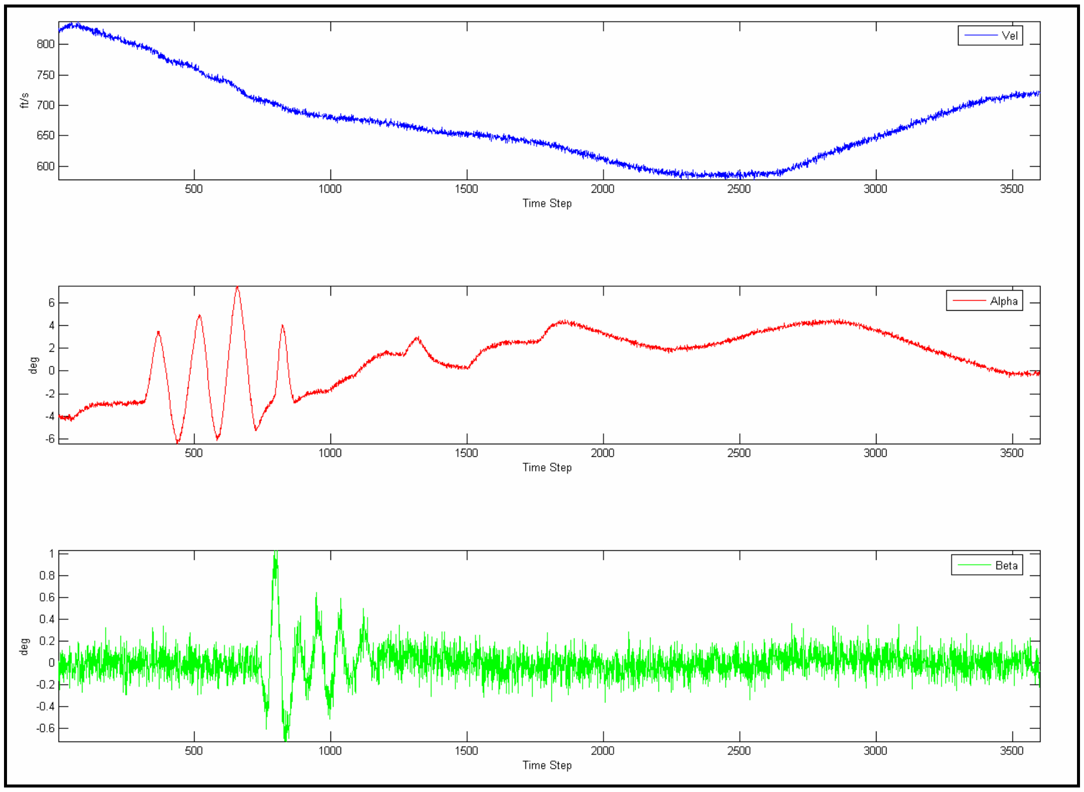


Figure 6-57 : Velocity, Alpha and Beta, for Failure on Left Elevator (0 deg, 50 % eff.)

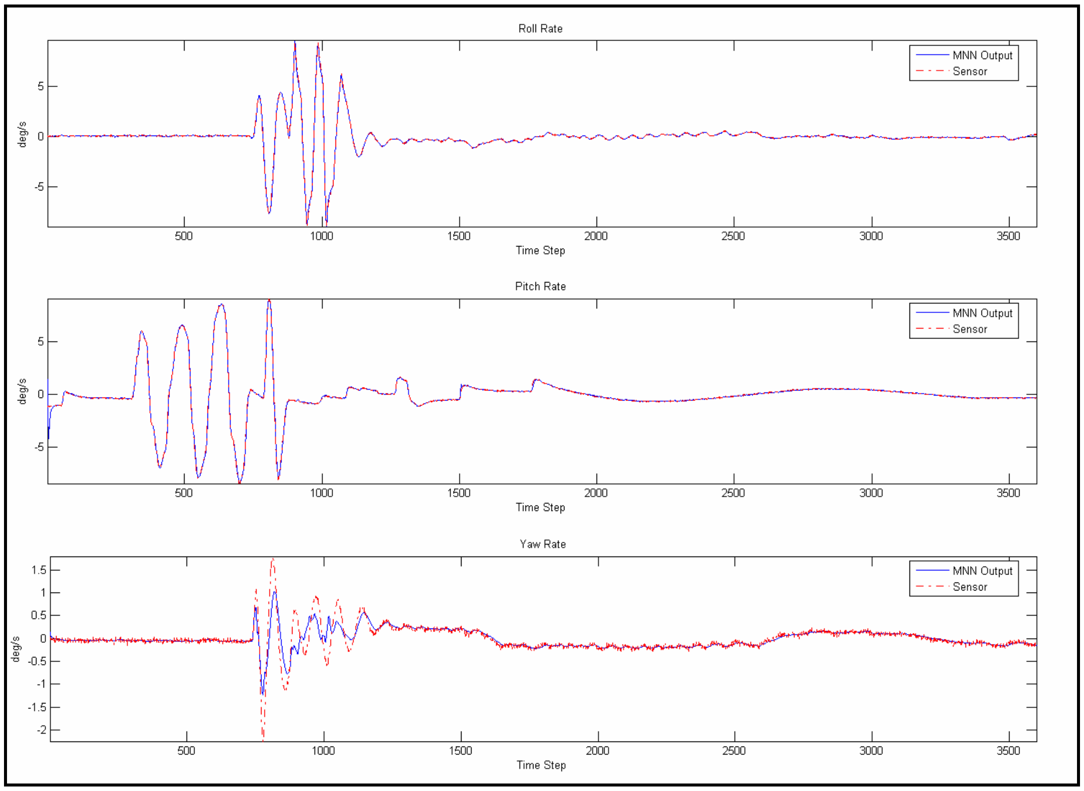


Figure 6-58 : MNN Outputs, for Failure on Left Elevator (0 deg, 50 % eff.)

The following figures 6-57 to 6-62 represent the results of Actuator Failure Detection, Identification and Accommodation on the Left Aileron for a 5 deg failure with 50% reduction in the efficiency of the failed surface. The error threshold on the MNN is exceeded at 50.066s, indicating a failure on the actuators. Subsequently, the error on the R_{ff} exceeds its thresholds at 53.7s, identifying the failure.

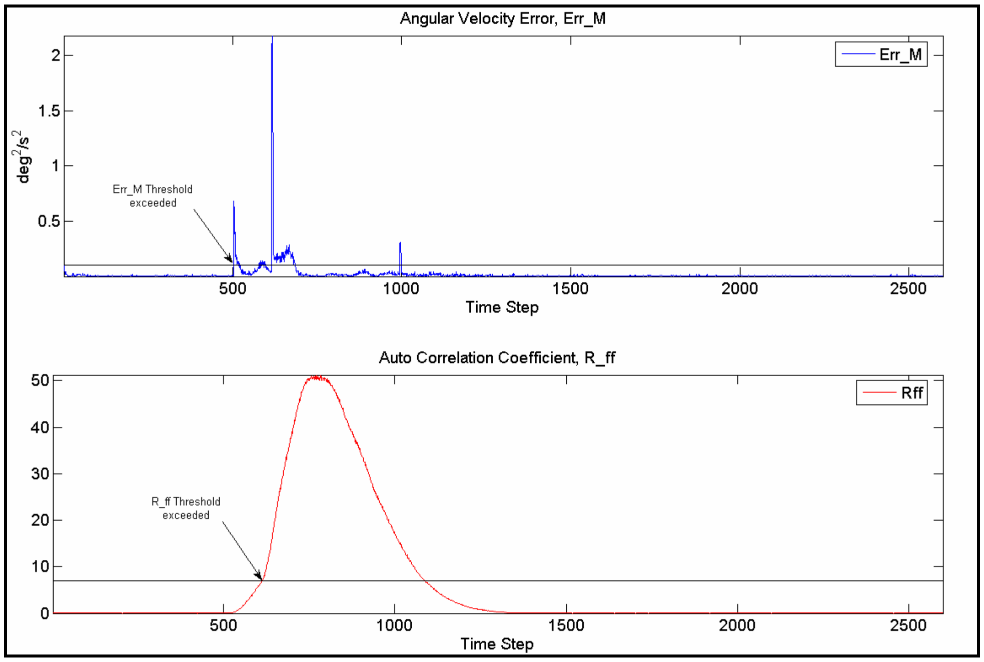


Figure 6-59 : Error Thresholds for Failure on Left Aileron (5 deg, 50% eff.)

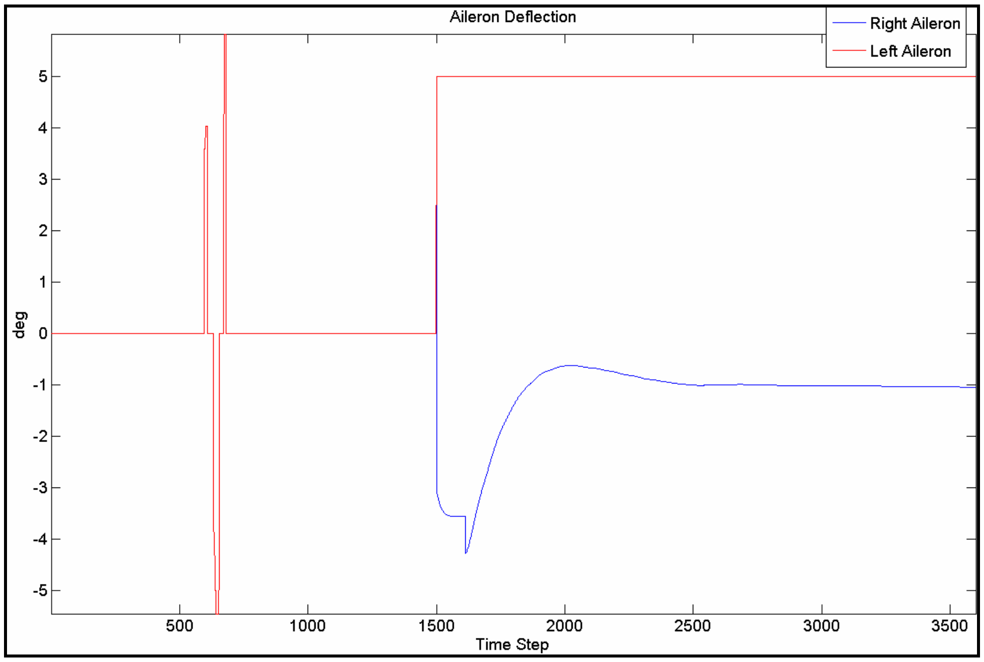


Figure 6-60 : Control Surface Deflection, for Failure on Left Aileron (5 deg, 50% eff.)

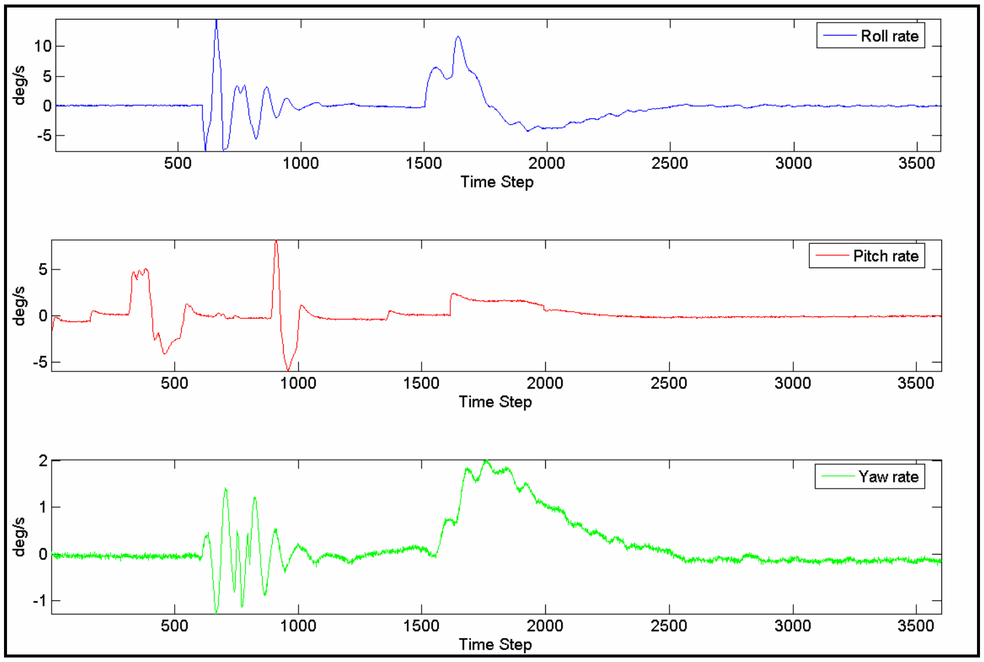


Figure 6-61 : Aircraft Angular Rates, for Failure on Left Aileron (5 deg, 50% eff.)

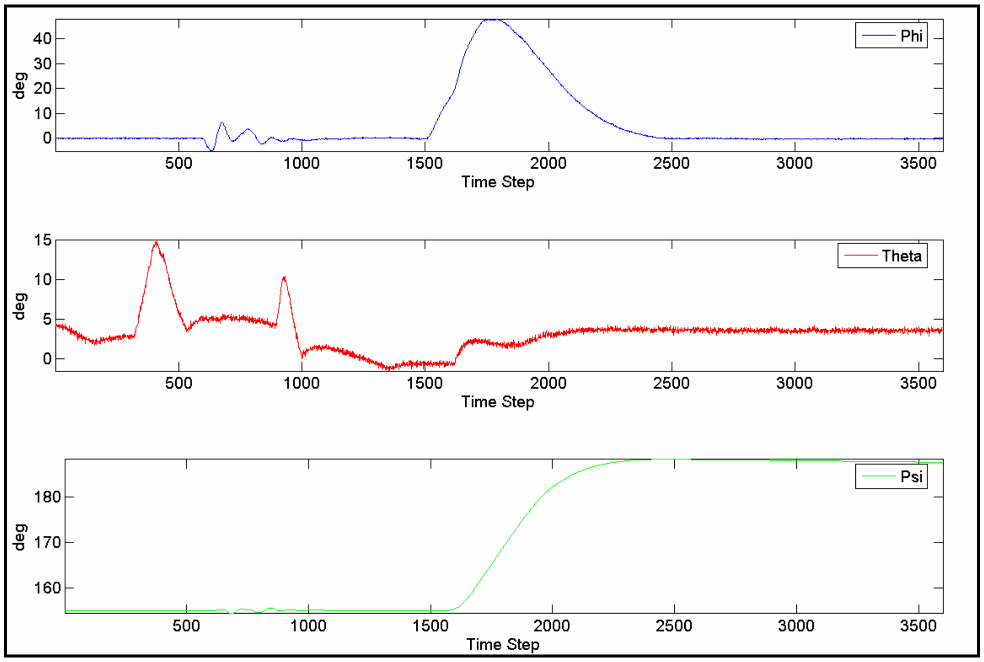


Figure 6-62 : Euler Angles, for Failure on Left Aileron (5 deg, 50% eff.)

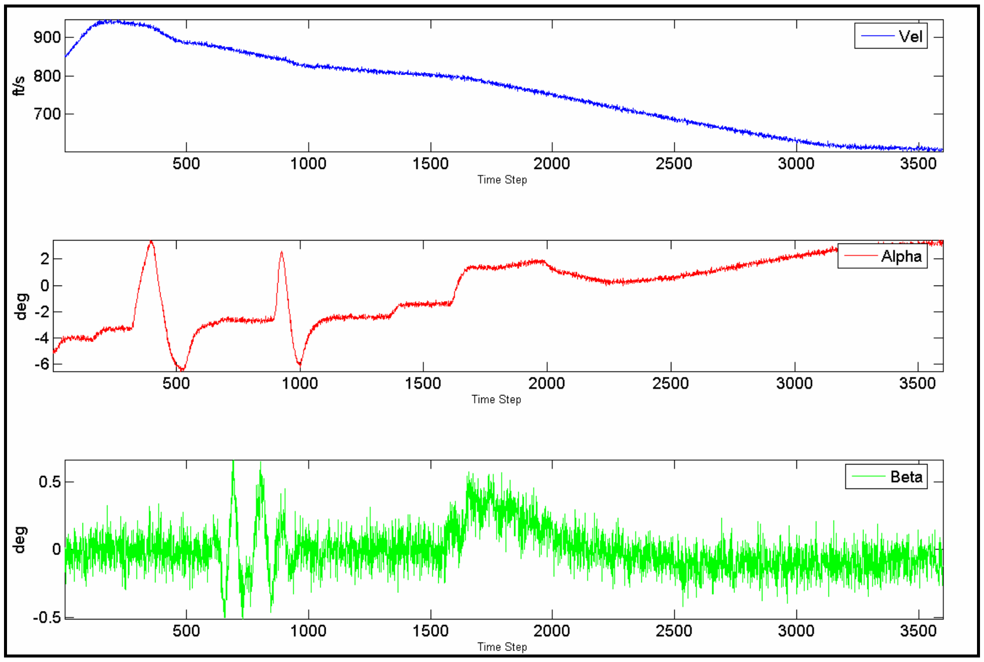


Figure 6-63 : Velocity, Alpha and Beta, for Failure on Left Aileron (5 deg, 50% eff.)

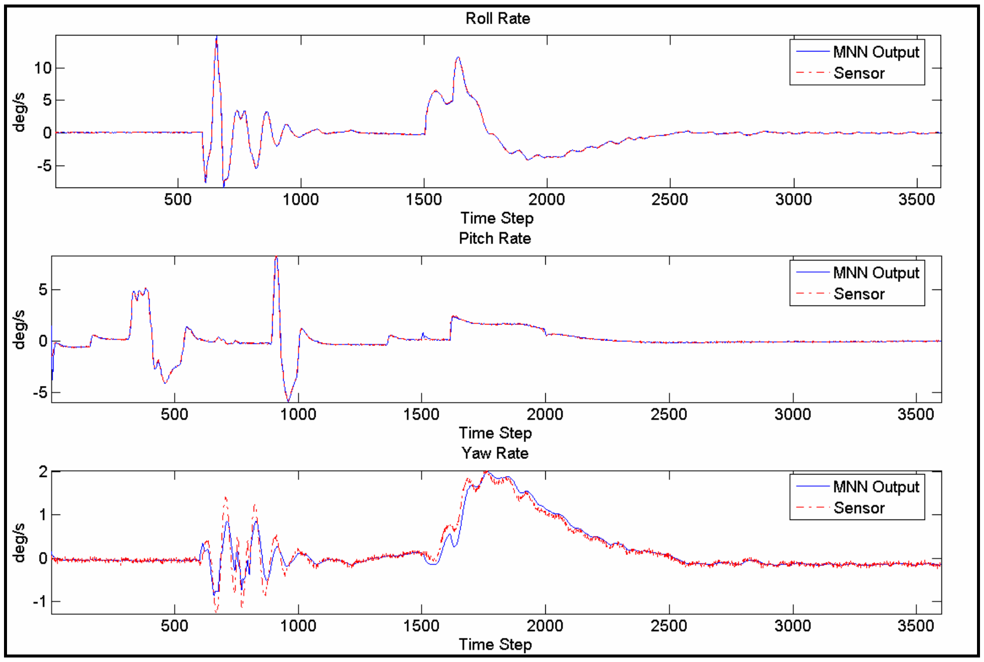


Figure 6-64 : MNN Outputs, for Failure on Left Aileron (5 deg, 50% eff.)

The following figures 6-63 to 6-68 represent the results of Actuator Failure detection, Identification and Accommodation on the Left Aileron for a 0 deg failure with 50% reduction in the efficiency of the failed surface. The error threshold on the MNN is exceeded at 50.078s, indicating a failure on the sensor subsystems. Although the error threshold on the MNN exceeds its threshold, the failure is not identified in this case as the auto correlation coefficient R_{ff} stays within its threshold. This could be the result of the nature of the failure on the aileron, with the aileron stuck at 0 degrees and with only a 50% loss in efficiency.

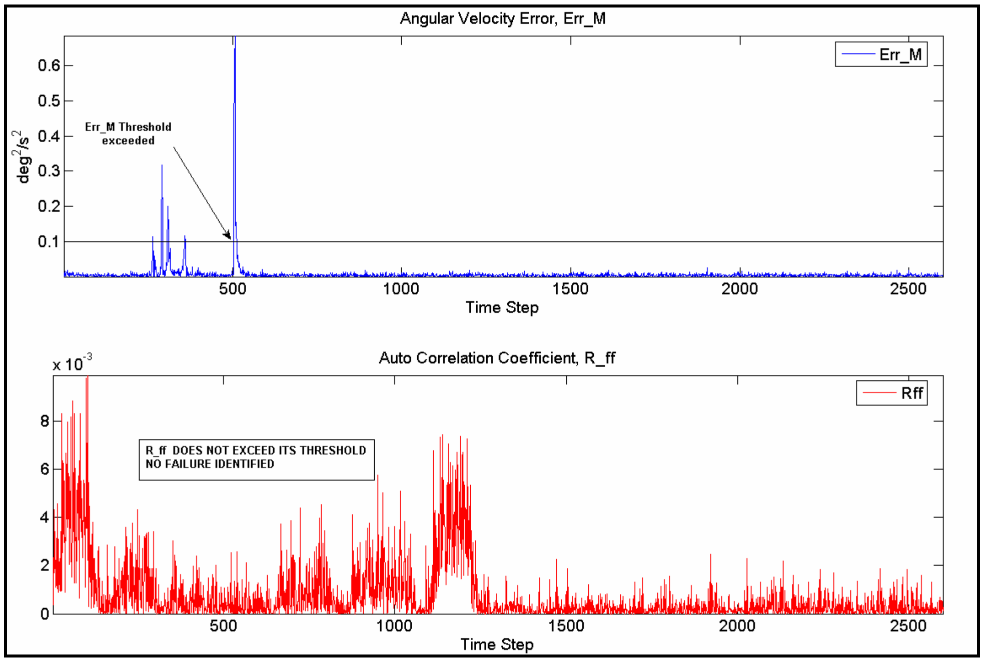


Figure 6-65 : Error Thresholds, for Failure on Left Aileron (0 deg, 50 % eff.)

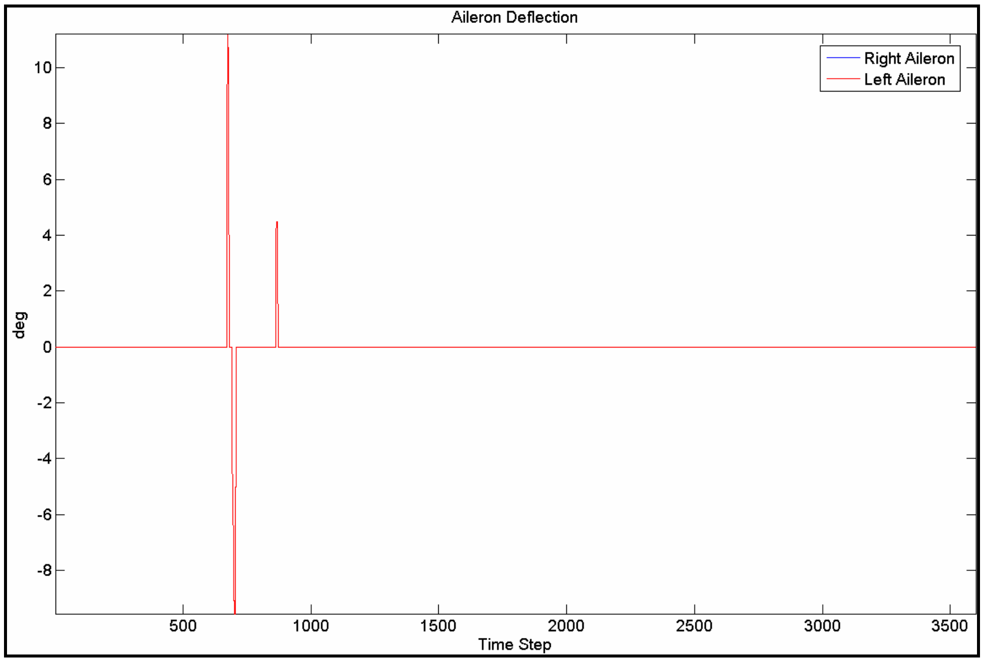


Figure 6-66 : Control Surface Deflection, for Failure on Left Aileron (0 deg, 50 % eff.)

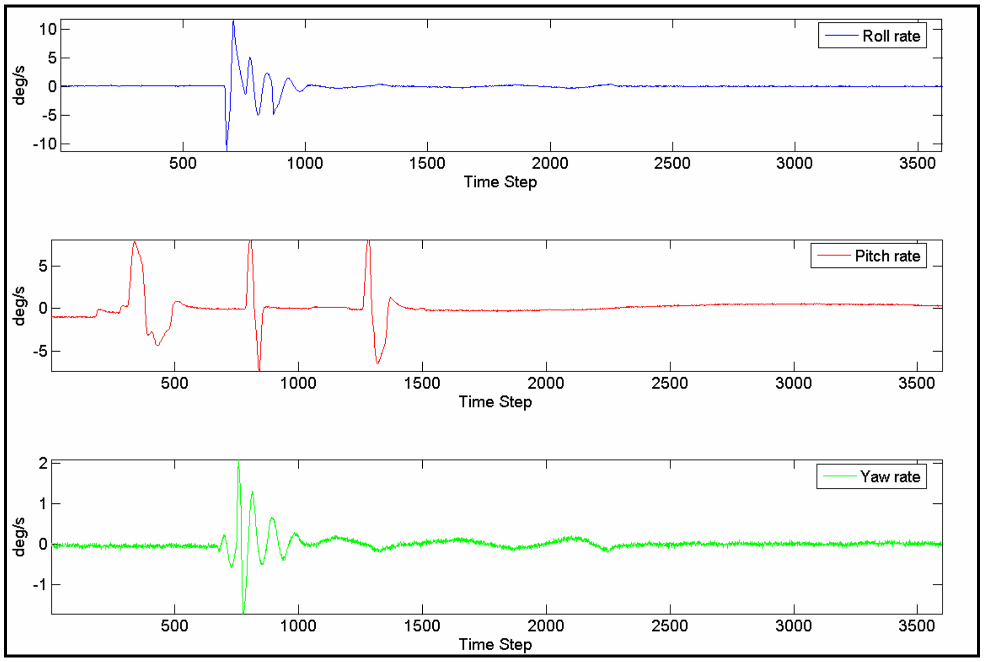


Figure 6-67 : Aircraft Angular rates, for Failure on Left Aileron (0 deg, 50 % eff.)

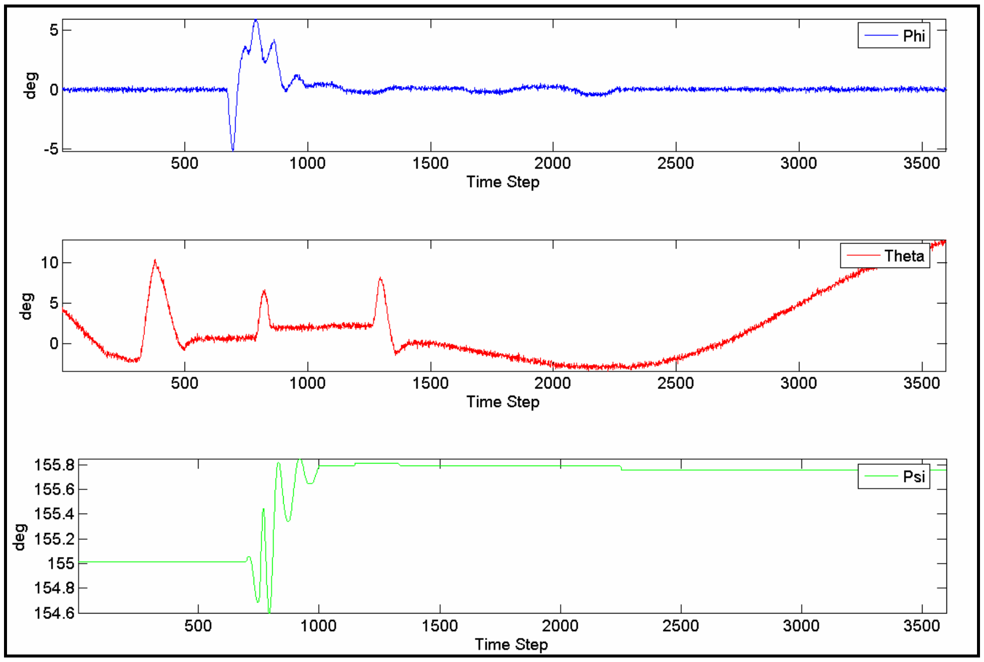


Figure 6-68 : Euler Angles, for Failure on Left Aileron (0 deg, 50 % eff.)

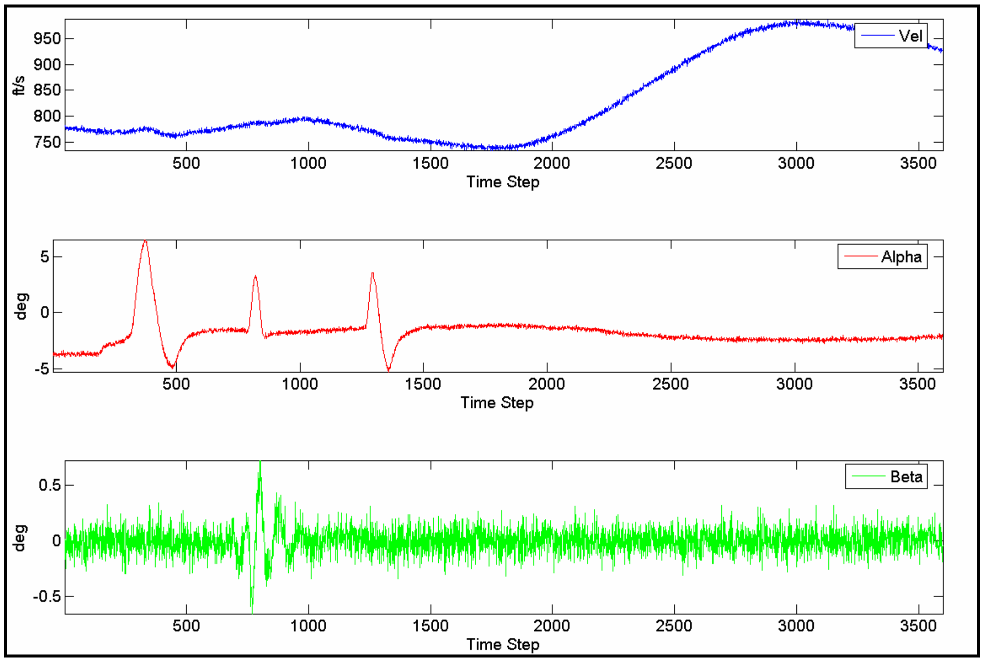


Figure 6-69 : Velocity, Alpha and Beta, for Failure on Left Aileron (0 deg, 50 % eff.)

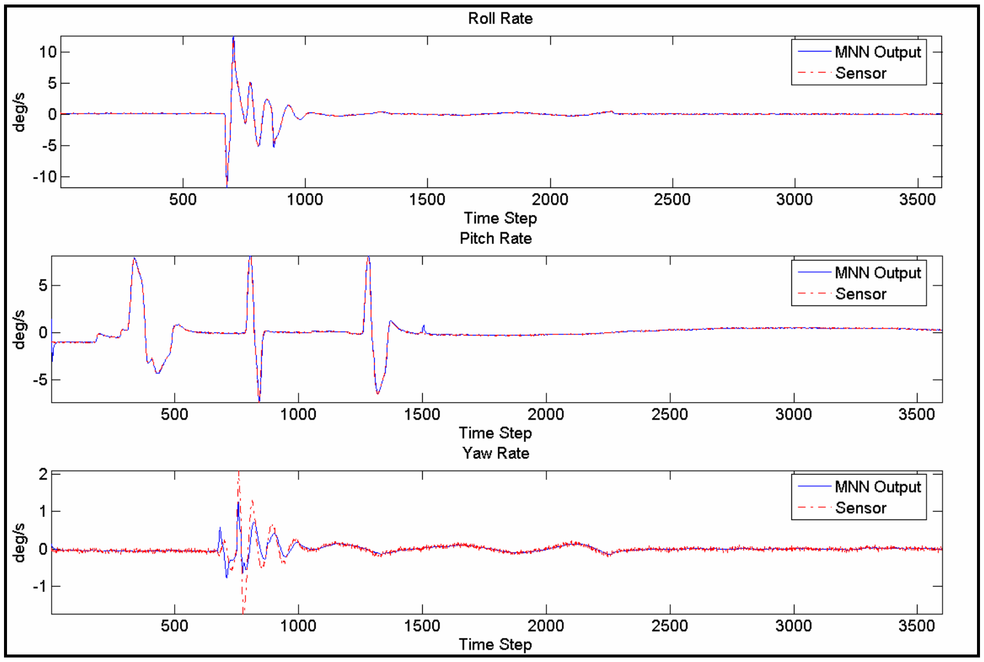


Figure 6-70 : MNN Outputs, for Failure on Left Aileron (0 deg, 50 % eff.)

Chapter 7 - Conclusions and Recommendations

This research effort addresses the design and development of an adaptive flight control scheme to recover an aircraft from failures to its sensor systems and perform the sensor failure detection, identification and accommodation (SFDIA) and its actuators and perform the actuator failure detection, identification and accommodation (AFDIA). The fault tolerant flight control schemes are based on a set of neural network structures named as the Main Neural Network, and Decentralized Neural Network and Neural Network Controller. For the purpose of SFDIA, a combination of the MNN and three DNNs called the P-DNN, Q-DNN and the R-DNN are used. The P, Q, and R DNNs are designed to estimate the outputs of the roll, pitch and yaw gyros at time instant ‘k’, based on inputs from sensor systems onboard the aircraft from time instant ‘k-1’ till ‘k-l’. The restriction on the DNNs is that the P-DNN cannot have the roll rate as its input, the Q-DNN cannot have the pitch rate as its input and the R-DNN cannot have the yaw rate as its input. This is done primarily to ensure that the outputs from these networks are not tainted by any failures on the rate gyros and thus can be relied upon to provide good estimates of the rates in case of failures on the gyros. The MNN generates the estimates of the three gyros at time ‘k’ based on inputs from the sensor system from time ‘k-1’ to ‘k-l’. The MNN does not have any restrictions on the use of the rate gyro measurements to estimate the current values. A quadratic error parameter, MQEE which is the sum of the squares of the error in the rate estimates from the MNN and the DNNs is used for the detection of failures on the gyros. Step type failures of two different magnitudes were introduced on the rate gyros to evaluate the sensor failure detection, identification and accommodation scheme. Upon the introduction of the failures on the sensor outputs, the error metric MQEE showed a sudden spike indicating a failure on one of the sensor systems, while the DQEE parameter, which is the square of the estimation error on each of the individual DNNs provided the failure identification. Step type failures of 4 deg/s and 8 deg/s were successfully injected on the sensors, and subsequently detected and identified successfully on the distributed computational platform.

Although accommodating for sensor failures are important in flight control systems, it takes a lower priority when compared to the accommodation for failures to the actuator systems onboard the aircraft. On the other hand, if the outputs from the rate gyros without physical redundancy are used in closed loop flight control laws, identifying and accommodating for failures on the gyros becomes as important as accommodating for on-board actuator failures. Within the scope of this research study, actuator failures refer to failures to the control surfaces on board the aircraft, especially the left elevator and the left aileron. Failures refer to the situations in which these control surfaces are locked at a given position, with or without the added effect of a loss of surface. Such failure conditions introduce a significant coupling of lateral and longitudinal directional dynamics on the aircraft. The AFDIA scheme is designed along the lines of the SFDIA scheme, with the MNN and three Neural Network Controllers, namely the Roll-NNC, Pitch-NNC and the Yaw-NNC. As with the SFDIA scheme, the MNN is designed to estimate the values of the pitch, roll and yaw rate gyros at time ‘k’, based on inputs from time ‘k-1’ till ‘k-l’. the NNCs on the other hand provide the estimates of the control surface deflections at time ‘k’, based on inputs from the sensor systems from time ‘k-1’ till ‘k-l’. Once failure is detected and identified, the controller takes over the “flying” of the aircraft and restores it to equilibrium conditions.

Within the goals of this research effort was the design of an interface between the neural network based fault tolerant flight control schemes and a simulation/visualization environment. AVDS was chosen as the simulation/visualization environment mainly due to previous experience in using the commercially available software package among members of Dr. Marcello Napolitano’s research group and the fact that AVDS allows the user to customize its code to suit their needs. Towards achieving the goal of interfacing this environment with the fault tolerant flight control schemes, a TCP/IP socket based communication and data transfer scheme was designed and successfully implemented. The scheme is based on the client-server model, with the AVDS environment acting as the client side socket.

Another important phase of this research effort was the successful design and development of a Matlab based GUI tool that would enable the user to interactively set the parameters to define the simulation tasks. This GUI tool provides the user with flexibility to change the structure of the NNs that are used within the fault tolerant flight control schemes. With the help of this tool, the user can change the inputs to the NN structure, the time history window of the inputs, the number of hidden layer neurons, and the learning and momentum rates for the interconnection weights between the nodes of the fully connected NN and the thresholds at each node. This tool also deploys a client side TCP/IP socket that is later used to transfer the simulation parameters to the master node on the cluster.

The typical implementations of flight control schemes are on standalone computational platforms. One of the goals of this research effort was the development of a distributed computing platform that would enable the fault tolerant flight control schemes to be executed in parallel. This goal was successfully realized and a Beowulf cluster was built with 7 compute nodes, *nodes1-7* from COTS components. All the compute nodes on the cluster were running the Linux operating system, kernel version 2.4.20-8 and the cluster communication was based on the Message Passing Interface. For all the tasks, source code was written in C and was structured in such a way that different sections of the code were conditionally executed, based on the rank of the process executing the code, loosely following a SIMD model. The node executing the code for MNN is designated as the master node and is responsible for coordinating the data communication between the AVDS environment and all the other compute nodes on the cluster.

The master node coordinates communication between the cluster and Matlab GUI on one hand and cluster and AVDS on the other hand, using sockets. Once the tasks are initialized on the cluster, a section of code within the master node creates a server side socket and waits for the Matlab GUI to initialize itself and create a client side socket. There is exchange of initialization flags between the master node and the Matlab GUI, followed by the transfer of all the simulation parameters that were set using the GUI, to the master node. The master node now has all the parameter that defines the four network structures MNN, PDNN, QDNN and RDNN and other simulation and task related parameters, such as the time of simulation, and failure specifications. This node then uses MPI directives to send these parameters to all the compute nodes on the cluster, which then proceed to initialize their respective network structures. When AVDS is initialized, it also receives simulation parameters and once the aircraft reaches 3000 ft in altitude, it starts to send sensor data back to the master node using its client side sockets. The master node then distributes this frame of data to all the compute nodes in the cluster. Once the learning cycle is complete on all the compute nodes, the estimates are sent back to the master node wherein error metrics are computed and failure checking is done. If a failure is detected, appropriate flags are set and sent along with the frame of sensor data from AVDS, during the next time step, to all the compute nodes, which tailor their execution responses based on the status of the failure flags.

The AFDIA scheme was successfully implemented on the distributed computation environment for failures on the Left Elevator and the Left Aileron. The failures on the surfaces included the surface being stuck at 0 deg deflection, 5 deg deflection and 15 degree deflection, with a loss of surface. The AFDIA scheme was able to detect the failures and able to accommodate for them, except in the case of 0 deg failures on the aileron.

7.1 Recommendations for Future Work

This research effort provided a proof of concept implementation of a COTS cluster based distributed computing environment for the purpose of implementation of a fault tolerant flight control scheme. Under the current setup, the compute nodes are made up of desktop computers, running a non real-time version of the Linux operating system. This entire setup can be shrunk into a more compact physical size by migrating the cluster to PC104 based compute nodes. Another suggestion would be to make use of real-time extensions to the Linux operating system, making the entire operation run in real time. With increasing emphasis towards flight tests of fault tolerant flight control schemes, the logical step forward from this research effort would be the flight tests on UAV test platforms. Within the fault tolerant flight control scheme, future work could include the integration of the SFDIA and the AFDIA tasks to detect, identify and accommodate for failures on both the sensor systems as well as the actuators. Also, the interactive GUI in Matlab could be expanded to allow the user to choose the learning algorithm of the neural networks, and possibly even the NN architecture itself.

References

[1] An, Y., “A Design of Fault Tolerant Flight Control Systems for Sensor and Actuator Failures Using On-Line Learning Neural Networks”, PhD Dissertation, WVU, Mechanical and Aerospace Dept., Morgantown, WV, 1998.

[2] Roskam, Jan, “Airplane Flight Dynamics and Automatic Flight Controls, Part I”, Design, Analysis and Research Corporation, Lawrence, Kansas, 1995

[3] Stevens, B.L., Lewis, F.L., “Aircraft Control and Simulation”, John Wiley and Sons, New York, 1992.

[4] Haykin, S., “Neural Networks: A comprehensive Foundation”, McMillan College Publishing Company, Inc., 1994.

[5] Motyka, P., Bonnice, W., Hall, S., Wagner, E. “The Evaluation of Failure Detection and Isolation Algorithms for Restructurable Control”, NASA Contractor Report 177983, 1985

[6] AVDS User Manual, Version 1.2.1, Artificial Horizons Inc. October 1999.

[7] Rumelhart, D. and McClelland, J., “Parallel Distributed Processing”, MIT Press, Cambridge MA. 1986.

[9] Alfonso C. Paris, “Estimation of the Longitudinal and Lateral-Directional Aerodynamic Parameters from Flight Data for the NASA F/A-18 HARV”, PhD Dissertation, Department of Mechanical and Aerospace Engineering, West Virginia University, 1997.

[10] Napolitano, M.R., Dale A. Windon II et. al., “Virtual Flight Data Recorder: A Neural Extension of Existing Capabilities” Journal of Guidance, Control, and Dynamics, Vol. 21, No. 4, pp.662~664, 1998.

[11] M. Hagiwara, “Theoretical Derivation of Momentum Term in Back-propagation”, In proceedings of the International Joint Conference on Neural Networks, Vol.I Baltimore, MD, 1992.

[13] Matlab User’s Guide. Mathworks Inc. Natick, MA.

[14] Douglas E. Comer, Internetworking with TCP/IP Vol.1: Principles, Protocols and Architecture (4th Edition).

[15] Kerr, T., “False Alarm and Correct Detection Probabilities Over a Time Interval for Restricted Classes of Failure Detection Algorithms”, IEEE Transactions of Information Theory, Vol. IT-20, No.4, 1982, pp 619~631.

[16] Kerr, T., “Decentralized Filtering and Redundancy Management/Failure Detection for Multisensor Integrated Navigation Systems,” IEEE Transactions on Information Theory, pp.191~208, 1986.

[17] Willsky, A.S., “A Survey of Design Methods for Failure Detection in Dynamic Systems,” Automatica, Vol. 12, pp.601~611, 1976.

[18] Willsky, A.S., “Failure Detection in Dynamic Systems,” Agard LS-109, Neuilly sur Seine, France, pp.2.1~2.14, 1980.

[19] T.-H. Guo, J. Nurre, “Sensor Failure Detection and Recovery by Neural Networks,” Proceedings of the International Joint Conference on Neural Networks, pp. I-221~I-226, 1991.

[20] Petri A. Jokinen, “Comparison of Neural Network Models for Process Fault Detection and Diagnosis Problems,” Proceedings of the International Joint Conference on Neural Networks, pp.I-239~I-244, 1991.

[21] R. Sreedhar, B. Fernandez, C.Y. Masada, “A Neural Network Based Adaptive Fault Detection Scheme,” Proceedings of the American Control Conference, pp.3259~3263, 1995.

[22] Narendra, K.S., Partasarathy, K. “Identification and Control of Dynamical Systems Using Neural Networks”, IEEE Transactions on Neural Networks, Vol. 1, No 1, pp.4~27, 1990.

[23] Levin, A.U., Narendra, K.S. “Control of Non-Linear Dynamical Systems Using Neural Networks: Controllability and Stabilization”, IEEE Transactions on Neural Networks, Vol.4, No.2, pp.192~206, 1993.

[24] M.M. Polycarpou, A.T. Vemuri, “Learning Methodology for Failure Detection and Accommodation,” IEEE Control Systems, pp.16~24, 1995.

[25] M.M. Polycarpou, A.J. Helmicki, “Automated Fault Detection and Accommodation: A Learning Systems Approach,” IEEE Transactions on Systems, Man, and Cybernetics, Vol.25, No.11, pp.1447~1458, 1995.

[26] Kline-Schoder, R.,Rauch, H.,Youssef, “Fault Detection, Isolation, and Reconfiguration for Aircraft Using Neural Networks,” Proceedings of the AIAA Guidance, Navigation and Control Conference, AIAA Paper 93-3876, Monterey, Ca, 1993.

[27] S.N. Balakrishnan, Victor Biega, “Adaptive-Critic-Based Neural Networks for Aircraft Optimal Control,” Journal of Guidance, Control, and Dynamics, Vol.19, No.4, pp.893~898, 1996.

[28] Huang, C., Tylock, J., Engel, S., Whitson, J., Eilbert, J., “Failure-Accommodating Neural Network Flight Control,” Proceedings of the AIAA Guidance, Navigation and Control Conference, AIAA Paper 92-4394, Hilton Head, SC, 1992.

[29] C.M. Ha, “Neural Networks Approaches to AIAA Aircraft Control Design Challenge,” Journal of Guidance, Control, and Dynamics, Vol. 18, No. 4, pp. 731~739, 1995.

[30] Ha, C.M., Wei, Y.P., Bessolo, J.A., “Reconfigurable Aircraft Flight Control System Via Neural Networks,” Proceedings of the 1992 Aerospace Design Conference, AIAA Paper 92-1075, Irvine, Ca, 1992.

[31] Chi-Yuan Chiang, H.M. Youssef, “Neural Network and Fuzzy Logic Approach to Aircraft Reconfigurable Control Design,” Proceedings of the American Control Conference, pp.3505~3509, 1995.

[32] B.S. Kim, A.J. Calise, “Nonlinear Flight Control Using Neural Networks,” Journal of Guidance, Control, and Dynamics, Vol.20, No.1, pp.26~33, 1997.

[33] Napolitano, M.R., Chen, C.I.,et. al., “Aircraft Failure Detection and Identification Using Neural Networks,” Journal of Guidance, Control, and Dynamics, Vol. 16, No. 6, pp.999~1009, 1993.

[34] Napolitano, M.R., Casdorph, V.,Neppach, C., Naylor, S, “On-line Learning Neural Architectures and Cross-Correlation Analysis for Actuator Failure Detection and Identification,” International Journal of Control, Vol.63, No.3, pp.433~455, 1996.

[35] Napolitano, M.R., Steve Naylor et. al., “On-Line Learning Nonlinear Direct Neurocontrollers for Restructurable Systems,” Journal of Guidance, Control, and Dynamics, Vol. 18, No. 1, pp.170~176, 1995.

[36] Mascarell, J. C., “Design and Comparison of Neural Network and Fuzzy Logic Actuator Failure Schemes for Flight Control System,” Master Thesis, Department of Mechanical and Aerospace Engineering, West Virginia University, 1996.

[37] Napolitano, M.R., Neppach, C. et. al., "Neural-Network-Based Scheme for Sensor Failure Detection, Identification, and Accommodation," Journal of Guidance, Control, and Dynamics, Vol.18, No.6, pp.1280~1286, 1995.

[38] Windon, D.A., "Design and Comparison of Neural Network and Kalman Predictor Based Sensor Validation Schemes for Implementation on the NASA-Aurora Theseus Aircraft," Master Thesis, Department of Mechanical and Aerospace Engineering, West Virginia University, 1996.

[39] Napolitano, M.R., Swaim, R.L., "New Technique for Aircraft Flight Control Reconfiguration," Journal of Guidance, Control, and Dynamics, Vol. 14, No.1, pp.184~190, 1991.

[40] Coetzee, L., "Parallel approaches to training feedforward neural nets". Ph.D. Thesis, University of Pretoria, 1996.

[41] Torrensen, J., "Parallelization of Backpropagation Training for Feed-Forward Neural Networks". Ph.D. Thesis, The Norwegian Institute of Technology, 1996.

[42] L. Fausett, "Fundamentals of Neural Networks: Architectures, Algorithms, and Applications", Prentice Hall, Englewood Cliffs, NJ, 1994.

[43] Chen, C.L., Nutter, R.S., "An Extended Back-Propagation Learning by Using Heterogeneous Processing Units," Proceedings of International Joint Conference on Neural Networks, pp. III-988~993, Baltimore, Maryland, 1992.

[44] Flynn, M., "Very high speed computing systems". Proceedings of the IEEE 54:1901-1909, December 1966.

[45] Marc Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., "MPI: The Complete Reference", MIT press, 1999.

[46] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., "PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.

[47] Pacheco, P.S., "Parallel Programming with MPI", Morgan Kaufman Publishers, Inc., 1997.

[48] Crossbow Technology Inc., "Acceleration, Tilt/Angle and Inertial/Gyro", Product Catalog.

[49] Napolitano, M.R., An,Y., Seanor, B., “A fault tolerant flight control system for sensor and actuator failures using neural networks”, Aircraft Design Journal, Vol.3, pp. 103-128.

[50] Gu, Yu, “Design And Flight Testing Actuator Failure Accommodation Controllers on WVU YF-22 Research UAVs”. PhD thesis, West Virginia University, 2004.

[51] Perhinschi, M.G, Lando, M, Massotti, L., Campa, G., Napolitano, M.R., Fravolini, M.L., “On-Line Parameter Estimation Issues for the NASA IFCS F-15Fault Tolerant Systems”.

[52] Brinkre, J., & Wise, K., “Flight testing of a reconfigurable flight control law on the X-36 tailless fighter aircraft”, AIAA GNC Conference, 2000.

[53] Air Force Research Laboratory, “First Flight Test Demonstration of Neural Network Software”, http://www.afrlhorizons.com/Briefs/0001/VA9904.html

[54] Dryden Flight Research Center, “X-36 Tailless Fighter Agility Research Aircraft in flight”,http://www.dfrc.nasa.gov/Gallery/Photo/X-36

[55] NASA Dryden Flight Research Center, “Intelligent Flight Control Systems”, http://www.dfrc.nasa.gov/Newsroom/FactSheets/FS-076-DFRC.html

[56] Perhinschi M. G., Napolitano M.R., Campa G., Seanor B., Gururajan S., "Design of Intelligent Flight Control Laws for the WVU F-22 Model Aircraft", *Proceedings of the AIAA Intelligent Systems Technical Conference* 2004 Chicago IL, AIAA2004-6282

[57] Perhinschi M. G., Napolitano M.R., Campa G., Burke H. E., Larson R. R., Burken J., Fravolini M. L., "Design and Testing of a Safety Monitor Scheme on the NASA Gen_2 IFCS F-15 Flight Simulator", *Proceedings of the AIAA Intelligent Systems Technical Conference* 2004 Chicago IL, AIAA2004-6284

[58] Perhinschi M. G., Burken J., Napolitano M.R., Campa G., Fravolini M. L.,"Performance Comparison of Different Neural Augmentation for the NASA Gen_2 IFCS F-15 Control Laws", *Proceedings of the American Control Conference 2004* Boston MA, pp3180-3184

[59] Perhinschi M. G., Napolitano M.R., Campa G., Fravolini M. L., "Integration of Fault Tolerant System for Sensor and Actuator Failures within the WVU NASA F-15 Simulator", *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2003, Austin, Texas, AIAA-2003-5643

[60] Perhinschi M. G., Napolitano M.R., Stolarik B., Hammaker S., Campa G., Rogers S., "Design Of Safety Monitor Schemes for a Fault Tolerant Flight Control System",

Proceedings of the AIAA Guidance, Navigation, and Control Conference, August 2003, Austin, Texas, AIAA-2003-5646

[61] Battipede M., Gili P., Lando M, Napolitano M. R., Perhinschi M. G., Campa G., "Comparative Analysis of Neural Control Systems Within the NASA IFCS F-15 WVU Simulator", *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2003, Austin, Texas, AIAA-2003-5643

[62] Perhinschi M. G., Napolitano M.R., Campa G., Fravolini M. L., Massotti L.,Lando M., "Augmentation of a Non Linear Dynamic Inversion Scheme Within the NASA IFCS F-15 WVU Simulator", *Proceedings of* the *American Control Conference 2003*, June 4-6, 2003 Denver CO, USA, pp1667-1672

[63] Perhinschi M. G., Campa G., Napolitano M.R., Fravolini M. L., Lando M., Massotti L., "Performance Comparison of Fault Tolerant Control Laws Within the NASA IFCS F-15 WVU Simulator", *Proceedings of the American Control Conference 2003*, June 4-6, 2003 Denver CO, USA, pp1661-1666

[64] Battipede M., Gili P., Napolitano M. R., Perhinschi M. G., Massotti L., Lando M., "Implementation of an Adaptive Predictor-Corrector Neural Controller within the NASA IFCS F-15 WVU Simulator", Proceedings of the *American Control Conference 2003*, June 4-6, 2003 Denver CO, USA, pp1302-1307

[65] Perhinschi M. G., Lando M., Massotti L., Fravolini M. L., Campa G., Napolitano M.R., “On-Line Parameter Estimation for Real Time Application for the NASA IFCS F-15 Fault Tolerant Systems”, *Proceedings of the American Control Conference*, Anchorage, AK, 2002, pp191-196

Appendix A

A.1 Sample Three Dimensional Lookup Table

This section shows a sample of the three dimensional lookup tables for the F4 aircraft dynamics.

```
#
#      Aero Coefficients for F4 aircraft dynamics
#

# Cdo : basic drag coefficient
#
:DB 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.0269 0.0269 0.0205 0.0439 0.0439

#
# Cdu : drag coefficient due to translational velocity
#
:Du 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.0    0.0    0.027  -0.054 -0.054


#
# Cda : drag coefficient due to angle of attack
#
:Da 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.555  0.555  0.300  0.400  0.400

#
# Cdf : drag coefficient due to flap deflection
#
:Df 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.0    0.0    0.0    0.0    0.0

#
# Cdg : drag coefficient due to gear deflection
#
:Dg 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.0    0.0    0.0    0.0    0.0

#
# Clo : basic lift coefficient
#
:LB 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.43   0.43   0.100  0.01   0.01
```

```
#
# Cla : lift coefficient due to angle of attack
#
:La 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       2.80   2.80   3.75   2.80   2.80

#
# Clde : lift coefficient due to elevator deflection
#
:Le 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.24   0.24   0.40   0.25   0.25

#
# Cldf : lift coefficient due to flap deflection
#
:Lf 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       0.0    0.0    0.0    0.0    0.0

#
# Clq : lift coefficient with dimensionless pitch rate
#
:Lq 5 0 0 Mach None None 1.0 0.0
       0.0    0.206  0.900  1.800  9.00
       1.33   1.33   1.80   1.30   1.30
```

A.2 User Defined Simulation Code Block Header Information

- *Blockname* - the name of the user code block. This is entered in the usr_cblock.c file to notify AVDS of this new code block.
- *Type* - the value in this location is either USR_FCS or USR_AC. The type of the block determines where it is displayed on the Aircraft Initialization page.
- *Description of code block* - a textual description of the code block. This description is displayed in the type menu for the aircraft of FCS on the Aircraft Initialization.
- *in1 in2 in3 in4 in5 in6 ...* - a list of the inputs to this block. These are displayed under FCS In or AC In on the Aircraft Initialization page.
- *out1 out2 out3 ...* - a list of the outputs from this block. These are displayed under FCS Out or AC Out on the Aircraft Initialization page.
- *param1 param2 param3 param4 ...* - a list of the parameters that will have corresponding look-up tables.
- *r* - the update rate in Hz is based on this integer number. The actual update may vary depending on the computer and complexity of the simulation model.
- *Init_func* - the name of a function that AVDS will call to initialize the code block.
- *update_func* - the name of the function that AVDS calls at the rate specified by "r" to update the simulation.

```
CBlock blockname = {
       type,                                /* Type */
       "description of code block" ,     /* Description */
       "in1 in2 in3 in4 in5 in6...",     /* Inputs */
       "out1 out2 out3...",              /* Outputs */
       "param1 param2 param3 param4...",/* Params*/
       r,                                /* Update Rate (Hz) */
       init_func                         /* Init Function*/
       update_func                       /* Update Function */
}
```

Appendix B

B.1 Sample User Defined Simulation Code Block.

```
#include <usr_cblock.h>
int nBlocks = 1;
CBlock FBgainFCS;
Cblock *pCBlockx[] = {&FBgainFCS};
void fcs_init();
void fcs_update();
/* Add declaration of code blocks */
CBlock FBgainFCS = {
       USR_FCS,                         /* Type */
       "Feedback Gains",                /* Description */
       "Ps Rs Ys Ts P Q R",             /* Inputs */
       "de da dr dt",                   /* Outputs */
       "k1 k2 k3 cmd1 cmd2 cmd3",       /* Parameters */
       20,                              /* Update Rate (Hz) */
       fcs_init,                        /* Init Function */
       fcs_update                       /* Update Function */
       };
enum { STICK_P, STICK_R, STICK_Y, STICK_T, ROLL, PITCH, YAW }; /* Inputs */
enum { D_ELE, D_AIL, D_RUD, D_THR }; /* Outputs */
enum { GAIN_P, GAIN_R, GAIN_Y, GAIN_CP, GAIN_CR, GAIN_CY}; /* Parameters
*/
void
fcs_init( input, output, param )
double *input, *output, *param;
{
       output[D_ELE] = input[STICK_P]*param[GAIN_CP]+input[PITCH]*param[GAIN_P];
       output[D_AIL] = input[STICK_R]*param[GAIN_CR]+input[ROLL]*param[GAIN_R];
       output[D_RUD] = input[STICK_Y]*param[GAIN_CY]+input[YAW]*param[GAIN_Y];
       output[D_THR] = input[STICK_T];
}
void
fcs_update( input, output, param )
double *input, *output, *param;
{
       output[D_ELE] = input[STICK_P]*param[GAIN_CP]+input[PITCH]*param[GAIN_P];
       output[D_AIL] = input[STICK_R]*param[GAIN_CR]+input[ROLL]*param[GAIN_R];
       output[D_RUD] = input[STICK_Y]*param[GAIN_CY]+input[YAW]*param[GAIN_Y];
       output[D_THR] = input[STICK_T];
}
```