Louisiana Tech University
Louisiana Tech Digital Commons

Doctoral Dissertations | Graduate School

Spring 2008

Wireless sensor network modeling using modified recurrent neural network: Application to fault detection

Azzam Issam Moustapha
Louisiana Tech University

Follow this and additional works at: https://digitalcommons.latech.edu/dissertations

 Part of the Artificial Intelligence and Robotics Commons, and the Electrical and Computer Engineering Commons

Recommended Citation
Moustapha, Azzam Issam, "" (2008). *Dissertation*. 513.
https://digitalcommons.latech.edu/dissertations/513

This Dissertation is brought to you for free and open access by the Graduate School at Louisiana Tech Digital Commons. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of Louisiana Tech Digital Commons. For more information, please contact digitalcommons@latech.edu.

WIRELESS SENSOR NETWORK MODELING USING MODIFIED RECURRENT NEURAL NETWORK: APPLICATION TO FAULT DETECTION

by

AZZAM ISSAM MOUSTAPHA, BE, MEE, Ph.D.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
of Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

May 2008

UMI Number: 3308059

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3308059

Copyright 2008 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

4/ 2 / 2008
Date

We hereby recommend that the dissertation prepared under our supervision by AZZAM ISSAM MOUSTAPHA entitled WIRELESS SENSOR NETWORK MODELING USING MODIFIED RECURRENT NEURAL NETWORKS : APPLICATION TO FAULT DETECTION be accepted in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY

Supervisor of Dissertation Research

Head of Department

Electrical Engineering
Department

Recommendation concurred in:

Advisory Committee

Approved:
Director of Graduate Studies

Approved:
Dean of the Graduate School

Dean of the College

GS Form 13a
(6/07)

ABSTRACT

Wireless Sensor Networks (WSNs) consist of a large number of sensors, which in turn have their own dynamics. They interact with each other and the base station, which controls the network. In multi-hop wireless sensor networks, information hops from one node to another and finally to the network gateway or base station. Dynamic Recurrent Neural Networks (RNNs) consist of a set of dynamic nodes that provide internal feedback to their own inputs. They can be used to simulate and model dynamic systems such as a network of sensors.

In this dissertation, a dynamic model of wireless sensor networks and its application to sensor node fault detection are presented. RNNs are used to model a sensor node, the node's dynamics, and the interconnections with other sensor network nodes. A neural network modeling approach is used for sensor node identification and fault detection in WSNs. The input to the neural network is chosen to include previous output samples of the modeling sensor node and the current and previous output samples of neighboring sensors. The model is based on a new structure of a backpropagation-type neural network. The input to the neural network (NN) and the topology of the network are based on a general nonlinear sensor model. A simulation example, including a comparison to the Kalman filter method, has demonstrated the effectiveness of the proposed scheme. The simulation with comparison to the Kalman filtering technique was carried out on a network with 15 sensor nodes. A fault such as drift was introduced and

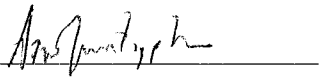
successfully detected with the modified recurrent neural net model with no early false alarm that could have resulted when using the Kalman filtering approach.

In this dissertation, we also present the real-time implementation of a neural network-based fault detection for WSNs. The method is implemented on a TinyOS operating system. A collection tree network is formed, and multi-hoping data is sent to the base station root. Nodes take environmental measurements every N seconds while neighboring nodes overhear the measurement as it is being forwarded to the base station for recording it. After nodes complete M and receive/store M measurements from each neighboring node, recurrent neural networks are used to model the sensor node, the node's dynamics, and the interconnections with neighboring nodes. The physical measurement is compared to the predicted value and to a given threshold of error to determine a sensor fault. The process of neural network training can be repeated indefinitely to maintain self-aware network fault detection. By simply overhearing network traffic, this implementation uses no extra bandwidth or radio broadcast power. The only costs of the approach are the battery power required to power the receiver for overhearing packets and the processor time to train the RNN.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Dissertation. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Dissertation. Further, any portions of the Dissertation used in books, papers, and other works must be appropriately referenced to this Dissertation.

Finally, the author of this Dissertation reserves the right to publish freely, in the literature, at any time, any or all portions of this Dissertation.

Author 

Date April 21 2008

Dedication

To my Dear Parents

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express great appreciation to Dr. Rastko R. Selmic for his knowledge transfer, advice, support, inspiring work, and encouragement. I also appreciate the opportunity of having been allowed to work as a teaching and research assistant under his supervision. I attribute most of my research career advancement to his insightful guidance and suggestions. I am deeply thankful for his wise guidance, encouragement, and support throughout my research work. I owe him an immense debt of gratitude for inspiring my love, passion, and curiosity for combining the sensors' fault tolerance field and neural network.

I would like to thank Dr. Lihe Zou, Dr. Vir V. Phoha, Dr. Sandra Selmic, and Dr. Hisham Hegab, for serving on my advisory committee and reviewing my dissertation. In addition, I wish to thank all of the friends in Ruston, Louisiana, and elsewhere in the U.S.A who helped me.

Finally, I thank my dear parents, for their love, patience, and encouragement has always accompanied me during my academic pursuits.

This research was supported by the following grants: NSF/LEQSF(2005)-PFUND-19, LaSPACE NASA Grant #822, NSF EPSCoR 32-0967-58159, and PKSFI LEQSF (2007-12)-ENH-PKSFI-PRS-03.

CHAPTER 1

MOTIVATION

A sensor is a dynamic system; thus, in order to understand the sensor performance, we need to understand its dynamics and the way they interact with the external physical world, other sensors, the environment, and humans. Due to the lower cost and development of networking technology, sensors are increseangly being networked in wired and wireless sensor networks.

Wireless Sensor Networks (WSNs) consist of a set of sensor nodes that can communicate with each other, sensors that measure a desired physical quantity, and the system base station for data collection, processing, and connection to the wide area network. Modern wireless sensor nodes have microprocessors for local data processing, networking, and control purposes [1]. WSNs have enabled numerous advanced monitoring and control applications in environmental, biomedical, and other applications.

Sensors in such networks have their own dynamics, often nonlinear, and modeling such a sensor network is often not trivial. Since Recurrent Neural Networks (RNNs) consist of interconnected dynamic nodes, we explore their similarities with WSNs and exploit those similarities in the WSN modeling. This dissertation presents modeling of WSNs using a modified dynamic RNN.

The real motivation for WSN modeling stems from the need for intelligent fault detection in complex distributed sensory systems. Since sensor networks often operate in

potentially hostile and harsh environments, most applications are mission critical. Sensors are often used to compute control actions [2-4] where sensor faults can cause catastrophic events. For instance, NASA was forced to abort the launch of space shuttle Discovery due to a failure of one of the sensors in the sensor network of the shuttle's external tank (failure was discovered by human inspection).

Components such as sensors and actuators have significantly higher fault rates than the traditional integrated semiconductor circuit-based systems. Multi-sensor systems need feedback information about the health status of their nodes in order to recover and heal from eventual faults. Such a system would have improved reliability over existing sensor networks. Since external and internal malfunctions or excessive noise can occur, sensor readings are somewhat uncertain in the sense that no existing sensor will deliver accurate readings at all times. It is therefore desirable to develop a WSN that will have the capability of fault detection, isolation, and accommodation. Efficiency in converting data to features while consistently accommodating the uncertainty inherent in the measurements form a key issue for diagnosis and dealing with sensor faults [5] [6]. Fault-tolerance has become essential and urgent for modern sensory systems [7]. The traditional way of achieving fault-tolerance in dynamic systems is through hardware redundancy such as the use of multiple sensors. But the multiplication of sensor devices adds cost, complexity, and power consumption to the sensor node and the whole network. Most of the present research efforts have concentrated on an analytical redundancy [8] [9] in which sensor measurements are processed analytically and mathematical models are compared with physical measurements. Instead of using additional hardware in the form of multiple sensors, we propose to use computational resources for intelligent fault

detection. A dynamic model of a sensor node is formed from information using neighboring nodes in the network. RNNs have been applied to model a network due to their topological similarities with WSNs.

CHAPTER 2

WIRELESS SENSOR NETWORK MODELING USING MODIFIED RECURRENT NEURAL NETWORKS: APPLICATION TO FAULT DETECTION

2.1 Introduction

Instead of using additional hardware in the form of multiple sensors, we propose to use computational resources for intelligent fault detection. A dynamic model of a sensor node is formed and based on information from neighboring nodes in the network. RNN's have been applied to model a network, due to their topological similarities with WSNs. Communication uncertainties are modeled using *confidence factors* based on received signal strength. More detailed communication models can be applied, but this is not the topic of the dissertation.

In addition to neural networks, the identification of a nonlinear dynamic system was studied using some alternative techniques. Gallman and Narendra [10] used an iterative algorithm to obtain the dynamics of the system from finite length input and noisy output data records. This algorithm has shown to converge for a class of inputs, including colored Gaussian processes. Haber [11] discussed a two-step identification method of least-squares parameter estimation based on correlation functions for nonlinear

dynamic systems with linear parameters. Shiavo and Luciano [12] presented a new, powerful, and flexible fuzzy algorithm for nonlinear dynamic system identification

The rest of the chapter is organized as follows. Section 2.2 covers briefly some background on RNNs and their function approximation property. In Section 2.3, a modified RNN and its model of a dynamic sensor node is introduced, including a result that shows how neighboring nodes can be used in sensor node modeling. Section 2.4 describes how such a tool can be used in the sensors' failure detection in distributed sensor networks. Numerical simulations are given in Section 2.5 to show the effectiveness of the proposed modeling scheme. Finally, Section 2.6 is the conclusion.

2.2 Background

Artificial RNNs have the ability to capture and model dynamic properties of nonlinear systems. The RNN nodes have their own dynamics with interconnecting weights between the nodes – similar to the wireless sensor networks where each sensor node has its own dynamics. Recurrent networks also include feedback loops which standard Neural Networks (NNs) do not have [13-15].

We have used a nonlinear output error model [13] given by

$$y(k) = F_{NN}(y(k-1),\ y(k-2),\ \ldots,\ y(k-m), u(k), u(k-1),\ u(k-2),\ \ldots,\ u(k-n)) \quad (2.1)$$

where $y(k)$ is the NN output, $y(k\text{-}i)$ are previous NN outputs, and $u(k\text{-}i)$ are inputs including the previous inputs. The nonlinear function F_{NN} is computed using a feedforward neural net given in matrix form by

$$F_{NN}(x) = W^T \sigma(V^T x) \quad (2.2)$$

where x is the NN input, V is the first-layer weights, W is the second layer weights, and $\sigma(\cdot)$ is the neural net activation function (usually chosen as standard sigmoid function).

The output activation function is chosen as a linear function. The structure of the NN is given in Figure 2.1.

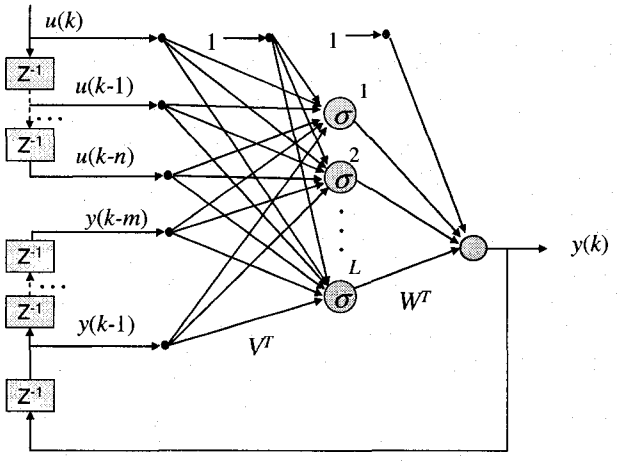


Figure 2.1 Two-layer recurrent neural network

The two-layer NN in Figure 2.1 consists of two layers of tunable weights and thresholds and has a hidden layer and an output layer. The hidden layer has L neurons, and the input layer is a combination of delayed input $u(k)$ and the output $y(k)$.

Many well-known results indicate that any sufficiently smooth function can be approximated arbitrarily close on a compact set using a two-layer NN with appropriate [16] [17] weights. Both layer weights V and W can be tuned. The NN *universal approximation property* says that any continuous function f can be approximated arbitrarily well using a linear combination of sigmoidal functions, namely

$$f(x) = W^T \sigma(V^T x) + \varepsilon(x) \ , \tag{2.3}$$

where the $\varepsilon(x)$ is the NN approximation error. The reconstruction error is bounded on a compact set S by $\|\varepsilon(x)\| < \varepsilon_N$. Moreover, for any ε_N one can find a NN such as $\|\varepsilon(x)\| < \varepsilon_N$ for all $x \in S$.

Given a function $g(x)$ and a domain set $D \subset \Re^n$, the function is said to satisfy Lipschitz condition on set D if

$$\|g(x) - g(y)\| \leq L\|x - y\| , \tag{2.4}$$

for any $x, y \in D$. The function is said to be globally Lipschitz if the above condition is valid on $\Re^n$. If the function g is mapping $\Re \to \Re$, then the condition is equivalent to

$$|g(x) - g(y)| \leq L|x - y| , \tag{2.5}$$

which says that a straight line connecting any two points of $g(x)$ cannot have a slope with an absolute value greater than L. Therefore, any function with an infinite slope at some point is not Lipschitz at that point.

2.3 Modified Recurrent Neural Nets in Sensor Network Modeling

Dynamic RNNs consist of a set of dynamic nodes that provide internal feedback to their own inputs (see Figure 2.1). They can be used to simulate and model dynamic systems such as a network of sensors. WSNs consist of a large number of sensors, which in turn have their own dynamics. They interact with each other and the base station, which controls the network. In multi-hop wireless sensor networks, information hops from one node to another and finally to the network gateway or base station.

To develop a dynamic model for such sensors, without a loss of generality, we assume that there is one sensor per sensor node. More sensors per node will simply increase the size of the RNNs.

Sensor nodes can be viewed as small dynamic systems with memory-like features. Output of one node forwards the information to the next node (for example, node 3

provides the input to node 5, Figure 2.2). While the standard RNN is structured in layers, we introduce an ad-hoc RNN analogous to WSN systems with confidence factors ($0 < C_{ij} < 1$) between the nodes i and j. The confidence factor depends on the signal strength and data quality in communication links between the nodes. For instance, in tuning node 2, valuable inputs are coming from node 1 and node 4, providing that corresponding confidence factors are close to 1. If node 7 is not in the coverage area of node 2, then the confidence factor is 0 and node 7 will not influence node 2 directly.

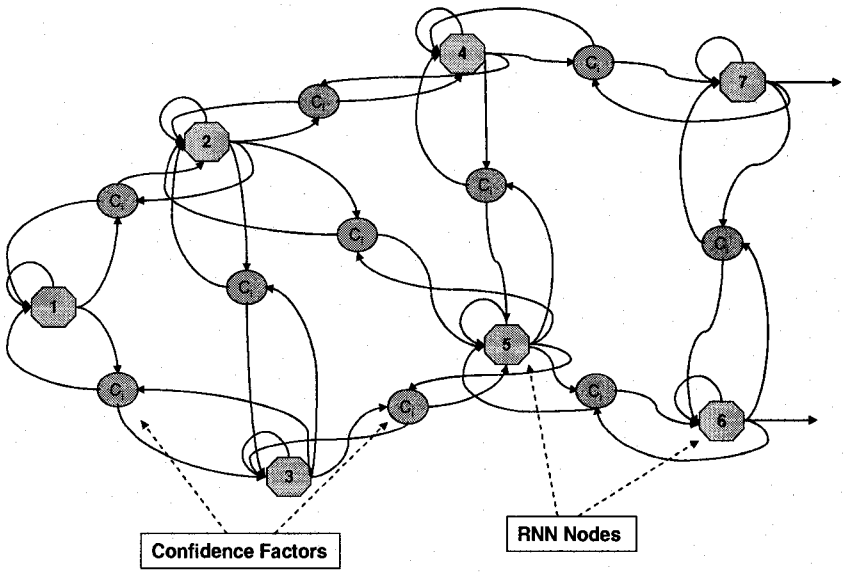


Figure 2.2 Ad hoc recurrent neural network with topology of a wireless sensor network

Note that the confidence factors do not provide stochastic modeling of the communication channel. The overall modeling process can be divided into two phases: the learning phase is where the neural network (NN) adjusts its weights that correspond to the healthy and N faulty models, where N is the number of fault types. The production phase is where the current output of the sensor node is compared with the output of the

NN. The difference between these two signals is used as a measure of a sensor's health status. In case of a fault, NN weights (model) are compared with the faulty models to isolate the fault. If no similar fault model is found, then the fault bank model is updated with the new type of fault and corresponding model parameters, i.e., NN weights. This whole process is repeated during the production phase.

Consider a nonlinear dynamic sensor model given by

$$y_i(k) = f_i(y_i(k-1),\, y_i(k-2),\, \ldots,\, y_i(k-m), u_i(k)) \tag{2.6}$$

where $u_i(k)$, $y_i(k)$ are the sensor input and output at sample k, and f_i s are unknown nonlinear functions. In order for a sensor to be operational and the user to determine the real sensor input, the function f_i has to be invertible.

$$u_i(k) = f_i^{-1}(y_i(k-1),\, y_i(k-2),\, \ldots,\, y_i(k-m),\, y_i(k)) \tag{2.7}$$

Equation 2.7 indicates that in order to determine the physical input at the sample k, knowledge of the present and past m sensor outputs is required. A more general dynamic sensor model is given by [12]

$$y_i(k) = f_i(y_i(k-1),\, y_i(k-2),\, \ldots,\, y_i(k-m), \vec{u}_i(k)) \tag{2.8}$$

where $\vec{u}_i^n(k)$ is a vector of input data $\vec{u}_i^n(k) = [u_i(k), u_i(k-1), \ldots, u_i(k-n)]$. Similarly, in order for the sensor to be usable and users to determine the physical input values based on the sensor outputs, the nonlinear function has to be invertible with respect to input signal arguments

$$\vec{u}_i^n(k) = f_i(y_i(k-1),\, y_i(k-2),\, \ldots,\, y_i(k-m),\, y_i(k)) \ . \tag{2.9}$$

Such sensor models correspond to a general sensor model given in [19], i.e., Hammerstein-Wiener nonlinear feedback dynamic sensor model (Figure 2.3), which

consists of a linear dynamic block surrounded by three nonlinear static blocks [20].

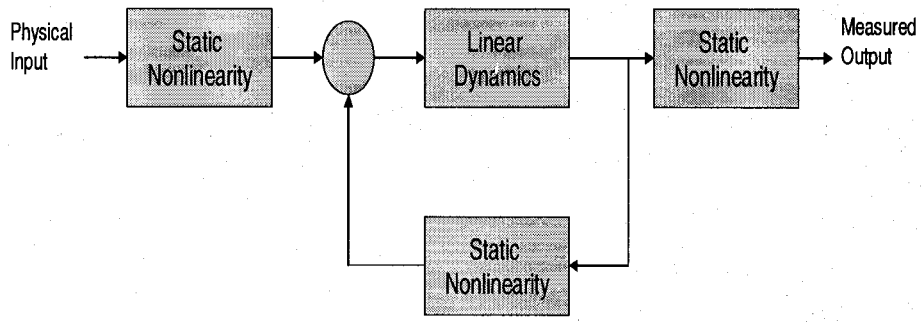


Figure 2.3 A linear dynamic block surrounded by three static nonlinear blocks representing a Hammerstein-Wiener dynamic sensor model

It is assumed that all sensors have models of the same order. If that is not the case, the analysis can still be carried out with slight modification.

Assumption 1: Sensor nodes have a nonlinear model of the same order given by (2.6).

Assumption 2: Functions f_i s are globally Lipschitz functions with L_is being their Lipschitz constant, respectively.

While wireless sensor nodes are distributed in the field, the neighboring nodes are assumed to have bounded differences in measured physical quantity. Mathematically, the assumption is given as follows.

Assumption 3: Neighboring sensor nodes have measurement events that differ by a bounded constant, i.e., for a sensor node's neighbors a and b,

$$u_a(k) - u_b(k) = e_{ab}(k), \tag{2.12}$$

and $\|e_{ab}(k)\| < e$.

The next result shows how to model a wireless sensor network using a recurrent

neural network and how to use such a tool in failure detection of sensor nodes.

***Theorem 1 (Wireless sensor network model using RNNs)*:**

Having a model of a sensor node i (Equation 2.6), assumptions 1-3, and the node neighbors that include nodes i_1, i_2,..., i_{Ni} (see Figure 2.4), the output of the sensor node can be approximated using a RNN with inputs consisting of the previous outputs from node i and its neighboring nodes

$$y_i(k) = RNN_i(y_i(k-1), y_i(k-2), \ldots, y_i(k-m), y_{i_j}(k), y_{i\,j}(k-1), \ldots, y_{i\,j}(k-m)) + c \quad (2.13)$$

where j=1, 2,..., N_i and c is a small bounded constant.

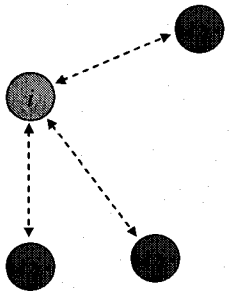


Figure 2.4 Sensor node i and its neighboring sensors i_1, i_2, ..., i_{Ni}.

Proof:

From Assumption 3 it follows that

$$u_i(k) = u_{i_j}(k) + e_{ij}(k), \quad (2.14)$$

where j=1, 2,..., N_i. Equivalently, the input $u_i(k)$ is given by

$$u_i(k) = \frac{1}{N_i}\sum_{j=1}^{N_i} u_{i\,j}(k) + e_{ij}(k). \quad (2.15)$$

Therefore, one has

$$y_i(k) = f_i(y_i(k-1),\ y_i(k-2), \ldots,\ y_i(k-m), \frac{1}{N_i}\sum_{j=1}^{N_i} u_{ij}(k) + e_{ij}(k)). \qquad (2.16)$$

Using expression (2.7) one has

$$y_i(k) = f_i(y_i(k-1), y_i(k-2), \ldots, y_i(k-m), \frac{1}{N_i}\sum_{j=1}^{N_i} f_{i_j}^{-1}(y_{ij}(k-1), y_{ij}(k-2), \ldots, y_{ij}(k-m), y_{i_j}(k)) + e_{ij}(k)). \qquad (2.17)$$

Knowing that the function f_i is Lipschitz yields

$$y_i(k) = g_i(y_i(k-1),\ y_i(k-2), \ldots,\ y_i(k-m), y_{i_j}(k), y_{ij}(k-1), \ldots,\ y_{ij}(k-m)) + d \qquad (2.18)$$

where j=1, 2,..., N_i, and $\|d\| \le e\max(L_j)$.

Using NN function approximation property, a RNN that approximates the unknown function g_i is such that

$$g_i(y_i(k-1),\ y_i(k-2), \ldots,\ y_i(k-m), y_{i_j}(k), y_{ij}(k-1), \ldots,\ y_{ij}(k-m)) = RNN_i(x) + \varepsilon_i(x) \qquad (2.19)$$

where the vector x is given by

$$x = [y_i(k-1),\ y_i(k-2), \ldots,\ y_i(k-m), y_{i_j}(k), y_{ij}(k-1), \ldots,\ y_{ij}(k-m)]. \qquad (2.20)$$

The bounded constant c is then given by

$$c = \varepsilon_{Ni} + e\max(L_j)\ . \qquad (2.21)$$

This completes the proof.

This proof shows that the sensor node output can be approximated as a RNN with inputs as m previous output samples of the same node and m previous output samples of neighboring sensors. The RNN approximates sensor dynamics which can in general be nonlinear. The proposed method can actually be applied for linear and nonlinear, dynamic and static sensor models.

The previous results assume ideal communication links. In cases containing communication link uncertainties, the exact value of $y_{i_j}(k)$ is not available. Instead, we

use the output values of the neighboring sensor nodes combined with confidence factors, i.e., $C_{ji}y_{i_j}(k)$. Then the recurrent neural net sensor node models is given by

$$y_i(k) = RNN_i(y_i(k-1), y_i(k-2), \ldots, y_i(k-m), C_{ji}y_{i_j}(k), C_{ji}y_{ij}(k-1), \ldots, C_{ji}y_{ij}(k-m)) + c. \quad (2.22)$$

Confidence factors for sensor node i are proportional to the signal strength between node i and its neighbors. A confidence factor between neighboring nodes i and j represents a "confidence" of sensor node i from data generated by sensor node j. The factor depends on certain parameters, such as the proximity and distance between two nodes, the terrain between the nodes, the topology of the sensor network, and the received signal strength. We use received signal strength, which can be obtained from receiving sensor nodes, as a measure of the confidence factor. The received signal strength indicator (RSSI) has been used in practical applications as a part of IEEE 802.11 standard, and existing commercial sensor nodes have this capability (Crossbow and MoteIV wireless sensor nodes).

2.4 Application to Sensor Node Fault Detection

Previous results provide a tool for approximating a wireless sensor node output using RNNs. The method can be applied to a wide range of nonlinear dynamic models. A motivation for the above results stems from the need to detect faults in a network of distributed, wireless network of sensor nodes.

In order to detect possible sensor faults at the node level, we compare the real output and the RNN approximation model. If such a difference is larger than a threshold then a fault has occurred at the sensor.

For a sensor node i, its real output $y_i(k)$, and a RNN model output $RNN_i(k)$, if

$\|RNN_i(k) - y_i(k)\| \geq \eta_i$, then a fault has occurred at the sensor node i.

Figure 2.5 and Figure 2.6 show the structure of the modified recurrent network with its inputs consisting of the delayed output signals of the same NN and the previous and current modified output signals from neighboring sensors. It is initially assumed that all confidence factors between the node i and the neighboring nodes are equal to 1. Figure 2.5 shows the topology during the learning phase and Figure 2.6 during the production phase, where a fault analyzer detects the difference between sensor and modified RNN.

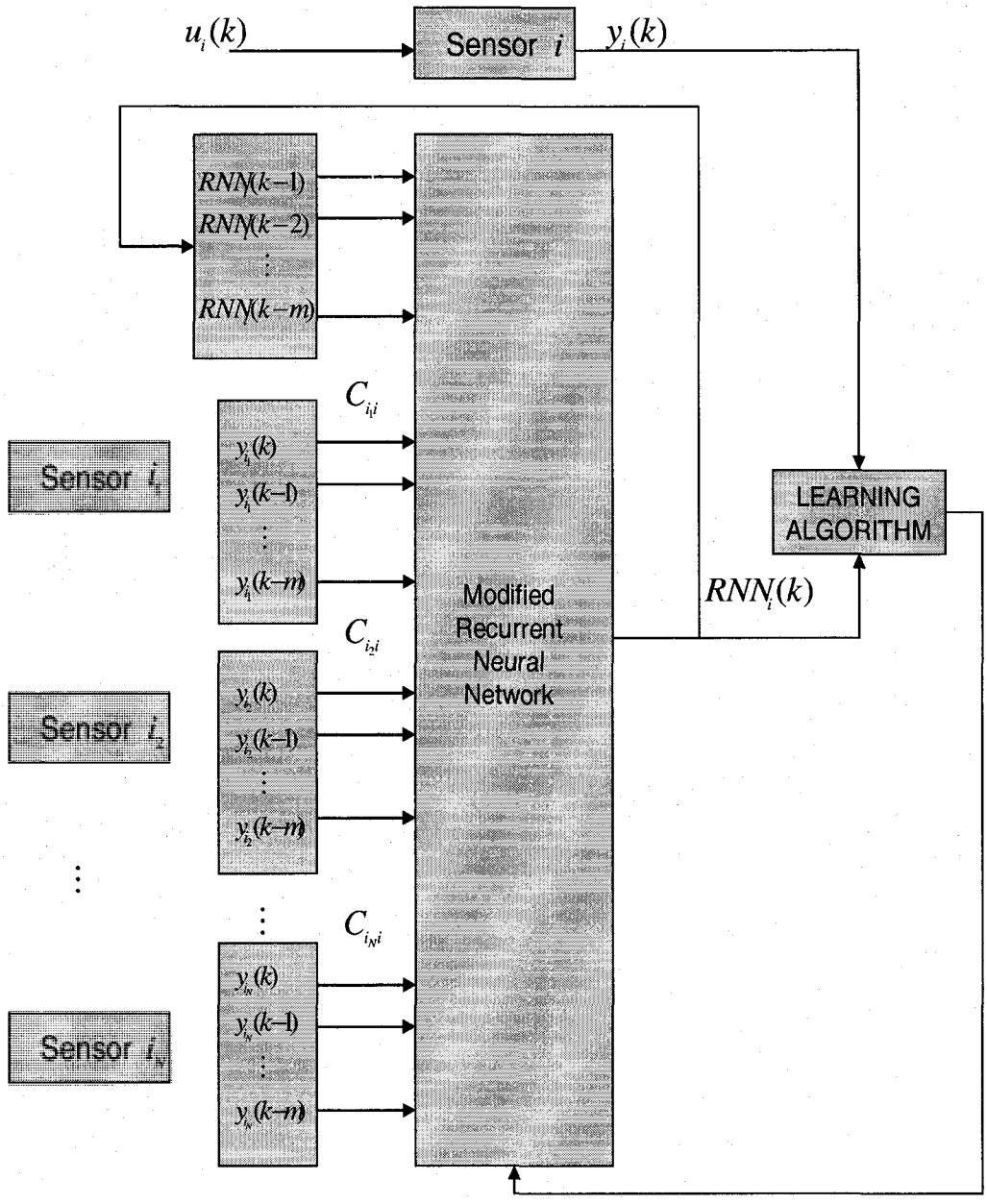


Figure 2.5 Block diagram of the system identification in the learning phase

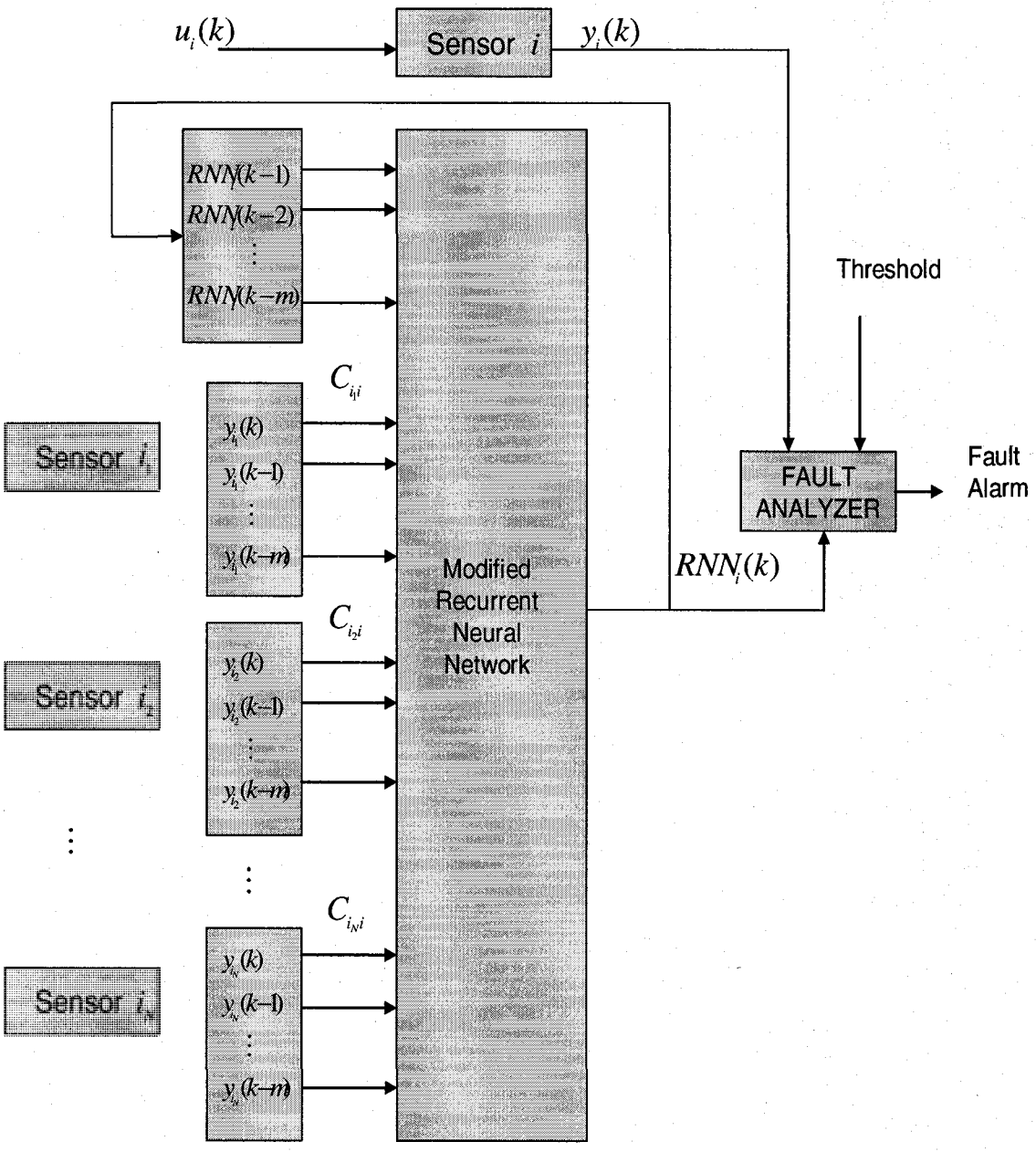


Figure 2.6 Block diagram of the system identification in the production phase

2.5 Simulation and Simulation Results

We have simulated a sensor network with 15 sensor nodes using one sensor per node. Each node has 2 or 3 "visible" neighbors. Of course, if sensor i is a neighbor of j, then the opposite is also true.

Each sensor is modeled as a Hammerstein-Wienner [12] nonlinear feedback dynamic sensor, (Figure 2.3), where the nonlinearity part is an *arctan*(·) function and the dynamical element is given by $H(s) = \frac{1}{s^2 + s + 1}$. Input to the sensor i during both training and production phases is chosen by

$$u_i(k) = 10 + \sin\left(\frac{i + 2\pi k}{3}\right) + n_i(t) \tag{2.23}$$

where $n_i(t)$ is a white noise at sensor node i with the variance of 0.6, and the sampling time is equal to 0.1 seconds.

Each sensor is modeled using a Modified Recurrent Neural Network (MRNN) described in a previous section. A MRNN node has input consisting of delayed output samples of the same node and current and previous outputs of the neighboring sensor nodes. At first, we assumed confidence factors equal to one, but later we made a more realistic assumption with confidence factors less than one.

In the simulation, the RNN has an input layer with 8 nodes, a hidden layer with 10 nodes, and an output layer with one node. The learning algorithm is the standard backpropagation. The learning rates for the first layer and the hidden layer are set to 0.01. The learning phase stopped after the difference between expected and actual artificial neural network (ANN) output reached a steady-state value. The simulation software was written in Microsoft Visual C++ .NET.

Sensor #1 results are shown in Figure 2.7. The output of the MRNN closely approximates the actual output of the sensor with a small error. The MRNN model can certainly reproduce the dynamic behavior of the sensor. Figure 2.8 shows the discrepancy between the actual and the MRNN model outputs with confidence factors set to 1.

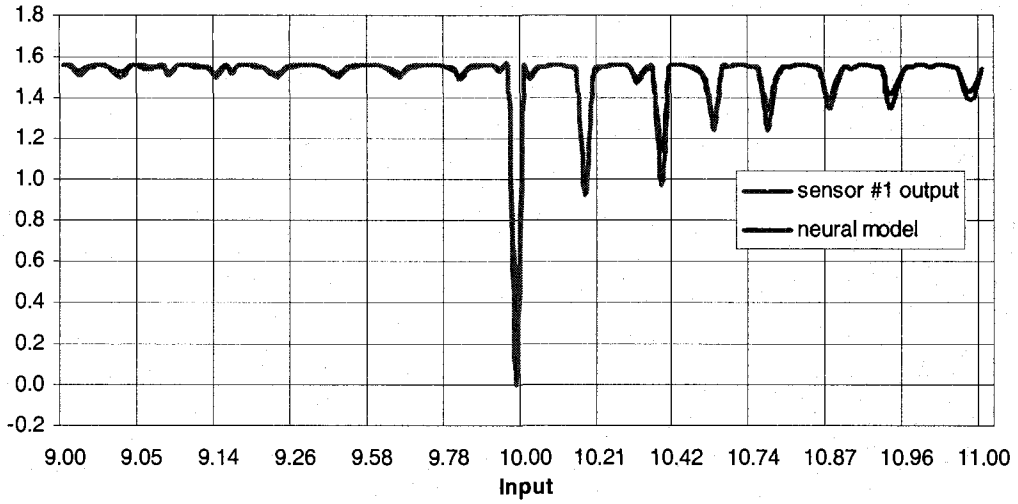


Figure 2.7 Actual output of sensor #1 and its modified recurrent neural network model with confidence factors set to 1

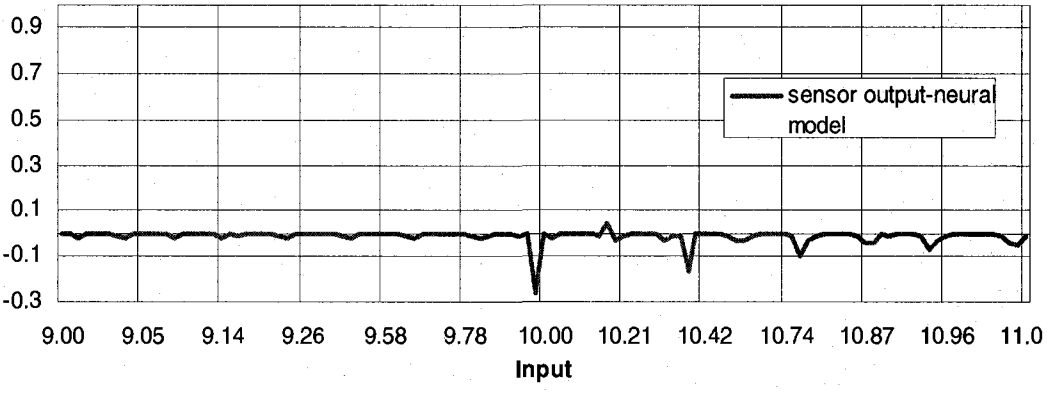


Figure 2.8 Discrepancy between actual output of sensor #1 and its modified recurrent neural network model with confidence factors set to 1

During the learning phase, Figure 2.9 shows the evolution of the difference between the neural network model and the actual sensor output. Notice that this error decreases as the number of iterations increases.

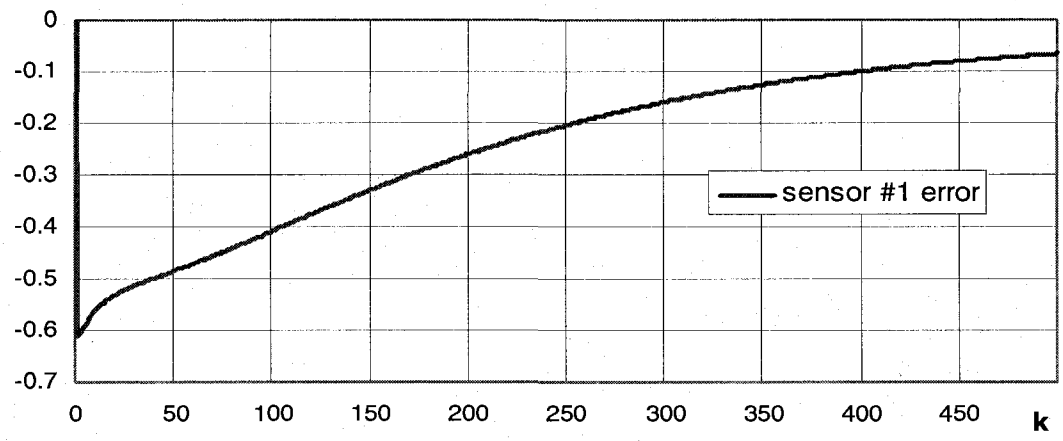


Figure 2.9 Evolution of the difference *e(k)* between the MRNN model and the actual output of sensor #1 with confidence factors set to 1

A more realistic assumption is to consider confidence factors between nodes as being less than one. Taking $C_{21} = 0.8$, $C_{31} = 0.6$, and $C_{41} = 0.95$, the results for sensor node 1 are shown in Figure 2.10.

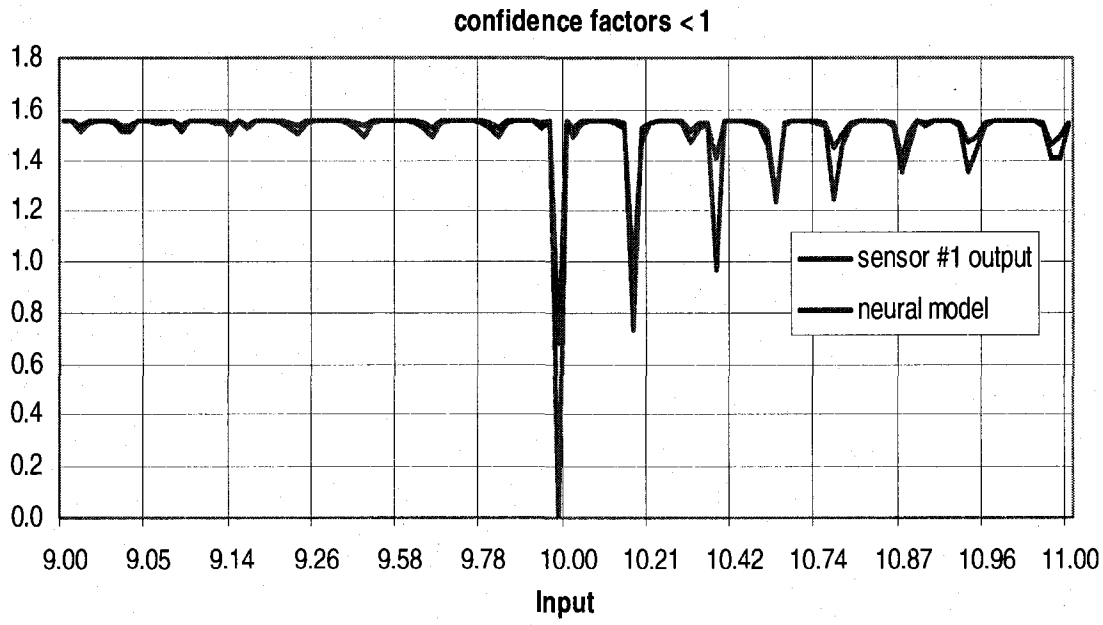


Figure 2.10 Actual output of sensor #1 and its modified recurrent neural network model with confidence factors less than 1

Figure 2.11 shows the sampled output of sensor #1 when this sensor has a fault (drift) starting at 1.6 seconds. Also shown is the estimated MRNN output when the sensor is in a normal healthy mode.

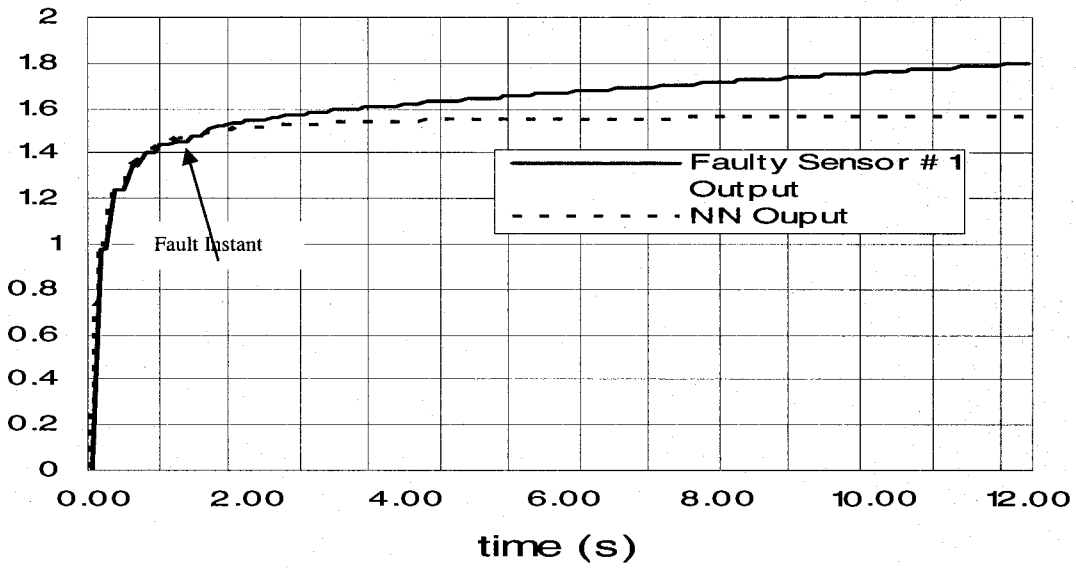


Figure 2.11 Output of faulty sensor #1 and its MRNN model

The neural net learning rate η, which has a value between 0 and 1, plays a key role in the learning process. It affects the rate of convergence during the learning phase. For too small values of the learning rate, the learning process will be very slow with a high probability of convergence. On the other hand, when η approaches one, the learning process is fast with a low probability of convergence. Therefore, using a moderate value of η is recommended. In addition, a number of training samples also plays an important role in modeling accuracy and sensor generalization. By generalization we refer to the ability of the network to approximate the output for an input different from the training set. The results in the simulation illustrate the result.

The initial NN weights also affect the learning time and convergence of the cost function (sum of errors between desired and actual outputs). For instance, when the initial weights are chosen near a local minimum, the cost function will converge to that minimum (particularly for a small learning rate). When the initial weights are chosen near a global minimum, the cost function will converge to this global minimum. In both, the choice of initial weights and the learning rate can affect the number of iterations needed for satisfactory neural network convergence.

A data window with n+1 samples corresponds to current and n previous sensor outputs. The window size affects the precision and accuracy of the step-ahead approximated sensor value as well as the sensitivity of the fault detection technique. We have chosen a window size of four samples. Increasing the window size adds more nodes to the MRNN. Therefore, a trade-off occurs between increased accuracy, additional complexity of the NN, and ultimately the duration of the training process. In particular, for online measurement applications, using a smaller window size is desirable. After satisfactory convergence has been achieved, the validation is provided by estimating the next (future) sensor sample.

2.6 Conclusion

We have developed a dynamic model of a wireless sensor network and its application to sensor failure detection and identification. This model shows how the NN model depends on the sensor model and the network structure. The overall network model corresponds to the topology of the wireless sensor network. The inputs to the NN are taken from the modeled node and neighboring nodes. Communication confidence factor was taken into account in the modeling.

Example simulation is carried on a network with 15 sensor nodes. A fault such as drift was introduced and was successfully detected with modified recurrent neural net model.

CHAPTER 3

MODIFIED RECURRENT NEURAL NETWORK VS. KALMAN FILTERING IN WSN FAULT DETECTION

3.1 Introduction

Many techniques are available for nonlinear dynamic system identification using NNs. Bernieri et al. [21] [22] compared output signals of a NN model and the sensor to detect faults. Once the fault has been detected, the parameters of the NN identifier are compared in order to isolate a fault. Narendra and Parthasarathy [13] demonstrated that NNs can effectively be used for the identification and control of nonlinear dynamic systems. Ahmed [23] presented a rapid neural network for identifying unknown nonlinear dynamic systems when the inputs and outputs are accessible for measurements. Straub and Shroder [24] presented a new approach to identifying nonlinear dynamic systems which is based on a general regression NN. Introducing, proving, and simulating a new tool for nonlinear dynamic systems, like the MRNN technique, need to be compared to some other powerful techniques like Kalman filtering.

3.2 A Simulation Example of MRNN Tool VS. Kalman Filtering

We compared the RNN model with a Kalman filter. The estimated value from the previous time step and the current measurement coming from the real sensor are used as input variables to the Kalman filter. The Kalman filter is a recursive estimator. Thus, only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state. The Kalman filter has two distinct phases: predict and update. The predict phase uses the state estimate from the previous timestep to produce an estimate of the state at the current timestep. In the update phase, measurement information at the current timestep is used to refine this prediction to arrive at a new, (hopefully) more accurate state estimate for the current timestep.

For sensor #1, the results for the recurrent neural network modeling and Kalman filtering techniques are shown in Figure 3.1, including a comparison of both results. The output of the MRNN closely approximates the actual output of the sensor with an error clearly smaller than the one produced using the Kalman filter model. The MRNN model can certainly better reproduce the dynamic behavior of the sensor. Also shown are the discrepancies between the actual output of sensor #1, its resulting model neural network, and Kalman filtering with confidence factors set to 1.

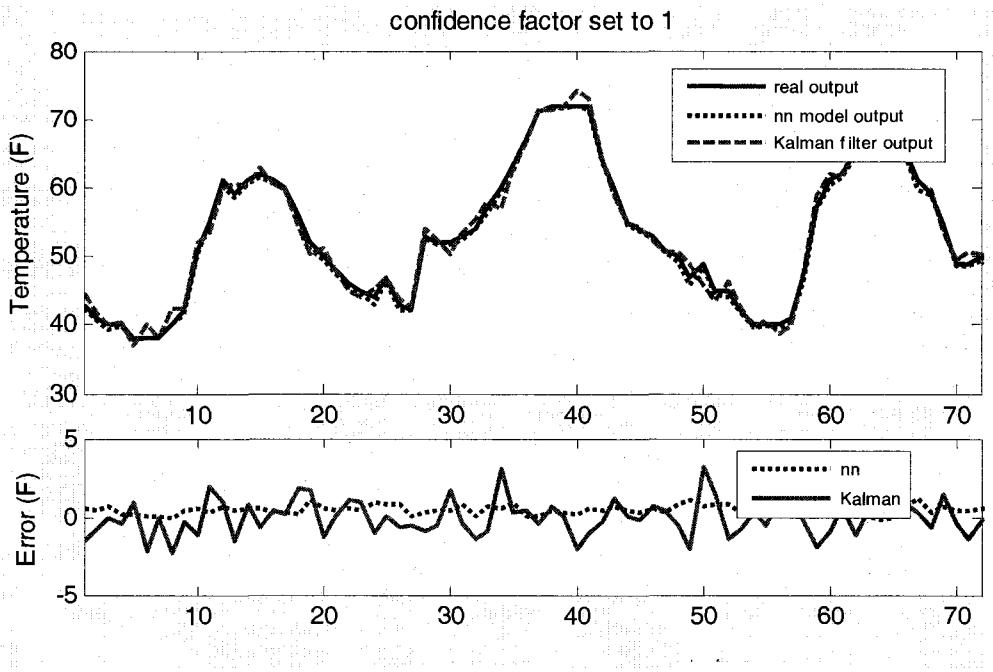


Figure 3.1 Actual output of sensor #1, its models using modified recurrent neural network and Kalman filter, and their corresponding discrepancies with confidence factors set to 1

Figure 3.2 shows the evolution of the error between the NN model and the actual sensor output during the learning phase. The error decreases with the number of training iterations.

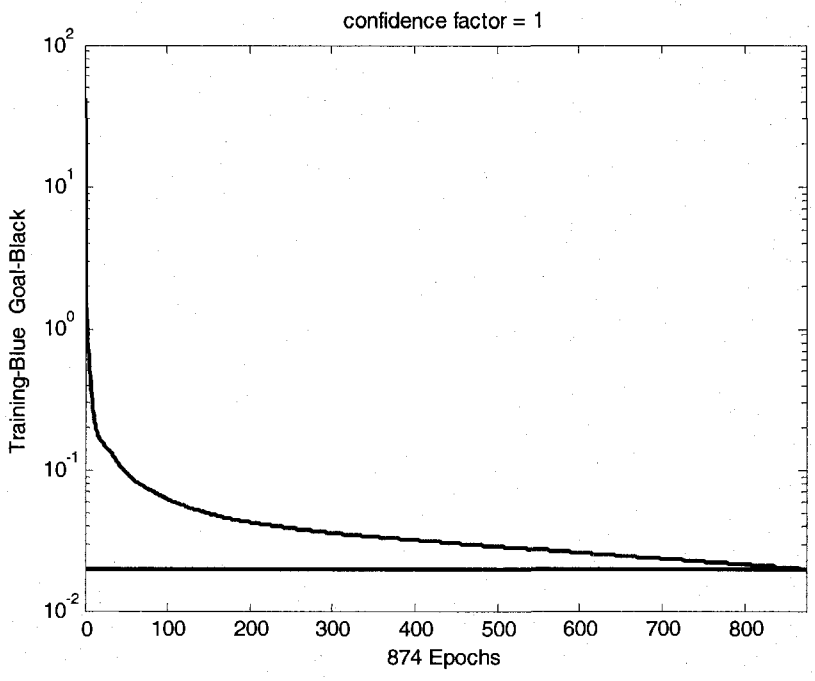


Figure 3.2 Evolution of the difference e(k) between the MRNN model and the actual output of sensor #1 with confidence factors set to 1.

A more realistic assumption is to consider confidence factors between nodes as less than one. Taking $C_{21} = 0.8,\ C_{31} = 0.6,$ and $C_{41} = 0.95$, the results for sensor node 1 are shown in Figure 3.3 and Figure 3.4. One can notice a larger difference between the sensor output and the MRNN model in this case, but the result of our approach is still better than that of the Kalman filtering technique.

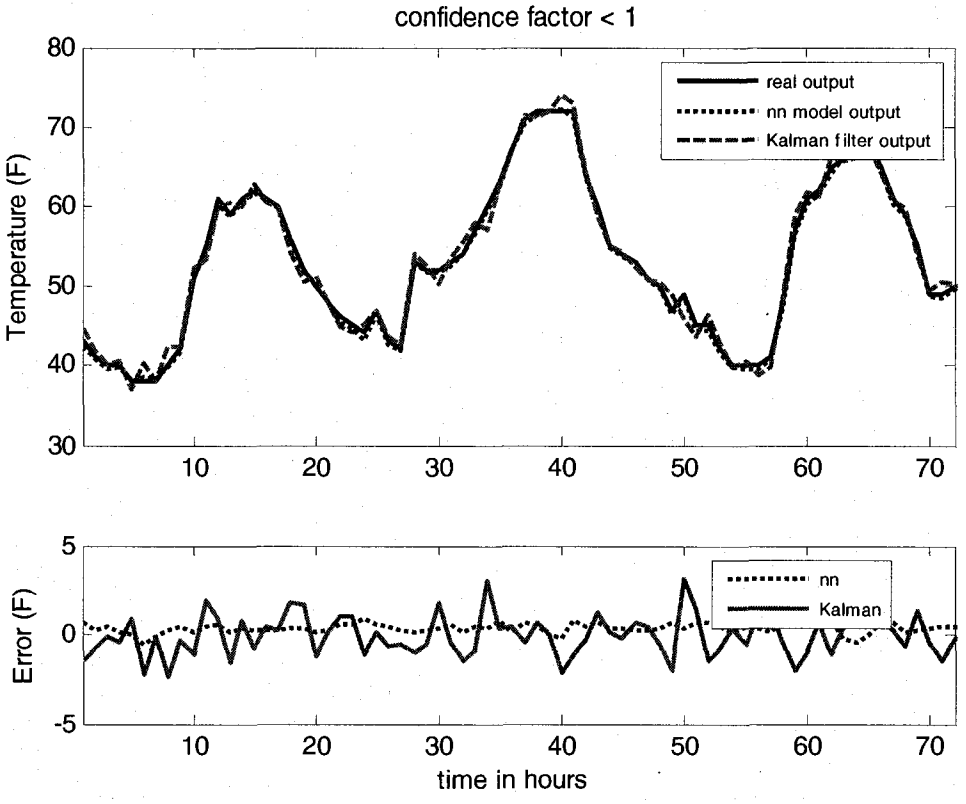


Figure 3.3 Actual output of sensor #1, its models using modified recurrent neural network and Kalman filtering technique, and their corresponding discrepancies with confidence factors less than 1

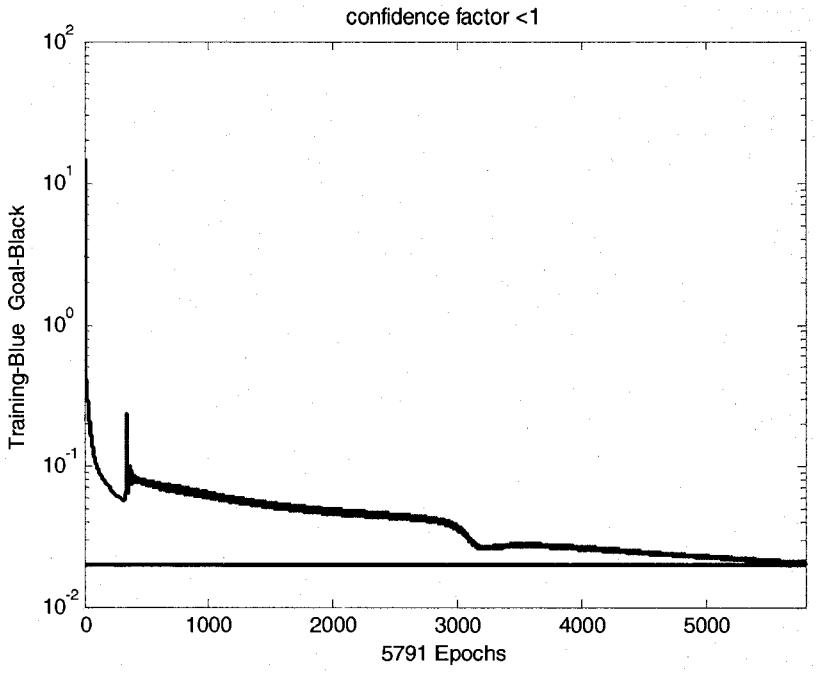


Figure 3.4 Evolution of the difference e(k) between the MRNN model and the actual output of sensor #1 with confidence factors less than 1

To model a sensor fault, we have used a linear drift given by $d(t) = 0.2\,t\,u(t-3)$ where $u(t)$ is a unit step function and time t is in hours. Figure 3.5 shows the sampled output of faulty sensor #1 when this sensor has a fault (linear drift) starting at t_0=3 hours. Also shown is the estimated MRNN output when the sensor is in a normal healthy mode. Using the MRNN modeling technique, the fault is successfully detected when the fault in the sensor output reaches 1 degree Fahrenheit at t=8 hours.

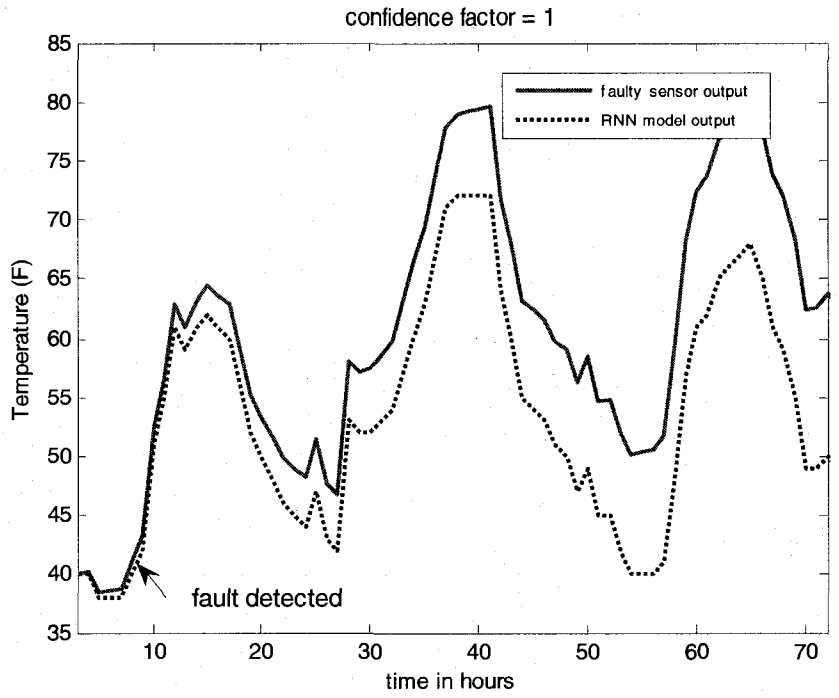


Figure 3.5 Output of faulty sensor #1 and its MRNN model

3.3 Conclusion

RNN modeling technique results are compared to the powerful Kalman filtering technique results. The inputs to the NN are taken from the modeled node and from neighboring nodes. A communication confidence factor is taken into account in the modeling. Simulation with comparison to the Kalman filtering technique is carried out on a network with 15 sensor nodes. A fault such as drift is introduced and can be successfully detected with the modified recurrent neural net model with no early false alarm that could have resulted when using the Kalman filtering approach.

CHAPTER 4

REAL-TIME IMPLEMENTATION OF FAULT DETECTION IN WIRELESS SENSOR NETWORKS USING NEURAL NETWORKS

4.1 Introduction

Wireless sensor networks (WSNs) consist of a set of sensor nodes that can communicate with each other; sensors that measure a desired physical quantity; and the system base station for data collection, processing, and connection to the wide area network. Modern wireless sensor nodes have microprocessors for local data processing, networking, and control purposes [1]. Increases in modern embedded computing power have given rise to many WSN applications. These applications range from medical projects to environmental measurements. For instance, networks have been developed to record vital signs and forward them to a base station for real time analysis, possibly improving triage time. Also being developed are sensor boards that record movement data during the rehabilitation of stroke patients [25]. This raw data would improve physical therapists' ability to track and quantify improvements.

The very heart of WSN technology is the ability to measure remote environmental qualities with low-cost nodes able to self-group into a network topology to reliably forward data to a base station. A vineyard monitoring system that measures soil moisture and the irrigation system's water pressure is given in [26]. Environmental measurements can also help analyze structural health. A WSN has also been implemented that spans the Golden Gate Bridge in San Francisco, CA [27]. These nodes allow engineers to monitor important remote qualities such as ambient vibrations both safely and cost effectively.

WSNs can take remote measurements, organize into a network, and forward data to a base station. Due to the environments they operate in and efforts to maintain cost effectiveness, node failures can occur. The probability of failure increases as the number of nodes in a network increases. The traditional solution to this problem is redundant systems; however, multiplication of sensor devices adds cost, complexity, and power consumption to both the sensor node and the whole network. Most of the present research efforts have concentrated on an analytical redundancy [28] [29] in which sensor measurements are processed analytically and mathematical models are compared with physical measurements. However, with the limited onboard microprocessors and battery power, these approaches decrease the amount of measurements taken while increasing processing time and battery consumption.

In Chapter 2, we presented the theory and modeling of WSNs with recurrent neural networks (RNNs). This approach has been implemented on Moteiv's Tmote Sky platform running the operating system TinyOS.

4.2 Implementation

The hardware testbed uses Moteiv's Tmote Sky wireless sensor node modules. The nodes have an onboard microcontroller; a wireless radio stack; and the ability to take sensor readings of temperature, light, and humidity [30]. The microcontroller is an 8 MHz Texas Instruments MSP430 microcontroller. The radio chipset is a 2.4 GHz Chipcon CC2420 wireless transceiver with an integrated PCB trace antenna [31]. The onboard temperature sensor used in this implementation is Sensirion's SHT11 temperature/humidity sensor [32]. The Tmote Sky node requires a minimum operating voltage of 2.1 volts. To conserve power, this implementation has taken aspects of bandwidth, transmitting/receiving power, and MCU processing time into consideration. All information needed for the training and prediction of the NN is gathered by simply overhearing radio transmissions. As environmental measurements are taken and multi-hopped forward to the root of the collection tree, neighboring nodes overhear the transmissions and record the neighbor's measurements for the local node's NN training. This approach uses no extra radio transmission power or network bandwidth.

The software coding of the implementation is done using TinyOS, an open-source operating system designed for wireless embedded sensor networks [33]. It is specifically designed for the embedded systems with memory constraints and low power consumption. TinyOS uses the programming language NesC [34] which is similar to the programming language C with big differences in the linking model. NesC programs include a configuration file and a module file. The module file looks much like event driven C coding, but the configuration file is the key to NesC. TinyOS has been ported for dozens of hardware platforms and many more chipsets. The configuration file allows

the programmer to link the module code to specific hardware chipsets, background functions, and communication network topologies.

To decrease development time, many common actions of WSNs are already built into TinyOS architecture. An essential function of any WSN is to relay the collected data to a base station for future processing and analysis. TinyOS has addressed this issue and has included provisions for Collection Tree Protocol (CTP) [35]. CTP is based on a tree network where the base station is defined as the root of the tree and all other nodes branch out from their parent nodes in the network. The routing engine is based on expected transmissions (EXT). The EXT of a node is the EXT of its parent plus the link level EXT to its parent. When a node searches for acceptable routes, it will choose the route with the lowest EXT. Because TinyOS is component based, sending messages to the root of a network is very similar to sending messages to a specific node.

To multi-hop a message to the root of the network, the `Send.Send()` command is linked to the collection component instead of the address driven message sending component, used to send node-to-node messages. This component will forward the packet to the Multi-hop Forwarding Engine and relay the packet through the tree to the base station via the route with the lowest EXT. The collection layer only triggers the `Receive.Receive()` event when a packet reaches its final destination (normally the base station). In the fault detection implementation, neighboring nodes need access to the forwarded information to train their NN. Under normal network operation, all packets are received from the radio and screened at the hardware level to determine if the packet is needed by the receiving node. The `Snoop.Receive()` event bypasses this check and is triggered any time a packet is received by a node. The packet's origin can then be

compared against the list of nodes in the routing engine's neighbor table. If the packet is from a valid neighbor, the information is stored for future training.

4.3 Code Overview

Upon applying voltage to the sensor node, the node will go through a pre-defined boot-up sequence. This sequence will initialize components such as the radio, the network engine, and specific sensors. Once the components have successfully come online, a periodic timer is started with a period *T*. Because TinyOS is event driven, the node remains idle until an event handler is triggered. From this point on in the program, the code is no longer sequentially executed. TinyOS will handle events as they occur.

Upon receiving a message, the node must consult the forwarding engine to ensure it is in the node's neighbor table. If the message was overheard from a viable neighbor, the local node stores the information and checks to ensure all the data is now gathered. If data collection is complete, the node sets a flag and checks another routine's flag status. If the situation dictates, the program will enter the NN training subroutine. This process can be seen in Figure 4.1. The workhorse of this program is the subroutine (Figure 4.2) that is executed each time the periodic timer fires.

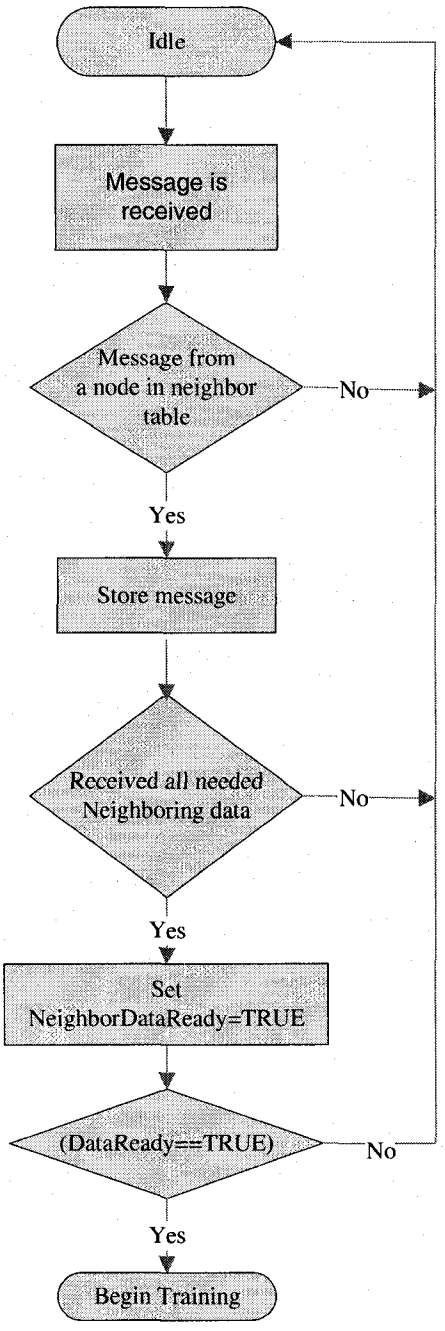


Figure 4.1 Flow chart of subroutine run after message is received

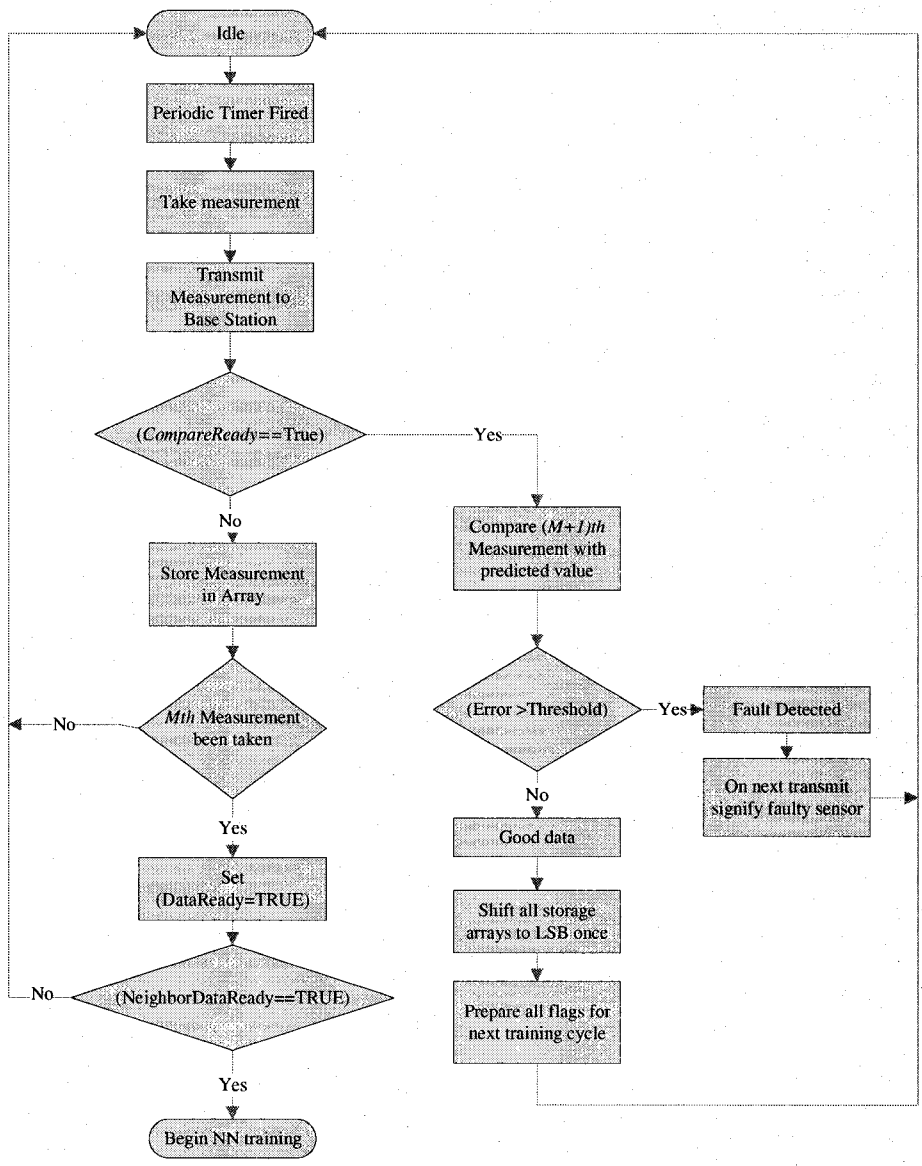


Figure 4.2 Flow chart of subroutine run when periodic timer fires

First, the node calls the sensor to take a reading. If training and prediction were completed prior to this timer cycle, the node would next compare this reading against the NN predicted reading. If the error was greater than a threshold setting, the sensor would have a fault. This information would be relayed so that proper maintenance attention

would be administered. However, if the data was within the threshold range, the data would be verified as good, and preparation would be taken to complete the cycle again.

Had training and prediction not been completed prior, the sensor would store the new measurement in an array. Next, the subroutine would inspect a counter to determine if this was the *M* measurement. If so, a flag would be set, and if other subroutine's flags allowed, NN training would begin. The actual training of the NN is completed as described earlier. Once training has been completed and the next measurement has been predicted, a flag is set to signal that the node is ready to compare the next measurement against the NN's prediction.

4.3.1 Experimental Setup

We conducted an experiment using a nine node network. These nodes were configured in a collection tree topology with two branches, as shown in Figure 4.3.

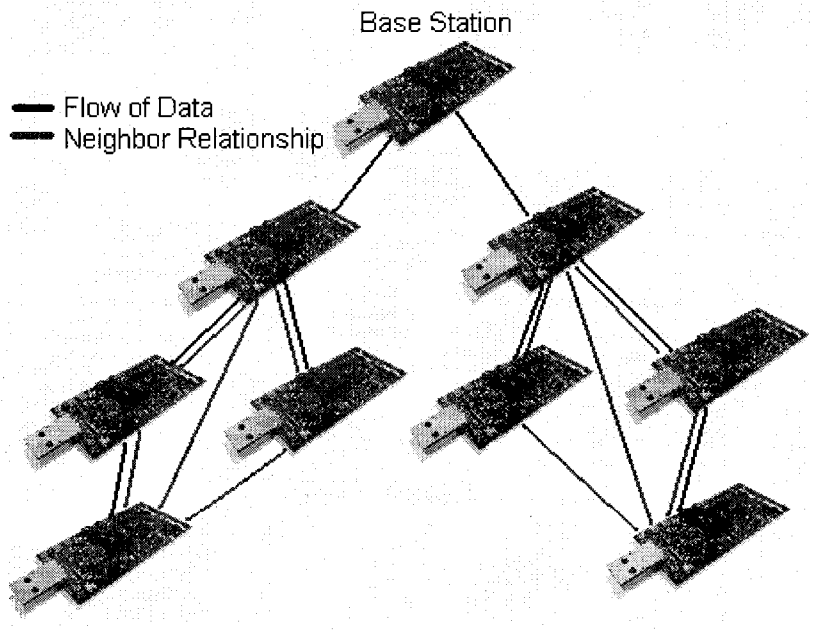


Figure 4.3 Network topology setup for experimental data collection

The two branches were placed in different climates (i.e. separate rooms with differing temperatures.) Temperature measurements were taken and forwarded to the base station node via the route shown by the black lines. As these data samples were sent to the base station, nodes were listening to overhear data from their neighbors, shown by the red lines. After sufficient data was collected to train the network, RNN predictions and fault detection began.

4.3.2 Results

To simplify the results, we will examine the data collected from a single branch shown in Figure 4.3. This branch consists of four neighboring nodes shown connected by red lines in Figure 4.4. Neighboring node 1 was placed in a sunny window, while the other three nodes were dispersed throughout the room. At sampling point 9, an air conditioning unit was turned on to emulate a faulty sensor. We can see that the training node was the closest to the air conditioning unit, followed by neighboring nodes 3 and 2. At sampling point 10, NN prediction and fault detection was started. The NN prediction is shown as a dotted line in Figure 4.4. At sampling point 15, the training node was placed directly on the air conditioning unit to simulate a quick drift fault. The fault was detected at sampling point 16. A fault is defined as the real world measured value lying outside a ±2 degree threshold of the prediction. Although only one node's training is shown here, all nodes except the base station node train, predict, and detect faults simultaneously.

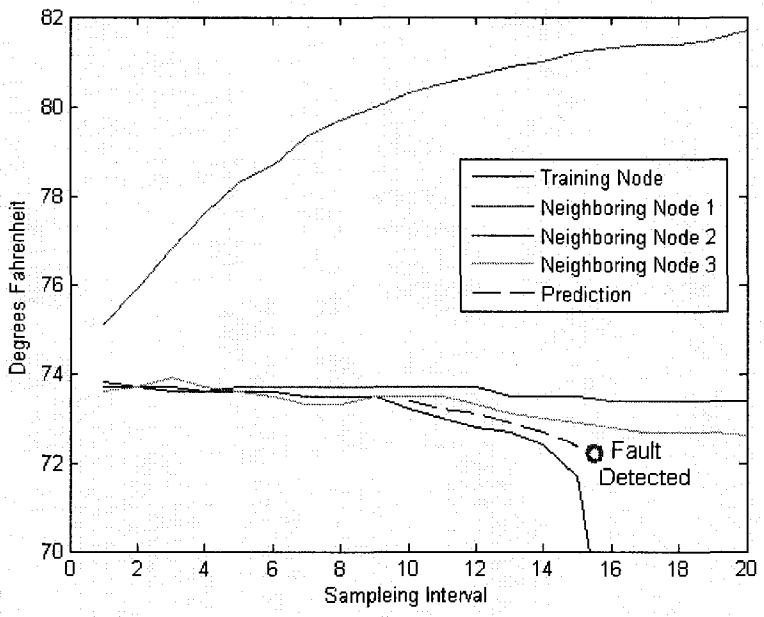


Figure 4.4 Experimental results from a branch of the collection tree

To reduce power consumption and MCU runtime needed to complete the training process, the RNN was configured to allow the highest level of error acceptable in the result. This tolerance saved valuable seconds of processor time during training iterations by omitting result calculations to unneeded significant digits. The implementation set a precision goal of less than one degree Fahrenheit. Obtaining accuracy to hundredths of a degree was considered wasted power and time for our application. With this goal in mind, we were able to tune down the NN to achieve the process from the beginning of the neural network training to prediction of the next measurements to less than 12 seconds. Depending upon the required accuracy needed, this time could be further shortened or extended in other applications.

Power consumption measurements were isolated to the MCU. This limitation removed all variables such as radio stack and sensor power consumption. Due to the

rapidly advancing nature of the node platforms, many available chipsets were all drawing differing amounts of power. Therefore, amperage readings were taken with all peripherals other than the MCU off. While the MCU was idle, the current draw was 7 μA; and while calculating weights for the NN training, it was 1800 μA. Using the standard 3.0 volts and requiring 12 seconds to train, this process only draws 0.0648 Joules. This amount is minuscule when compared to the amount of power drawn by only the onboard CC2420 low power radio. While initialized, the idle radio consumes 20 mA. Over the same 12 second timeframe, the radio alone draws 0.72 Joules. Therefore, this fault detection application has extremely low power consumption.

4.4 Conclusion

The traditional methods of fault detection rely on hardware. Because this approach is software based, it can be implemented at a lower cost and can be easily upgraded in older systems. By simply overhearing data as it is forwarded to the base station, no extra bandwidth or redundant hardware is needed to detect a fault.

The price to be paid for this approach is a small amount of processor time and battery power. Neural network training can be completed in a matter of seconds, and with the implementation of TinyOS's task scheduler, the system remains extremely responsive during the training. During training, nodes can continue to take measurements, transmit data, and forward packets to the base station.

Based on distance, the neighboring nodes closest to the local node are more likely to have similar values. This tendency is especially true in environmental measurements. Due to laws of diffusion, nodes with closer proximity will likely have closer temperature readings than neighboring nodes of farther distance. This information can increase

prediction accuracy when attached to NN weights during training. Without adding extra hardware to preserve the nodes' cost effectiveness, the only onboard equipment to estimate physical distance between the nodes is the received signal strength indictor (RSSI). Implementation of the confidence factor was attempted, but due to interferences and noise, the values changed wildly, even though the actual distance change was only inches. From numerous projects' efforts on localization, we know current indoor RSSI measurements are not capable of reliably measuring precise distance [35]. While the confidence factor approach is ahead of its time, future advances in low cost localization algorithms will allow its implementation.

For indefinite real time fault detection, the minimum sampling interval N must be less than the time required to train. A node must be able to train, predict, compare, and store new data every N seconds. With sampling interval N greater than training time, uniform sampling points can be taken with enough time between sampling for training, prediction, and fault detection.

CHAPTER 5

KALMAN FILTER

5.1 Example Applications

An example application would be providing accurate continuously-updated information about the position and velocity of an object given only a sequence of observations about its position, each of which includes some error. It is used in a wide range of engineering applications from radar to computer vision. Kalman filtering is an important topic in control theory and control systems engineering.

For example, in a radar application, where one is interested in tracking a target, information about the location, speed, and acceleration of the target is measured with a great deal of noise corruption at any instant. The Kalman filter exploits the dynamics of the target, which govern its time evolution, to remove the effects of the noise and get a good estimate of the location of the target at the present time (filtering), at a future time (prediction), or at a time in the past (interpolation or smoothing). A simplified version of a Kalman filter is the alpha beta filter (still commonly used) which has static weighting constants instead of using co-variance matrices.

5.2 Naming and Historical Development

The filter is named after Rudolf E. Kalman, though Thorvald Nicolai Thiele and Peter Swerling actually developed a similar algorithm earlier. Stanley F. Schmidt is generally credited with developing the first implementation of a Kalman filter. It was during a visit of Kalman to the NASA Ames Research Center that he saw the applicability of his ideas to the problem of trajectory estimation for the Apollo program, leading to its incorporation in the Apollo navigation computer. The filter was developed in papers by Swerling (1958), Kalman (1960), and Kalman and Bucy (1961).

5.3 The Kalman Filter

The Kalman filter is a recursive estimator [37]-[41], meaning that only the estimated state from the previous time step and the current measurement are needed to compute the estimate for the current state. In contrast to batch estimation techniques, no history of observations and/or estimates is required. It is unusual in being purely a time domain filter; most filters (for example, a low-pass filter) are formulated in the frequency domain and then transformed back to the time domain for implementation. In what follows, the notation $\hat{x}_{n|m}$ represents the estimate of x at time n given observations up to, and including time m.

The state of the filter is represented by two variables:

- $\hat{x}_{k|k}$, the estimate of the state at time *k*.
- $P_{k|k}$, the error covariance matrix (a measure of the estimated accuracy of the state estimate) .

The Kalman filter has two distinct phases: Predict and Update. The predict phase uses the state estimate from the previous timestep to produce an estimate of the state at the current timestep. In the update phase, measurement information at the current timestep is used to refine this prediction to arrive at a new, (hopefully) more accurate state estimate, again for the current timestep.

Predict

In predicted state,

$$\hat{x}_{k|k-1} = \hat{F}_k \hat{x}_{k-1|k-1} + B_k u_{k-1}, \tag{5.1}$$

where F_k is the state transition model which is applied to the previous state x_{k-1}, and B_k is the control-input model which is applied to the control vector u_k.

For predicted estimate covariance,

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_{k-1}, \tag{5.2}$$

where Q_k is the covariance process noise which is assumed to be drawn from a zero mean multivariate normal distribution.

For innovation or measurement residual,

$$\tilde{y}_k = z_k - H_k \hat{x}_{k|k-1}. \tag{5.3}$$

For innovation (or residual) covariance,

$$S_k = H_k P_{k|k-1} H_k^T + R_k \tag{5.4}$$

where H_k is the observation model which maps the true state space into the observed space, and R_k is the covariance observation noise which is assumed to be zero mean Gaussian white noise.

For optimal Kalman gain,

$$K_k = P_{k|k-1} H_k^T S_k^{-1}. \tag{5.5}$$

To find the updated state estimate

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k. \tag{5.6}$$

Updated estimate covariance is

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}. \tag{5.7}$$

The formula for the updated estimate covariance above is only valid for the optimal Kalman gain.

$$E[x_k - \tilde{x}_{k|k}] = E[x_k - \hat{x}_{k|k-1}] = 0 \tag{5.8}$$

$$E[\tilde{y}] = 0 \tag{5.9}$$

where E[ξ] is the expected value of ξ, and covariance matrices accurately reflect the covariance of estimates.

$$P_{k|k} = \text{cov}(x_k - \hat{x}_{k|k}) \tag{5.10}$$

$$P_{k|k-1} = \text{cov}(x_k - \hat{x}_{k|k-1}) \tag{5.11}$$

$$S_k = \text{cov}(\tilde{y}_k) \tag{5.12}$$

CHAPTER 6

CONCLUSIONS AND RECOMMENDED FUTURE WORK

6.1 Conclusions

Since sensor networks often operate in potentially hostile and harsh environments, components such as sensors have significantly higher fault rates than the traditional integrated semiconductor circuits-based systems. The traditional way of achieving fault-tolerance in dynamic systems is through hardware redundancy such as the use of multiple sensors.

We developed a dynamic model of WSNs and its application to sensor node fault detection where a neural network modeling approach is used for sensor node identification and fault detection. Neural network models will periodically learn the dynamic of the sensors. After each learning period, sensor output is compared to its model so that when a malfunction happens, it can be detected. In case a fault occurs, an alert signal is displayed at both node and base station level.

Since external and internal malfunctions or excessive noise can occur, sensor readings are somewhat uncertain in a sense that no existing sensor will deliver accurate readings at all times. Our model provides an improved reliability over existing sensor

networks to develop a WSN that will have capability of fault detection, isolation, and accommodation.

Advantages:

- Dynamic systems need feedback information about the health status of their nodes in order to recover and heal from the eventual faults. Our model enables such systems to have improved reliability over existing devices.
- The traditional way of achieving fault-tolerance in dynamic systems is through hardware redundancy such as the use of multiple sensors. But multiplication of sensor devices adds cost, complexity, and power consumption to the sensor node and the whole network.
- A fault can be detected with reduced false alarm occurrence, as compared to traditional techniques such as Kalman filter approach.
- Our model fits well for fast online measurement applications.

The presented model can be applied to a wide range of applications in systems where multiple devices are running in parallel, such as WSNs, so that it has the capability of fault detection, isolation, and accommodation. The system can be used in any mission critical WSN application. Any time higher fault awareness is required, the presented model can be applied.

Power consumption and processor time preserved for learning remain an obstacle toward fulfilling a clean, safe, and self-aware wireless sensor network.

6.2 Recommended Future Work

Implementation of the confidence factor was attempted, but due to interferences and noise, the values changed wildly even though the actual distance change was only

inches. From numerous projects' efforts on localization, it is known that the current indoor received signal strength intensity (RSSI) measurements are not capable of reliably measuring distance precisely. While the confidence factor approach is ahead of its time, future advances in low cost localization algorithms will allow its implementation.

Because the neural model uses outputs from neighboring sensors, a failure in one sensor will propagate to dynamically interconnected sensor nodes. This failure can happen between neural network learning sessions and can cause false alarms. This failure will stay until sensor models are trained again. Therefore, more network training causes fewer error propagations between dynamically interconnected nodes. The long term solution is to allow the neural model to grow, meaning to increase the number of layers and/or number of neurons in layers, and train the network for all events, including faulty cases. The result would be a dynamic network which would have immunity against faults and faults propagation. This is an analogy to the human brain, which continuously learns and adjusts to prevent false alarms.

The parameters of each parallel model (a neuron) are estimated separately. In the next step of the process synthesis, the partial models are evaluated, selected, and included in the newly created neuron layers. During the network synthesis new layers are added to the network. The process of the network synthesis leads to the evolution of the resulting model structure in such a way so as to obtain the best quality approximation of the real system output signals. The process is completed when the optimal degree of the network complexity is achieved.

APPENDIX A

SIMULATION SOURCE CODE USING VISUAL C++. NET

```
//SIMULATION CODE USING VISUAL C++. NET


//HEADER FILE


#define time_step 0.1
#define pi        3.14

// Definition of tuning rates
#define tuning_rate_1 0.01 // First layer
#define tuning_rate_2 0.01 // Second layer

#define c21       1
#define c31       1
#define c41       1
#define c12       0.8
#define c32       0.9
#define c52       0.6
#define c13       0.6
#define c23       0.9
#define c43       0.75
#define c14       0.95
#define c34       0.7
#define c54       0.85
#define c15       0.55
#define c25       0.6
#define c45       0.85

double sensor_1(double t);
double sensor_2(double t);
double sensor_3(double t);
double sensor_4(double t);
double sensor_5(double t);

double static_sensor1(double x,int flag);
double static_sensor3(double x,int flag);
double static_sensor4(double x,int flag);

double *get_nn_input_sensor_1(double, double *);
double *get_nn_input_sensor_2(double, double *);
double *get_nn_input_sensor_3(double, double *);
double *get_nn_input_sensor_4(double, double *);
double *get_nn_input_sensor_5(double, double *);

void get_nn_output_1(double [8][10],double *, double *,double);
void get_nn_output_2(double [8][10],double *, double *,double);
void get_nn_output_3(double [8][10],double *, double *,double);
void get_nn_output_4(double [8][10],double *, double *,double);
void get_nn_output_5(double [8][10],double *, double *,double);

void   weight_tuning_nn_1();
void   weight_tuning_nn_2();
void   weight_tuning_nn_3();
void   weight_tuning_nn_4();
void   weight_tuning_nn_5();
```

```
double get_output_sensor_1(double);
double get_output_sensor_2(double);
double get_output_sensor_3(double);
double get_output_sensor_4(double);
double get_output_sensor_5(double);

void failure_detection_sensor_1(FILE *);
void failure_detection_sensor_2(FILE *);
void failure_detection_sensor_3(FILE *);
void failure_detection_sensor_4(FILE *);
void failure_detection_sensor_5(FILE *);

double  get_noise(int );



//MAIN FUNCTION



/*This code is used for modeling a wireless sensor network using
  a modified recurrent neural network. The inputs to the recurrent
  neural network are:The previous and current outputs of the same
  network and the previous and current outputs of the neighboring
  sensors.The learning rates are tuned according to the error
  between the output of output of the sensor and the recurrent
  neural network output.
                        Author: Azzam I. Moustapha
                        Professor: R. Selmic
                          Louisiana Tech University
                        06-15-05*/

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <cstdlib>
#include <time.h>
#include "header.h"

 using namespace std;

FILE  *fweight_1;
FILE  *fweight_2;
FILE  *fweight_3;
FILE  *fweight_4;
FILE  *fweight_5;
main()
{

      int c;

      //cout<<get_noise(i)<<endl;
```

```
  weight_tuning_nn_1();
   //weight_tuning_nn_2();
  //weight_tuning_nn_3();
  //weight_tuning_nn_4();
  //weight_tuning_nn_5();
  //failure_detection_sensor_1(fweight_1);
  //failure_detection_sensor_2(fweight_2);
  //failure_detection_sensor_3(fweight_3);
  //failure_detection_sensor_4(fweight_4);
  //failure_detection_sensor_5(fweight_5);

      while (c=getchar());
      return 0;
}



//SENSOR 1



   /*This code is used for modelling a sensor using
     Hammerstein-Wiener nonlinear feedback Dynamic
     sensor which involve linear dynamic block
     sorrounded by three nonlinear static blocks*/


/*-------------------------------------------------------------------------
------------------
                                          |----------------|
--------------------------|Discrete Version|--------------------------
------------------
                                          |----------------|
-------------------------------------------------------------------------
------------------

  Sensor model consits of 4 sub-blocks:

  |---------|           |-----------|                  |------------|
  | input st|    ---    | Dynamic   |                  | output st  |
  | nonlin. |---|- |--->| Linear    |-------|-------->| nonlin.    |
  |---------|   ---     |-----------|       |         |------------|
                ^                           |
                 |                          |
                |      |-----------|        |
                |      | feedback  |        |
                |<-----| st nonlin.|<-------|
                       |-----------|
-------------------------------------------------------------------------
------------------
       Author: Azzam I. Moustapha
       Professor: R. Selmic
         Louisiana Tech University
       12-07-04
```

```
------------------------------------------------------------------------
------------------*/

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


double sensor_1(double t)
{
double i;
double physical_input;
double final_output;
double out_static_block1;
double out_feedback_block;
int static_flag1;
int dynamic_flag2;
int  static_flag3;
int static_flag4;
double tau;
double a=-1;
double b=-1;
double m=1;
i=0;

//setting the flags in such a way to have a Nonlinear Dynamic Sensor
static_flag1=1;
dynamic_flag2=1;
static_flag3=1;
static_flag4=1;



double out_dynamic_sensor_t;
double out_dynamic_sensor_t_1;
double out_dynamic_sensor_t_2;
out_dynamic_sensor_t_1=0;
out_dynamic_sensor_t_2=0;
tau=0;


// Open the files for the storage of the data


/* main iteration loop */

        // specify measured physical quantity (for example temperature)
        while (i<=t)
            {
        // we have added noise to the sensor input in order to have a
more realistic input
        physical_input =10.0+sin(i*2*(pi)/3);//+get_noise(i);    //24
specified physical input dependance on time
                                        // change this accordingly
```

```
        // find output of the input static block
        out_static_block1 = static_sensor1(physical_input,
static_flag1);
        //tau = out_static_block1 - out_feedback_block;
            // find output of the dynamic block

        out_dynamic_sensor_t=-a*out_dynamic_sensor_t_2-
b*out_dynamic_sensor_t_1+m*tau;

            // replace old values with new values
        out_dynamic_sensor_t_2=out_dynamic_sensor_t_1;
        out_dynamic_sensor_t_1=out_dynamic_sensor_t;

            // find output of the feedback static block
        out_feedback_block = static_sensor3( out_dynamic_sensor_t,
static_flag3);// input to this block is the state x1 from dynamic block

            tau = out_static_block1 - out_feedback_block;

            // find output of the output static block
            final_output = static_sensor4(out_dynamic_sensor_t,
static_flag4);// input to this block is the state x1 from dynamic block
          i=i+time_step;
            }
        return (final_output);

} // END of main function


//nonlinear input static sensor block
double static_sensor1(double x,int flag)

{

      double in_sensor_nonlinearity;
      in_sensor_nonlinearity = atan(x);   /* here we specify the input
sensor nonlinearity;
                                                          change this
function depending on application */

      if (flag==1)
        return(in_sensor_nonlinearity);
      else
      return(x);

}



//nonlinear feedback static sensor function
double static_sensor3(double x,int flag)
{
      if (flag==1)
            return(x);
      else
            return(0);
```

```
}


//nonlinear output static sensor block
double static_sensor4(double x,int flag)

{

      double out_sensor_nonlinearity;
    out_sensor_nonlinearity =atan(x);    /* here we specify the output
sensor nonlinearity;
                                                              change this
function depending on application */

      if (flag==1)
            return(out_sensor_nonlinearity);
      else
            return(x);

}

//END OF PROGRAM


//SENSOR 2

/*This code is used for modelling a sensor using
  Hammerstein-Wiener nonlinear feedback Dynamic
  sensor which involve linear dynamic block
  sorrounded by three nonlinear static blocks
-------------------------------------------------------------------------
------------------
                                        |----------------|
--------------------------|Discrete Version|--------------------------
------------------
                                        |----------------|
-------------------------------------------------------------------------
------------------

  Sensor model consits of 4 sub-blocks:

  |---------|           |-----------|                |------------|
  | input st|    ---    | Dynamic   |                | output st  |
  | nonlin. |---|- |--->| Linear    |-------|------->| nonlin.    |
  |---------|   ---     |-----------|       |        |------------|
                ^                           |
                 |                          |
                |      |-----------|        |
                |      | feedback  |        |
                |<-----| st nonlin.|<-------|
                       |-----------|
/*-----------------------------------------------------------------------
------------------
     Author: Azzam I. Moustapha
     Professor: R. Selmic
       Louisiana Tech University
     12-07-04
```

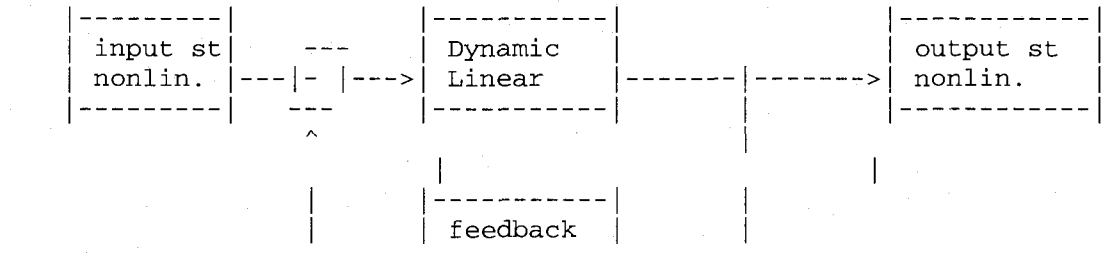
```
------------------------------------------------------------------------
------------------*/

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


using namespace std;

double sensor_2(double t)
{
double i;
double physical_input;
double final_output;
double out_static_block1;
double out_feedback_block;
int static_flag1;
int dynamic_flag2;
int  static_flag3;
int static_flag4;
double tau;
double a=-1;
double b=-1;
double m=1;
i=0;

//setting the flags in such a way to have a Nonlinear Dynamic Sensor
static_flag1=1;
dynamic_flag2=1;
static_flag3=1;
static_flag4=1;



double out_dynamic_sensor_t;
double out_dynamic_sensor_t_1;
double out_dynamic_sensor_t_2;
out_dynamic_sensor_t_1=0;
out_dynamic_sensor_t_2=0;
tau=0;


// Open the files for the storage of the data


/* main iteration loop */

        // specify measured physical quantity (for example temperature)
        while (i<=t)
            {
        physical_input =5+sin(i*(pi/24));    // specified physical
input dependance on time
                                          // change this accordingly
```

```
        // find output of the input static block
        out_static_block1 = static_sensor1(physical_input,
static_flag1);
        //tau = out_static_block1 - out_feedback_block;
            // find output of the dynamic block

        out_dynamic_sensor_t=-a*out_dynamic_sensor_t_2-
b*out_dynamic_sensor_t_1+m*tau;

            // replace old values with new values
        out_dynamic_sensor_t_2=out_dynamic_sensor_t_1;
        out_dynamic_sensor_t_1=out_dynamic_sensor_t;

            // find output of the feedback static block
        out_feedback_block = static_sensor3( out_dynamic_sensor_t,
static_flag3);// input to this block is the state x1 from dynamic block

            tau = out_static_block1 - out_feedback_block;

            // find output of the output static block
            final_output = static_sensor4(out_dynamic_sensor_t,
static_flag4);// input to this block is the state x1 from dynamic block
        i=i+time_step;
            }
        return (final_output);

} // END of main function

//SENSOR 3

/*This code is used for modelling a sensor using
  Hammerstein-Wiener nonlinear feedback Dynamic
  sensor which involve linear dynamic block
  sorrounded by three nonlinear static blocks
-------------------------------------------------------------------------
-----------------
                                        |----------------|
--------------------------|Discrete Version|--------------------------
-----------------
                                        |----------------|
-------------------------------------------------------------------------
-----------------

  Sensor model consits of 4 sub-blocks:

  |---------|           |-----------|                |------------|
  | input st|    ---    | Dynamic   |                | output st  |
  | nonlin. |---|- |--->| Linear    |-------|------->| nonlin.    |
  |---------|   ---     |-----------|       |        |------------|
                ^                           |
                |                           |
                |       |-----------|       |
                |       | feedback  |       |
                |<------| st nonlin.|<------|
                        |-----------|
/*-----------------------------------------------------------------------
-----------------
```

```
     Author: Azzam I. Moustapha
     Professor: R. Selmic
       Louisiana Tech University
     12-07-04
------------------------------------------------------------------------
------------------*/

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


using namespace std;

double sensor_3(double t)
{
double i;
double physical_input;
double final_output;
double out_static_block1;
double out_feedback_block;
int static_flag1;
int dynamic_flag2;
int  static_flag3;
int static_flag4;
double tau;
double a=-1;
double b=-1;
double m=1;
i=0;
//setting the flags in such a way to have a Nonlinear Dynamic Sensor
static_flag1=1;
dynamic_flag2=1;
static_flag3=1;
static_flag4=1;



double out_dynamic_sensor_t;
double out_dynamic_sensor_t_1;
double out_dynamic_sensor_t_2;
out_dynamic_sensor_t_1=0;
out_dynamic_sensor_t_2=0;
tau=0;


// Open the files for the storage of the data


/* main iteration loop */

        // specify measured physical quantity (for example temperature)
while (i<=t)
{
```

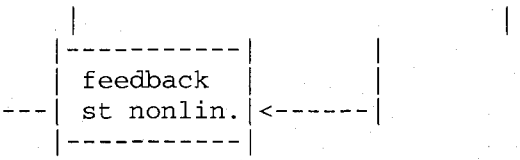
```
        physical_input =15+sin(i*(pi/24));     // specified physical
input dependance on time
                                          // change this accordingly

        // find output of the input static block
        out_static_block1 = static_sensor1(physical_input,
static_flag1);
        //tau = out_static_block1 - out_feedback_block;
            // find output of the dynamic block

        out_dynamic_sensor_t=-a*out_dynamic_sensor_t_2-
b*out_dynamic_sensor_t_1+m*tau;

            // replace old values with new values
        out_dynamic_sensor_t_2=out_dynamic_sensor_t_1;
        out_dynamic_sensor_t_1=out_dynamic_sensor_t;

            // find output of the feedback static block
        out_feedback_block = static_sensor3( out_dynamic_sensor_t,
static_flag3);// input to this block is the state x1 from dynamic block

            tau = out_static_block1 - out_feedback_block;

            // find output of the output static block
            final_output = static_sensor4(out_dynamic_sensor_t,
static_flag4);// input to this block is the state x1 from dynamic block
i=i+time_step;
}
        return (final_output);

} // END of main function

//SENSOR 4

/*This code is used for modelling a sensor using
  Hammerstein-Wiener nonlinear feedback Dynamic
  sensor which involve linear dynamic block
  sorrounded by three nonlinear static blocks
-------------------------------------------------------------------------
------------------
                                        |----------------|
--------------------------|Discrete Version|--------------------------
------------------
                                        |----------------|
-------------------------------------------------------------------------
------------------
```

Sensor model consits of 4 sub-blocks:



```
                |<-----| st nonlin.|<------|
                       |-----------|
/*-------------------------------------------------------------------------
------------------
     Author: Azzam I. Moustapha
     Professor: R. Selmic
       Louisiana Tech University
     12-07-04
-------------------------------------------------------------------------
------------------*/

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"
//#define time_step 0.01

using namespace std;

double sensor_4(double t)
{
double i;
double physical_input;
double final_output;
double out_static_block1;
double out_feedback_block;
int static_flag1;
int dynamic_flag2;
int  static_flag3;
int static_flag4;
double tau;
double a=-1;
double b=-1;
double m=1;
i=0;
//setting the flags in such a way to have a Nonlinear Dynamic Sensor
static_flag1=1;
dynamic_flag2=1;
static_flag3=1;
static_flag4=1;



double out_dynamic_sensor_t;
double out_dynamic_sensor_t_1;
double out_dynamic_sensor_t_2;
out_dynamic_sensor_t_1=0;
out_dynamic_sensor_t_2=0;
tau=0;


// Open the files for the storage of the data


/* main iteration loop */

        // specify measured physical quantity (for example temperature)
```

```
while(i<=t)
{

        physical_input =20+sin(i*(pi/24));    // specified physical
input dependance on time
                                              // change this accordingly

        // find output of the input static block
        out_static_block1 = static_sensor1(physical_input,
static_flag1);
        //tau = out_static_block1 - out_feedback_block;
            // find output of the dynamic block

        out_dynamic_sensor_t=-a*out_dynamic_sensor_t_2-
b*out_dynamic_sensor_t_1+m*tau;

            // replace old values with new values
        out_dynamic_sensor_t_2=out_dynamic_sensor_t_1;
        out_dynamic_sensor_t_1=out_dynamic_sensor_t;

            // find output of the feedback static block
        out_feedback_block = static_sensor3( out_dynamic_sensor_t,
static_flag3);// input to this block is the state x1 from dynamic block

            tau = out_static_block1 - out_feedback_block;

            // find output of the output static block
            final_output = static_sensor4(out_dynamic_sensor_t,
static_flag4);// input to this block is the state x1 from dynamic block
          i=i+time_step;
}
        return (final_output);

} // END of main function

//SENSOR 5

/*This code is used for modelling a sensor using
  Hammerstein-Wiener nonlinear feedback Dynamic
  sensor which involve linear dynamic block
  sorrounded by three nonlinear static blocks
-------------------------------------------------------------------------
------------------
                                          |----------------|
--------------------------|Discrete Version|--------------------------
------------------
                                          |----------------|
-------------------------------------------------------------------------
------------------

  Sensor model consits of 4 sub-blocks:

  |---------|           |-----------|                |------------|
  | input st|    ---    | Dynamic   |                | output st  |
  | nonlin. |---|- |--->| Linear    |-------|------->| nonlin.    |
  |---------|   ---     |-----------|       |        |------------|
                ^                           |
```




```
/*-------------------------------------------------------------------------
------------------
     Author: Azzam I. Moustapha
     Professor: R. Selmic
       Louisiana Tech University
     12-07-04
-------------------------------------------------------------------------
------------------*/

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


using namespace std;

double sensor_5(double t)
{
double i;
double physical_input;
double final_output;
double out_static_block1;
double out_feedback_block;
int static_flag1;
int dynamic_flag2;
int  static_flag3;
int static_flag4;
double tau;
double a=-1;
double b=-1;
double m=1;
i=0;
//setting the flags in such a way to have a Nonlinear Dynamic Sensor
static_flag1=1;
dynamic_flag2=1;
static_flag3=1;
static_flag4=1;



double out_dynamic_sensor_t;
double out_dynamic_sensor_t_1;
double out_dynamic_sensor_t_2;
out_dynamic_sensor_t_1=0;
out_dynamic_sensor_t_2=0;
tau=0;


// Open the files for the storage of the data
```

```
/* main iteration loop */

        // specify measured physical quantity (for example temperature)
while (i<=t)
{
        physical_input =11+sin(i*(pi/24));     // specified physical
input dependance on time
                                          // change this accordingly

        // find output of the input static block
        out_static_block1 = static_sensor1(physical_input,
static_flag1);
        //tau = out_static_block1 - out_feedback_block;
            // find output of the dynamic block

        out_dynamic_sensor_t=-a*out_dynamic_sensor_t_2-
b*out_dynamic_sensor_t_1+m*tau;

            // replace old values with new values
        out_dynamic_sensor_t_2=out_dynamic_sensor_t_1;
        out_dynamic_sensor_t_1=out_dynamic_sensor_t;

            // find output of the feedback static block
        out_feedback_block = static_sensor3( out_dynamic_sensor_t,
static_flag3);// input to this block is the state x1 from dynamic block

            tau = out_static_block1 - out_feedback_block;

            // find output of the output static block
            final_output = static_sensor4(out_dynamic_sensor_t,
static_flag4);// input to this block is the state x1 from dynamic block
i=i+time_step;
}
        return (final_output);

} // END of main function



//NOISE

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"

using namespace std;
double get_noise(int i)
{
double  RAN_MAX = 3267600;
double n;
int e;
int co;
srand(1002*i);
co=0;
```

```
e=rand()*1000/RAN_MAX;
n=100*(rand()/(RAN_MAX*100))*pow(-1,e);
return(n);
}


//NN for SENSOR 1

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w's)
     for the first and second layer used in the modified recurrent
     neural network */


using namespace std;
extern FILE  *fweight_1;

void weight_tuning_nn_1()
{
 FILE  *ferror_1;
 FILE  *fweight_1;
 int i,j,T;
 double t;
 double u[8];
 double v[8][10];
 double v0[10];
 double w[10];
 double w0;
 double *k;
 double p1[10],s1[10],sigma[10];
 double p2[10];
 double output[2];
 double E0,SE1[10],E1[8][10],SE,SE0;

 ferror_1=fopen("nn_error_1","w");
 fweight_1=fopen("nn_weight_1","w");
 t=0.0;
 T=0;
 SE0=0;

for (i=0;i<5;i++)
{
  output[0]=0;
  output[1]=0;
}

for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
}
```

```
for (i=0; i<8; i++)
  for (j=0; j<10;j++)
      {
           v[i][j] = 0;
    }


for (i=0;i<10;i++)
   {
            v0[i] =0;
   }

for (i=0; i<10; i++)
   {
        w[i]=0;
   }
w0=0;

do
{

for (i=0;i<10;i++)
  {
       SE1[i]=0;
  }
t=0;

      do
      {

      // Getting inputs from neighboring sensors
      k=get_nn_input_sensor_1(t, output);

      for(i=0;i<8;i++)
      u[i]=*(k+i);

// Resetting variables
for (i=0;i<10;i++)
   {
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
    SE1[i]=0;
   }

// Calculating output of the modified recurrent NN during learing phase
for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               p1[i]=p1[i]+v[j][i]*u[j];


for (i=0;i<10;i++)

    {
        s1[i]=p1[i]+v0[i];
```

```
      sigma[i] = 1.0/(1.0 + exp(-s1[i]));
    }

for (i=0;i<10;i++)
   {
     p2[i]=w[i]*sigma[i];
     output[0]=output[0]+p2[i];
   }

output[0]=output[0]+w0;
output[1]=output[0];


  // Calculating error between sensor output and RNN output
E0=sensor_1(t)-output[0];
output[0]=0;

for (i=0;i<8;i++)
 for (j=0;j<10;j++)
  {
      E1[i][j]=sigma[j]*(1-sigma[j])*E0*w[i];
      SE1[i]=SE1[i]+E1[i][j];
  }

  // Updating weights of the second layer according to the error
for (i=0;i<10;i++)
  {
      w[i]=w[i]+tuning_rate_2*E0*sigma[i];
  }

 // Updating weights of the first layer according to the error
for (i=0;i<8;i++)
  for (j=0;j<10;j++)
   {
     v[i][j]=v[i][j]+tuning_rate_1*SE1[j]*u[i];
   }
 // Calculating the sum of errors
SE0=SE0+E0;

t=t+time_step;
//cout<<"Modeling Sensor #1 ...";
//system("cls");
}
while (t<12.0);

SE=0.5*SE0*SE0;

fprintf(ferror_1,"%d           %.16f\n",T,SE0);
SE0=0;
T=T+1;
cout<<SE<<endl;
//system("cls");
}
//Checking if the error is higher than threshold
while (SE>2.61e-10);//0.00041);//2.61e-8);//-6
cout<<endl<<"Error is: "<<SE<<endl<<endl;
for (i=0;i<8;i++)
```

```
{
   for (j=0;j<10;j++)
   fprintf(fweight_1,"%lf\n",v[i][j]);
}
for (i=0;i<10;i++)
fprintf(fweight_1,"%lf\n",w[i]);
for (i=0;i<10;i++)
fprintf(fweight_1,"%lf\n",v0[i]);
fprintf(fweight_1,"%lf\n",w0);
t=0;
get_nn_output_1(v,w,v0,w0);
fcloseall();
return;
}

double *get_nn_input_sensor_1(double t, double *output)
{
double in[8];

// input to NN from previous output of same NN
in[0]=output[0];
// input to NN from previous output of same NN
in[1]=output[1];
// input to NN from output of neighbor sensor 2
in[2]=c21*sensor_2(t);
// input to NN from previous output of neighbor sensor 2
in[3]=c21*sensor_2(t-time_step);
//input to NN from output of neighbor sensor 3
in[4]=c31*sensor_3(t);
// input to NN from previous output of neighbor sensor 3
in[5]=c31*sensor_3(t-time_step);
//input to NN from output of neighbor sensor 4
in[6]=c41*sensor_4(t);
// input to NN from previous output of neighbor sensor 4
in[7]=c41*sensor_4(t-time_step);

return(in);
}



void get_nn_output_1( double v[8][10],double *w, double *v0, double w0
)

{
  FILE  *fsensor_1;
  int C;
  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  fsensor_1=fopen("NN_output_1","w");
```

```
  // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
C=0;
do
{
// Getting input from neighboring sensors
 l=get_nn_input_sensor_1(t, output);
 for(i=0;i<8;i++)
 s[i]=*(l+i);

  // Resetting variables
for (i=0;i<10;i++)
  {
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
  }
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
   output[1]=output[0];

   fprintf(fsensor_1, "%f        %f
%.16f\n",t,sensor_1(t),output[0]);
   //Resetting the output
   output[0]=0;


  t=t+time_step;
}
  while (t<12.0);

  fcloseall();
  return  ;
}
```

```
//NN for SENSOR 2

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


/* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


using namespace std;

extern FILE  *fweight_2;
void weight_tuning_nn_2()
{
FILE  *ferror_2;
FILE  *fweight_2;
int i,j,T;
double t;
double u[8];
double v[8][10];
double v0[10];
double w[10];
double w0;
double *k;
double p1[10],s1[10],sigma[10];
double p2[10];
double output[2];
double E0,SE1[10],E1[8][10],SE,SE0;
ferror_2=fopen("nn_error_2","w");
fweight_2=fopen("nn_weight_2","w");
t=0.0;
T=0;
SE0=0;

for (i=0;i<5;i++)
  {
   output[0]=0;
   output[1]=0;
  }



for (i=0;i<10;i++)
  {
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
  }



for (i=0; i<8; i++)
```

```
   for (j=0; j<10;j++)
      {
      v[i][j] = 0;
    }



for (i=0;i<10;i++)
   {
      v0[i] =0;
   }

for (i=0; i<10; i++)
   {
    w[i]=0;
   }

w0=0;

do
{

for (i=0;i<10;i++)
      {
            SE1[i]=0;
    }
t=0;

 do
      {


// Getting inputs from neighboring sensors
k=get_nn_input_sensor_2(t, output);

for(i=0;i<8;i++)
u[i]=*(k+i);



// Resetting variables
for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
    SE1[i]=0;
}

// Calculating output of the modified recurrent NN during learing phase


for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               p1[i]=p1[i]+v[j][i]*u[j];
```

```
   for (i=0;i<10;i++)
   {
      s1[i]=p1[i]+v0[i];
    sigma[i] = 1.0/(1.0 + exp(-s1[i]));
   }

     for (i=0;i<10;i++)
   {
    p2[i]=w[i]*sigma[i];
    output[0]=output[0]+p2[i];
   }

output[0]=output[0]+w0;
output[1]=output[0];

 // Calculating error between sensor output and RNN output
E0=sensor_2(t)-output[0];
output[0]=0;

for (i=0;i<8;i++)
  for (j=0;j<10;j++)
    {
       E1[i][j]=sigma[j]*(1-sigma[j])*E0*w[i];
       SE1[i]=SE1[i]+E1[i][j];
    }

 // Updating weights of the second layer according to the error
for (i=0;i<10;i++)
   {
      w[i]=w[i]+tuning_rate_2*E0*sigma[i];
   }

// Updating weights of the first layer according to the error
for (i=0;i<8;i++)
  for (j=0;j<10;j++)
    {
      v[i][j]=v[i][j]+tuning_rate_1*SE1[j]*u[i];
    }

// Calculating the sum of errors
SE0=SE0+E0;
t=t+time_step;
}
while (t<12.0);

cout<<"Modeling Sensor #2 ...";
system("cls");

SE=0.5*SE0*SE0;
SE0=0;
fprintf(ferror_2,"%d           %.16f\n",T,SE);
T=T+1;
}
// Checking if the error is higher than threshold
while (SE>2.61e-10);

cout<<endl<<"Error is: "<<SE<<endl<<endl;
```

```
for (i=0;i<8;i++)
{
   for (j=0;j<10;j++)
    fprintf(fweight_2,"%lf\n",v[i][j]);
}
for (i=0;i<10;i++)
  fprintf(fweight_2,"%lf\n",w[i]);
for (i=0;i<10;i++)
  fprintf(fweight_2,"%lf\n",v0[i]);
  fprintf(fweight_2,"%lf\n",w0);
t=0;
get_nn_output_2(v,w,v0,w0);
fcloseall();
return ;
}

double *get_nn_input_sensor_2(double t, double *output)
{
double in[8];
// input to NN from previous output of same NN
in[0]=output[0];
// input to NN from previous output of same NN
in[1]=output[1];
// input to NN from output of neighbor sensor 1
in[2]=sensor_1(t);//c12
// input to NN from previous output of neighbor sensor 1
in[3]=sensor_1(t-time_step);//c12
//input to NN from output of neighbor sensor 3
in[4]=sensor_3(t);//c32
// input to NN from previous output of neighbor sensor 3
in[5]=sensor_3(t-time_step);//c32
//input to NN from output of neighbor sensor 4
in[6]=sensor_5(t);//c52
// input to NN from previous output of neighbor sensor 4
in[7]=sensor_5(t-time_step);//c52

return(in);
}


void get_nn_output_2( double v[8][10],double *w, double *v0, double w0
)

{
  FILE  *fsensor_2;

  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  fsensor_2=fopen("NN_output_2","w");
  // Initialization of output
```

```
  output[0]=0;
  output[1]=0;
  t=0;

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_2(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }


   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

output[0]=output[0]+w0;
output[1]=output[0];
    // Printing results
fprintf(fsensor_2, "%f        %f
%.16f\n",t,sensor_2(t),output[0]);
  // Resetting the output
   output[0]=0;
   t=t+time_step;
}
  while (t<12.0);
  fcloseall();
  return  ;
}

//NN for SENSOR 3


/* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
```

```
     neural network */

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"

using namespace std;

extern FILE  *fweight_3;

void weight_tuning_nn_3()
{
 FILE  *ferror_3;
 FILE  *fweight_3;
int i,j,T;
double t;
double u[8];
double v[8][10];
double v0[10];
double w[10];
double w0;
double *k;
double p1[10],s1[10],sigma[10];

double p2[10];

double output[2];
double E0,SE1[10],E1[8][10],SE,SE0;
ferror_3=fopen("nn_error_3","w");
fweight_3=fopen("nn_weight_3","w");

t=0.0;
T=0;
SE0=0;
for (i=0;i<5;i++)
{

output[0]=0;
output[1]=0;
}


for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
}



         for (i=0; i<8; i++)
         for (j=0; j<10;j++)
      {
        v[i][j] = 0;
    }
```

```
   for (i=0;i<10;i++)
   {
     v0[i] =0;
   }
   for (i=0; i<10; i++)
        {
            w[i]=0;
        }

            w0=0;

do
{

      for (i=0;i<10;i++)
{     SE1[i]=0;
}
      t=0;

      do
      {


// Getting inputs from neighboring sensors
k=get_nn_input_sensor_3(t, output);

for(i=0;i<8;i++)
u[i]=*(k+i);



/* Resetting variables
for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
    SE1[i]=0;
}

// Calculating output of the modified recurrent NN during learing phase


for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               p1[i]=p1[i]+v[j][i]*u[j];


   for (i=0;i<10;i++)
   {
      s1[i]=p1[i]+v0[i];
    sigma[i] = 1.0/(1.0 + exp(-s1[i]));
   }
```

```
   for (i=0;i<10;i++)
   {
    p2[i]=w[i]*sigma[i];
    output[0]=output[0]+p2[i];
   }

 output[0]=output[0]+w0;
output[1]=output[0];
 // Calculating error between sensor output and RNN output
 E0=sensor_3(t)-output[0];
output[0]=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{
      E1[i][j]=sigma[j]*(1-sigma[j])*E0*w[i];
      SE1[i]=SE1[i]+E1[i][j];
}

 // Updating weights of the second layer according to the error
for (i=0;i<10;i++)
{
      w[i]=w[i]+tuning_rate_2*E0*sigma[i];

}

// Updating weights of the first layer according to the error
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{
v[i][j]=v[i][j]+tuning_rate_1*SE1[j]*u[i];
}
// Calculating the sum of errors
SE0=SE0+E0;



t=t+time_step;

}
while (t<12.0);

cout<<"Modeling Sensor #3 ...";
system("cls");

SE=0.5*SE0*SE0;
SE0=0;
fprintf(ferror_3,"%d          %.16f\n",T,SE);
T=T+1;
}
// Checking if the error is higher than threshold
while (SE>2.61e-10);

cout<<endl<<"Error is: "<<SE<<endl<<endl;
for (i=0;i<8;i++)
{
```

```
   for (j=0;j<10;j++)
   fprintf(fweight_3,"%lf\n",v[i][j]);
}
for (i=0;i<10;i++)
fprintf(fweight_3,"%lf\n",w[i]);
for (i=0;i<10;i++)
fprintf(fweight_3,"%lf\n",v0[i]);
fprintf(fweight_3,"%lf\n",w0);
t=0;
get_nn_output_3(v,w,v0,w0);
fcloseall();
return ;
}

double *get_nn_input_sensor_3(double t, double *output)
{
double in[8];

// input to NN from previous output of same NN
in[0]=output[0];
// input to NN from previous output of same NN
in[1]=output[1];
// input to NN from output of neighbor sensor 1
in[2]=c13*sensor_1(t);
// input to NN from previous output of neighbor sensor 1
in[3]=c13*sensor_1(t-time_step);
//input to NN from output of neighbor sensor 2
in[4]=c23*sensor_2(t);
// input to NN from previous output of neighbor sensor 2
in[5]=c23*sensor_2(t-time_step);

//input to NN from output of neighbor sensor 4
in[6]=c43*sensor_4(t);
// input to NN from previous output of neighbor sensor 4
in[7]=c43*sensor_4(t-time_step);

return(in);
}


void get_nn_output_3( double v[8][10],double *w, double *v0, double w0
)

{
  FILE  *fsensor_3;

  int i,j;
  double t;
  double *1;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  fsensor_3=fopen("NN_output_3","w");
  // Initialization of output
```

```
  output[0]=0;
  output[1]=0;
  t=0;

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_3(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);



  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }

   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
output[1]=output[0];
    // Printing results
   fprintf(fsensor_3, "%f         %f
%.16f\n",t,sensor_3(t),output[0]);
  // Resetting the output
   output[0]=0;
   t=t+time_step;
}
  while (t<12.0);
  fcloseall();
  return  ;
}

//NN for SENSOR 1

/* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
```

```
     neural network */

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"

using namespace std;

extern FILE  *fweight_4;

void weight_tuning_nn_4()
{
 FILE  *ferror_4;
 FILE  *fweight_4;
int i,j,T;
double t;
double u[8];
double v[8][10];
double v0[10];
double w[10];
double w0;
double *k;
double p1[10],s1[10],sigma[10];

double p2[10];

double output[2];
double E0,SE1[10],E1[8][10],SE,SE0;
ferror_4=fopen("nn_error_4","w");
fweight_4=fopen("nn_weight_4","w");
t=0.0;
T=0;
SE0=0;
for (i=0;i<5;i++)
{

output[0]=0;
output[1]=0;
}

for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
}


         for (i=0; i<8; i++)
         for (j=0; j<10;j++)
      {

                  v[i][j] = 0;

   }
```

```
   for (i=0;i<10;i++)
   {
            v0[i] =0;

   }

   for (i=0; i<10; i++)
        {
            w[i]=0;

        }

            w0=0;


do
{

      for (i=0;i<10;i++)
{     SE1[i]=0;
}
      t=0;

      do
      {


// Getting inputs from neighboring sensors
k=get_nn_input_sensor_4(t, output);

for(i=0;i<8;i++)
u[i]=*(k+i);


// Resetting variables
for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
    SE1[i]=0;
}

// Calculating output of the modified recurrent NN during learing phase


for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               p1[i]=p1[i]+v[j][i]*u[j];


   for (i=0;i<10;i++)
   {
      s1[i]=p1[i]+v0[i];
    sigma[i] = 1.0/(1.0 + exp(-s1[i]));
```

```
   }


   for (i=0;i<10;i++)
   {
    p2[i]=w[i]*sigma[i];
    output[0]=output[0]+p2[i];
   }

 output[0]=output[0]+w0;
output[1]=output[0];
 // Calculating error between sensor output and RNN output
E0=sensor_4(t)-output[0];
output[0]=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{
      E1[i][j]=sigma[j]*(1-sigma[j])*E0*w[i];
      SE1[i]=SE1[i]+E1[i][j];
}

 // Updating weights of the second layer according to the error
for (i=0;i<10;i++)
{
      w[i]=w[i]+tuning_rate_2*E0*sigma[i];

}

// Updating weights of the first layer according to the error
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{
v[i][j]=v[i][j]+tuning_rate_1*SE1[j]*u[i];
}
// Calculating the sum of errors
SE0=SE0+E0;

t=t+time_step;

}
while (t<12.0);

cout<<"Modeling Sensor #4 ...";
system("cls");

SE=0.5*SE0*SE0;
SE0=0;
fprintf(ferror_4,"%d          %.16f\n",T,SE);
T=T+1;
}
// Checking if the error is higher than threshold
while (SE>2.61e-10);

cout<<endl<<"Error is: "<<SE<<endl<<endl;
for (i=0;i<8;i++)
{
```

```
   for (j=0;j<10;j++)
   fprintf(fweight_4,"%lf\n",v[i][j]);
}
for (i=0;i<10;i++)
fprintf(fweight_4,"%lf\n",w[i]);
for (i=0;i<10;i++)
fprintf(fweight_4,"%lf\n",v0[i]);
fprintf(fweight_4,"%lf\n",w0);
t=0;
get_nn_output_4(v,w,v0,w0);
fcloseall();
return ;
}
double *get_nn_input_sensor_4(double t, double *output)
{
double in[8];
// input to NN from previous output of same NN
in[0]=output[0];
// input to NN from previous output of same NN
in[1]=output[1];
//input to NN from output of neighbor sensor 1
in[2]=c14*sensor_1(t);
// input to NN from previous output of neighbor sensor 1
in[3]=c14*sensor_1(t-time_step);
// input to NN from output of neighbor sensor 3
in[4]=c34*sensor_3(t);
// input to NN from previous output of neighbor sensor 3
in[5]=c34*sensor_3(t-time_step);
//input to NN from output of neighbor sensor 5
in[6]=c54*sensor_5(t);
// input to NN from previous output of neighbor sensor 5
in[7]=c54*sensor_5(t-time_step);



return(in);
}


void get_nn_output_4( double v[8][10],double *w, double *v0, double w0
)

{
  FILE  *fsensor_4;

  int i,j;
  double t;
  double *1;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  fsensor_4=fopen("NN_output_4","w");
  // Initialization of output
  output[0]=0;
```

```
  output[1]=0;
  t=0;

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_4(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);



  // Resetting variables
     for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
   output[1]=output[0];
    // Printing results
   fprintf(fsensor_4, "%f        %f
%.16f\n",t,sensor_4(t),output[0]);
  // Resetting the output
   output[0]=0;
   t=t+time_step;
}
  while (t<12.0);
  fcloseall();
  return  ;
}

//NN FOR SENSOR 5
```

```
/* The following function is used for tuning the weights (v's,w's)
     for the first and second layer used in the modified recurrent
     neural network */

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"

using namespace std;

extern FILE  *fweight_5;
void weight_tuning_nn_5()
{
 FILE  *ferror_5;
 FILE  *fweight_5;
int i,j,T;
double t;
double u[8];
double v[8][10];
double v0[10];
double w[10];
double w0;
double *k;
double p1[10],s1[10],sigma[10];

double p2[10];

double output[2];
double E0,SE1[10],E1[8][10],SE,SE0;
ferror_5=fopen("nn_error_5","w");
fweight_5=fopen("nn_weight_5","w");
t=0.0;
T=0;
SE0=0;
for (i=0;i<5;i++)
{

output[0]=0;
output[1]=0;
}



for (i=0;i<10;i++)
{
      p1[i]=0;
      p2[i]=0;
      s1[i]=0;
}



         for (i=0; i<8; i++)
         for (j=0; j<10;j++)
      {
```

```
                    v[i][j] = 0;

    }


    for (i=0;i<10;i++)
    {
            v0[i] =0;

    }

    for (i=0; i<10; i++)
         {
             w[i]=0;

         }

             w0=0;


do
{

       for (i=0;i<10;i++)
{      SE1[i]=0;
}
       t=0;

       do
       {


// Getting inputs from neighboring sensors
k=get_nn_input_sensor_5(t, output);

for(i=0;i<8;i++)
u[i]=*(k+i);



// Resetting variables
for (i=0;i<10;i++)
{
       p1[i]=0;
       p2[i]=0;
       s1[i]=0;
     SE1[i]=0;
}

// Calculating output of the modified recurrent NN during learing phase


for (i=0;i<10;i++)
          for (j=0;j<8;j++)
```

```
                p1[i]=p1[i]+v[j][i]*u[j];


   for (i=0;i<10;i++)
   {
      s1[i]=p1[i]+v0[i];
    sigma[i] = 1.0/(1.0 + exp(-s1[i]));
   }



   for (i=0;i<10;i++)
   {
    p2[i]=w[i]*sigma[i];
    output[0]=output[0]+p2[i];
   }

 output[0]=output[0]+w0;
output[1]=output[0];

 // Calculating error between sensor output and RNN output
E0=sensor_5(t)-output[0];
output[0]=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{
      E1[i][j]=sigma[j]*(1-sigma[j])*E0*w[i];
      SE1[i]=SE1[i]+E1[i][j];
}

 // Updating weights of the second layer according to the error
for (i=0;i<10;i++)
{
      w[i]=w[i]+tuning_rate_2*E0*sigma[i];

}

// Updating weights of the first layer according to the error
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{
v[i][j]=v[i][j]+tuning_rate_1*SE1[j]*u[i];
}
// Calculating the sum of errors
SE0=SE0+E0;



t=t+time_step;

}
while (t<12.0);


cout<<"Modeling Sensor #5 ...";
system("cls");
SE=0.5*SE0*SE0;
```

```
SE0=0;
fprintf(ferror_5,"%d          %.16f\n",T,SE);
T=T+1;
}
// Checking if the error is higher than threshold
while (SE>2.61e-10);

cout<<endl<<"Error is: "<<SE<<endl<<endl;
for (i=0;i<8;i++)
{
   for (j=0;j<10;j++)
   fprintf(fweight_5,"%lf\n",v[i][j]);
}
for (i=0;i<10;i++)
fprintf(fweight_5,"%lf\n",w[i]);
for (i=0;i<10;i++)
fprintf(fweight_5,"%lf\n",v0[i]);
fprintf(fweight_5,"%lf\n",w0);
t=0;
get_nn_output_5(v,w,v0,w0);
fcloseall();
return ;
}




double *get_nn_input_sensor_5(double t, double *output)
{
double in[8];

// input to NN from previous output of same NN
in[0]=output[0];
// input to NN from previous output of same NN
in[1]=output[1];
// input to NN from output of neighbor sensor 1
in[2]=c15*sensor_1(t);
// input to NN from previous output of neighbor sensor 1
in[3]=c15*sensor_1(t-time_step);
//input to NN from output of neighbor sensor 2
in[4]=c25*sensor_2(t);
// input to NN from previous output of neighbor sensor 2
in[5]=c25*sensor_2(t-time_step);
//input to NN from output of neighbor sensor 4
in[6]=c45*sensor_4(t);
// input to NN from previous output of neighbor sensor 4
in[7]=c45*sensor_4(t-time_step);

return(in);
}

void get_nn_output_5( double v[8][10],double *w, double *v0, double w0
)

{
  FILE  *fsensor_5;
```

```
  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  fsensor_5=fopen("NN_output_5","w");
  // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_5(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);



  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
    output[1]=output[0];
    // Printing results
```

```
   fprintf(fsensor_5, "%f        %f
%.16f\n",t,sensor_5(t),output[0]);
  // Resetting the output
   output[0]=0;
   t=t+time_step;
}
  while (t<12.0);
  fcloseall();
  return  ;
}


//NN SENSOR 1 WITH FAILURE

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"

#define drift  0.02
 /*  The following function is used for tuning the weights (v's,w's)
     for the first and second layer used in the modified recurrent
     neural network */

using namespace std;


void failure_detection_sensor_1(FILE *fweight_1)
{
  FILE *ffailure_1;
  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  ffailure_1=fopen("nn_failure_1","w");
  fweight_1=fopen("nn_weight_1","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0.0;//0.1;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
{fscanf(fweight_1,"%lf\n",&v[i][j]);

}
for (i=0;i<10;i++)
fscanf(fweight_1,"%lf\n",&w[i]);

for (i=0;i<10;i++)
fscanf(fweight_1,"%lf\n",&v0[i]);
```

```
fscanf(fweight_1,"%lf\n",&w0);

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_1(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }

   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

  output[0]=output[0]+w0;
  output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"  "<<output[1]<<endl;
   t=t+time_step;
  fprintf(ffailure_1,"%f        %.10f       %.10f\n", t-
time_step,get_output_sensor_1(t-time_step),output[1]);
}
while ((t<12.0)&&(abs(get_output_sensor_1(t-time_step)-
output[1])<0.02));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #1"<<endl;
  fcloseall();
  return;
}

double get_output_sensor_1(double t)
{
      if (t<1.45)
        return(sensor_1(t));
```

```
      else return(sensor_1(t)+t*drift);
}


//NN SENSOR 2 WITH FAILURE

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


// Definition of tuning rates
#define tuning_rate_1 0.1 // First layer
#define tuning_rate_2 0.1 // Second layer

#define c2   0.8
#define c3   0.6
#define c4   0.9

using namespace std;



void failure_detection_sensor_2(FILE *fweight_2)
{

  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  fweight_2=fopen("nn_weight_2","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
fscanf(fweight_2,"%lf\n",&v[i][j]);

for (i=0;i<10;i++)
fscanf(fweight_2,"%lf\n",&w[i]);

for (i=0;i<10;i++)
```

```
fscanf(fweight_2,"%lf\n",&v0[i]);

fscanf(fweight_2,"%lf\n",&w0);

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_2(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"  "<<output[1]<<endl;
   t=t+time_step;
}
  while ((t<12.0)&&(abs(get_output_sensor_2(t-time_step)-
output[1])<0.015));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #2"<<endl;
  fcloseall();
  return  ;
}

double get_output_sensor_2(double t)
{
```

```
      if ((t>4.9)&&(t<5.09))
            return(2.0);
      else return(sensor_2(t));
}

//NN SENSOR 2 WITH FAILURE

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


// Definition of tuning rates
#define tuning_rate_1 0.1 // First layer
#define tuning_rate_2 0.1 // Second layer

#define c2   0.8
#define c3   0.6
#define c4   0.9

using namespace std;

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


// Definition of tuning rates
#define tuning_rate_1 0.1 // First layer
#define tuning_rate_2 0.1 // Second layer

#define c2   0.8
#define c3   0.6
#define c4   0.9

using namespace std;




void failure_detection_sensor_4(FILE *fweight_4)
{

  int i,j;
  double t;
```

```
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  fweight_4=fopen("nn_weight_4","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
fscanf(fweight_4,"%lf\n",&v[i][j]);

for (i=0;i<10;i++)
fscanf(fweight_4,"%lf\n",&w[i]);

for (i=0;i<10;i++)
fscanf(fweight_4,"%lf\n",&v0[i]);

fscanf(fweight_4,"%lf\n",&w0);

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_4(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
```

```
      output[0]=output[0]+pr2[i];
    }

    output[0]=output[0]+w0;
output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"   "<<output[1]<<endl;
   t=t+time_step;
}
  while ((t<12.0)&&(abs(get_output_sensor_4(t-time_step)-
output[1])<0.015));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #4"<<endl;
  fcloseall();
  return  ;
}

double get_output_sensor_4(double t)
{
      if ((t>4.9)&&(t<5.09))
            return(2.0);
      else return(sensor_4(t));
}


void failure_detection_sensor_3(FILE *fweight_3)
{

  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  fweight_3=fopen("nn_weight_3","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
fscanf(fweight_3,"%lf\n",&v[i][j]);

for (i=0;i<10;i++)
fscanf(fweight_3,"%lf\n",&w[i]);

for (i=0;i<10;i++)
fscanf(fweight_3,"%lf\n",&v0[i]);

fscanf(fweight_3,"%lf\n",&w0);
```

```
do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_3(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"  "<<output[1]<<endl;
   t=t+time_step;
}
  while ((t<12.0)&&(abs(get_output_sensor_3(t-time_step)-
output[1])<0.015));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #3"<<endl;
  fcloseall();
  return  ;
}

double get_output_sensor_3(double t)
{
      if ((t>4.9)&&(t<5.09))
            return(2.0);
      else return(sensor_3(t));
}
```

```
//NN for SENSOR 3 with FAILURE

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


// Definition of tuning rates
#define tuning_rate_1 0.1 // First layer
#define tuning_rate_2 0.1 // Second layer

#define c2   0.8
#define c3   0.6
#define c4   0.9

using namespace std;



void failure_detection_sensor_3(FILE *fweight_3)
{

  int i,j;
  double t;
  double *1;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  fweight_3=fopen("nn_weight_3","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
fscanf(fweight_3,"%lf\n",&v[i][j]);

for (i=0;i<10;i++)
fscanf(fweight_3,"%lf\n",&w[i]);

for (i=0;i<10;i++)
fscanf(fweight_3,"%lf\n",&v0[i]);

fscanf(fweight_3,"%lf\n",&w0);
```

```
do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_3(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }



   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"  "<<output[1]<<endl;
   t=t+time_step;
}
  while ((t<12.0)&&(abs(get_output_sensor_3(t-time_step)-
output[1])<0.015));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #3"<<endl;
  fcloseall();
  return  ;
}

double get_output_sensor_3(double t)
{
      if ((t>4.9)&&(t<5.09))
            return(2.0);
      else return(sensor_3(t));
}
```

```
//NN for SENSOR 4 with FAILURE

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


// Definition of tuning rates
#define tuning_rate_1 0.1 // First layer
#define tuning_rate_2 0.1 // Second layer

#define c2   0.8
#define c3   0.6
#define c4   0.9

using namespace std;

void failure_detection_sensor_4(FILE *fweight_4)
{

  int i,j;
  double t;
  double *1;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  fweight_4=fopen("nn_weight_4","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
fscanf(fweight_4,"%lf\n",&v[i][j]);

for (i=0;i<10;i++)
fscanf(fweight_4,"%lf\n",&w[i]);

for (i=0;i<10;i++)
fscanf(fweight_4,"%lf\n",&v0[i]);

fscanf(fweight_4,"%lf\n",&w0);

do
{
```

```
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_4(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);

  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }

   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"  "<<output[1]<<endl;
   t=t+time_step;
}
  while ((t<12.0)&&(abs(get_output_sensor_4(t-time_step)-
output[1])<0.015));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #4"<<endl;
  fcloseall();
  return  ;
}

double get_output_sensor_4(double t)
{
      if ((t>4.9)&&(t<5.09))
            return(2.0);
      else return(sensor_4(t));
}

//NN FOR SENSOR 5 WITH FAILURE

#include <iostream>
```

```
#include <math.h>
#include <stdlib.h>
#include "header.h"


 /* The following function is used for tuning the weights (v's,w'w)
     for the first and second layer used in the modified recurrent
     neural network */


// Definition of tuning rates
#define tuning_rate_1 0.1 // First layer
#define tuning_rate_2 0.1 // Second layer

#define c2   0.8
#define c3   0.6
#define c4   0.9

using namespace std;

void failure_detection_sensor_5(FILE *fweight_5)
{

  int i,j;
  double t;
  double *l;
  double output[2];
  double pr1[10];
  double su1[10];
  double pr2[10];
  double s[8];
  double sigm[10];
  double v[8][10],w[10],v0[10],w0;
  fweight_5=fopen("nn_weight_5","r");
    // Initialization of output
  output[0]=0;
  output[1]=0;
  t=0;
for (i=0;i<8;i++)
for (j=0;j<10;j++)
fscanf(fweight_5,"%lf\n",&v[i][j]);

for (i=0;i<10;i++)
fscanf(fweight_5,"%lf\n",&w[i]);

for (i=0;i<10;i++)
fscanf(fweight_5,"%lf\n",&v0[i]);

fscanf(fweight_5,"%lf\n",&w0);

do
{
  // Getting input from neighboring sensors
  l=get_nn_input_sensor_5(t, output);
  for(i=0;i<8;i++)
  s[i]=*(l+i);
```

```
  // Resetting variables
      for (i=0;i<10;i++)
{
      pr1[i]=0;
      pr2[i]=0;
      su1[i]=0;
}
 // Calculating the output of the neural network during the testing
phase
   for (i=0;i<10;i++)
         for (j=0;j<8;j++)
               pr1[i]=pr1[i]+v[j][i]*s[j];


   for (i=0;i<10;i++)
   {
      su1[i]=pr1[i]+v0[i];
    sigm[i] = 1.0/(1.0 + exp(-su1[i]));
   }

   for (i=0;i<10;i++)
   {
     pr2[i]=w[i]*sigm[i];
     output[0]=output[0]+pr2[i];
   }

   output[0]=output[0]+w0;
output[1]=output[0];
  // Resetting the output
  output[0]=0;
  cout<<endl<<t<<"  "<<output[1]<<endl;
   t=t+time_step;
}
 while ((t<12.0)&&(abs(get_output_sensor_5(t-time_step)-
output[1])<0.015));
        if (t<12.0)
              cout<<endl<<endl<<"time= "<<t-time_step<<"     FAILURE
DETECTED IN SENSOR #5"<<endl;
  fcloseall();
  return  ;
}

double get_output_sensor_5(double t)
{
      if ((t>4.9)&&(t<5.09))
            return(2.0);
      else return(sensor_5(t));
}
```

APPENDIX B

SIMULATION SOURCE CODE USING

MATLAB 7.1

```
//SIMULATION CODE USING MATLAB 7.1

//conf fac =1


%Real data for five temperature sensors
s1 = [43 41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47];
s_1 = [41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47 43];

s2 = [44 41 45 43 42 40 43 43 45 44 48 52 56 60 62 62 62 61 56 49 48 48
45 49 50 50 48 52 52 51 50 52 50 55 57 60 64 67 70 72 72 71 60 57 53 55
55 53 52 51 51 48 47 45 45 44 43 47 52 55 60 62 66 67 68 65 62 57 52 50
50 48 45];
s_2 = [41 45 43 42 40 43 43 45 44 48 52 56 60 62 62 62 61 56 49 48 48
45 49 50 50 48 52 52 51 50 52 50 55 57 60 64 67 70 72 72 71 60 57 53 55
55 53 52 51 51 48 47 45 45 44 43 47 52 55 60 62 66 67 68 65 62 57 52 50
50 48 45 44];

s3 = [42 41 43 43 41 40 42 42 40 40 45 52 57 55 60 62 64 64 64 59 56 53
50 47 46 46 45 47 48 48 47 43 46 48 52 60 62 68 70 70 71 71 70 66 63 60
55 53 49 49 48 45 45 41 42 39 40 47 54 59 63 65 68 67 67 66 65 54 53 50
50 48 46];
s_3 = [41 43 43 41 40 42 42 40 40 45 52 57 55 60 62 64 64 64 59 56 53
50 47 46 46 45 47 48 48 47 43 46 48 52 60 62 68 70 70 71 71 70 66 63 60
55 53 49 49 48 45 45 41 42 39 40 47 54 59 63 65 68 67 67 66 65 54 53 50
50 48 46 42];

s4 = [42 40 45 46 47 44 45 42 44 44 43 45 52 54 55 57 59 57 57 55 49 46
45 49 49 52 51 52 53 52 51 51 50 53 54 57 60 62 64 65 66 68 67 54 57 55
55 52 55 50 50 52 50 50 46 46 44 53 55 58 59 60 62 62 62 61 55 52 48 46
50 48 45];
s_4 = [40 45 46 47 44 45 42 44 44 43 45 52 54 55 57 59 57 57 55 49 46
45 49 49 52 51 52 53 52 51 51 50 53 54 57 60 62 64 65 66 68 67 54 57 55
55 52 55 50 50 52 50 50 46 46 44 53 55 58 59 60 62 62 62 61 55 52 48 46
50 48 45 42];

s5 = [39 40 41 40 38 39 39 38 38 39 46 53 57 60 61 62 62 60 53 51 46 45
43 41 40 39 38 36 36 40 48 39 49 57 62 64 66 68 70 71 72 69 62 59 54 51
50 48 48 46 46 43 43 40 39 42 41 48 53 60 62 65 65 66 67 64 60 54 52 50
48 47 47];
s_5 = [40 41 40 38 39 39 38 38 39 46 53 57 60 61 62 62 60 53 51 46 45
43 41 40 39 38 36 36 40 48 39 49 57 62 64 66 68 70 71 72 69 62 59 54 51
50 48 48 46 46 43 43 40 39 42 41 48 53 60 62 65 65 66 67 64 60 54 52 50
48 47 47 39];

%Setting up the neural network learning session
p=[s1; s_1; s2; s_2; s3; s_3; s4; s_4]./10;

net=newff(minmax(p),[10,1],{'logsig','purelin'},'traingd');
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
```

```
net.trainParam.min_grad = 1e-30;
net.trainParam.epochs = 100000;
net.trainParam.goal = 2e-4;

%training the network
%[net,tr]=train(net,p,s1./10);

%calculating the model
%a1=sim(net,p)*10;
%a1

u = [43 41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47]';
t=1:73;

%KALMAN FILTERING TECHNIQUE
A=0;
B=1;
C=1;
n=73;

randn('seed',0)

Q = 10 ; R = 1;

w = sqrt(Q)*randn(n,1);
v = 0*sqrt(R)*randn(n,1);

%Building sensor as a system

Plant = ss(0,[1 1],1,0,-1,'inputname',{'u' 'w' },'outputname','y');

%Building Kalaman Filter model

[kalmf,L,P,M] = kalman(Plant,Q,R);
kalmf = kalmf(1,:);
kalmf

a = A;
b = [B B0 0*B];
c = [C;C];
d = [0 0 0;0 0 1];
P = ss(a,b,c,d,-1,'inputname',{'u' 'w' 'v'},'outputname',{'y' 'yv'});
sys = parallel(P,kalmf,1,1,[],[])
%Close loop around input #4 and output #2
SimModel = feedback(sys,1,4,2,1)

%Delete yv from I/O list
SimModel = SimModel([1 3],[1 2 3])

SimModel.inputname

SimModel.outputname

[out,x] = lsim(SimModel,[w,v,u]);
```

```
y = out(:,1);    % true response
ye = out(:,2);   % filtered response
yv = y + v;      % measured response

ye(1)=s1(1);
y(1)=s1(1);

for i=1:72
ye(i)=ye(i+1);
y(i)=y(i+1);
end
ye(73)=s1(73);
y(73)=s1(73);
%and compare the true and filtered responses graphically.
%subplot(211), plot(t,y,'--',t,ye,'-'),
%xlabel('No. of samples'), ylabel('Output')
%title('Kalman filter response')
%subplot(212), plot(t,y-yv,'-.',t,y-ye,'-'),
%xlabel('No. of samples'), ylabel('Error')
%ye=ye+5;
%subplot(211)
plot(s1,'bl');
%hold on
%plot(a1,'b');
hold on
plot(ye,'r');
%hold on
%plot(y,'g');
%subplot(212),
%plot(s1-a1,'b');
%hold on
%plot(s1-ye','r');

//conf fac <1


s1 = [43 41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47];
s_1 = [41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47 43];

s2 = [44 41 45 43 42 40 43 43 45 44 48 52 56 60 62 62 62 61 56 49 48 48
45 49 50 50 48 52 52 51 50 52 50 55 57 60 64 67 70 72 72 71 60 57 53 55
55 53 52 51 51 48 47 45 45 44 43 47 52 55 60 62 66 67 68 65 62 57 52 50
50 48 45];
s_2 = [41 45 43 42 40 43 43 45 44 48 52 56 60 62 62 62 61 56 49 48 48
45 49 50 50 48 52 52 51 50 52 50 55 57 60 64 67 70 72 72 71 60 57 53 55
55 53 52 51 51 48 47 45 45 44 43 47 52 55 60 62 66 67 68 65 62 57 52 50
50 48 45 44];

s3 = [42 41 43 43 41 40 42 42 40 40 45 52 57 55 60 62 64 64 64 59 56 53
50 47 46 46 45 47 48 48 47 43 46 48 52 60 62 68 70 70 71 71 70 66 63 60
```

```
55 53 49 49 48 45 45 41 42 39 40 47 54 59 63 65 68 67 67 66 65 54 53 50
50 48 46];
s_3 = [41 43 43 41 40 42 42 40 40 45 52 57 55 60 62 64 64 64 59 56 53
50 47 46 46 45 47 48 48 47 43 46 48 52 60 62 68 70 70 71 71 70 66 63 60
55 53 49 49 48 45 45 41 42 39 40 47 54 59 63 65 68 67 67 66 65 54 53 50
50 48 46 42];

s4 = [42 40 45 46 47 44 45 42 44 44 43 45 52 54 55 57 59 57 57 55 49 46
45 49 49 52 51 52 53 52 51 51 50 53 54 57 60 62 64 65 66 68 67 54 57 55
55 52 55 50 50 52 50 50 46 46 44 53 55 58 59 60 62 62 62 61 55 52 48 46
50 48 45];
s_4 = [40 45 46 47 44 45 42 44 44 43 45 52 54 55 57 59 57 57 55 49 46
45 49 49 52 51 52 53 52 51 51 50 53 54 57 60 62 64 65 66 68 67 54 57 55
55 52 55 50 50 52 50 50 46 46 44 53 55 58 59 60 62 62 62 61 55 52 48 46
50 48 45 42];

s5 = [39 40 41 40 38 39 39 38 38 39 46 53 57 60 61 62 62 60 53 51 46 45
43 41 40 39 38 36 36 40 48 39 49 57 62 64 66 68 70 71 72 69 62 59 54 51
50 48 48 46 46 43 43 40 39 42 41 48 53 60 62 65 65 66 67 64 60 54 52 50
48 47 47];
s_5 = [40 41 40 38 39 39 38 38 39 46 53 57 60 61 62 62 60 53 51 46 45
43 41 40 39 38 36 36 40 48 39 49 57 62 64 66 68 70 71 72 69 62 59 54 51
50 48 48 46 46 43 43 40 39 42 41 48 53 60 62 65 65 66 67 64 60 54 52 50
48 47 47 39];
c21=0.8;
c31=0.6;
c41=0.95;
p=[s1; s_1; c21*s2; c21*s_2; c31*s3; c31*s_3; c41*s4; c41*s_4]./10;

net=newff(minmax(p),[10,1],{'logsig','purelin'},'traingd');

%net=init(net);
%net.trainParam.show = 50;

net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.min_grad = 1e-30;
net.trainParam.epochs = 100000;
net.trainParam.goal = 2e-2;
[net,tr]=train(net,p,s1./10);
a1=sim(net,p)*10;
a1
u = [43 41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47]';
t=1:73;
A=0;
B=1;
C=1;
n=73;
randn('seed',0)

Q = 3 ; R = 1;
w = sqrt(Q)*randn(n,1);
v = sqrt(R)*randn(n,1);
```

```
Plant = ss(0,[1 1],1,0,-1,'inputname',{'u' 'w'},'outputname','y');
[kalmf,L,P,M] = kalman(Plant,Q,R);
kalmf = kalmf(1,:);
kalmf
a = A;
b = [B B 0*B];
c = [C;C];
d = [0 0 0;0 0 1];
P = ss(a,b,c,d,-1,'inputname',{'u' 'w' 'v'},'outputname',{'y' 'yv'});

sys = parallel(P,kalmf,1,1,[],[])

%Close loop around input #4 and output #2
SimModel = feedback(sys,1,4,2,1)

%Delete yv from I/O list
SimModel = SimModel([1 3],[1 2 3])

SimModel.inputname

SimModel.outputname

[out,x] = lsim(SimModel,[w,v,u]);

y = out(:,1);   % true response
ye = out(:,2);  % filtered response
yv = y + v;     % measured response

ye(1)=s1(1);

%and compare the true and filtered responses graphically.
%subplot(211), plot(t,y,'--',t,ye,'-'),
%xlabel('No. of samples'), ylabel('Output')
%title('Kalman filter response')
%subplot(212), plot(t,y-yv,'-.',t,y-ye,'-'),
%xlabel('No. of samples'), ylabel('Error')
%{
subplot(211),plot(s1,'bl');
hold on
plot(a1,'b');
hold on
plot(ye,'r');
subplot(212),
%plot (s1,'b');
plot(s1-a1,'b');
hold on
plot(s1-ye','r');

%}

//WITH DRIFT
t=3:73;
s1 = [43 41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47];
```

```
s_1 = [41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47 43];

s2 = [44 41 45 43 42 40 43 43 45 44 48 52 56 60 62 62 62 61 56 49 48 48
45 49 50 50 48 52 52 51 50 52 50 55 57 60 64 67 70 72 72 71 60 57 53 55
55 53 52 51 51 48 47 45 45 44 43 47 52 55 60 62 66 67 68 65 62 57 52 50
50 48 45];
s_2 = [41 45 43 42 40 43 43 45 44 48 52 56 60 62 62 62 61 56 49 48 48
45 49 50 50 48 52 52 51 50 52 50 55 57 60 64 67 70 72 72 71 60 57 53 55
55 53 52 51 51 48 47 45 45 44 43 47 52 55 60 62 66 67 68 65 62 57 52 50
50 48 45 44];

s3 = [42 41 43 43 41 40 42 42 40 40 45 52 57 55 60 62 64 64 64 59 56 53
50 47 46 46 45 47 48 48 47 43 46 48 52 60 62 68 70 70 71 71 70 66 63 60
55 53 49 49 48 45 45 41 42 39 40 47 54 59 63 65 68 67 67 66 65 54 53 50
50 48 46];
s_3 = [41 43 43 41 40 42 42 40 40 45 52 57 55 60 62 64 64 64 59 56 53
50 47 46 46 45 47 48 48 47 43 46 48 52 60 62 68 70 70 71 71 70 66 63 60
55 53 49 49 48 45 45 41 42 39 40 47 54 59 63 65 68 67 67 66 65 54 53 50
50 48 46 42];

s4 = [42 40 45 46 47 44 45 42 44 44 43 45 52 54 55 57 59 57 57 55 49 46
45 49 49 52 51 52 53 52 51 51 50 53 54 57 60 62 64 65 66 68 67 54 57 55
55 52 55 50 50 52 50 50 46 46 44 53 55 58 59 60 62 62 62 61 55 52 48 46
50 48 45];
s_4 = [40 45 46 47 44 45 42 44 44 43 45 52 54 55 57 59 57 57 55 49 46
45 49 49 52 51 52 53 52 51 51 50 53 54 57 60 62 64 65 66 68 67 54 57 55
55 52 55 50 50 52 50 50 46 46 44 53 55 58 59 60 62 62 62 61 55 52 48 46
50 48 45 42];

s5 = [39 40 41 40 38 39 39 38 38 39 46 53 57 60 61 62 62 60 53 51 46 45
43 41 40 39 38 36 36 40 48 39 49 57 62 64 66 68 70 71 72 69 62 59 54 51
50 48 48 46 46 43 43 40 39 42 41 48 53 60 62 65 65 66 67 64 60 54 52 50
48 47 47];
s_5 = [40 41 40 38 39 39 38 38 39 46 53 57 60 61 62 62 60 53 51 46 45
43 41 40 39 38 36 36 40 48 39 49 57 62 64 66 68 70 71 72 69 62 59 54 51
50 48 48 46 46 43 43 40 39 42 41 48 53 60 62 65 65 66 67 64 60 54 52 50
48 47 47 39];

p=[s1; s_1; s2; s_2; s3; s_3; s4; s_4]./10;

net=newff(minmax(p),[10,1],{'logsig','purelin'},'traingd');

%net=init(net);
%net.trainParam.show = 50;

net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.min_grad = 1e-30;
net.trainParam.epochs = 100000;
net.trainParam.goal = 2e-2;
[net,tr]=train(net,p,s1./10);
a1=sim(net,p)*10;
a1
drift=0.2*(t-3);  % linear drift
```

```
fs1=s1+drift; % output of faulty sensor #1


u = [43 41 40 40 38 38 38 40 42 51 55 61 59 61 62 61 60 56 52 50 48 46
45 44 47 43 42 53 52 52 53 54 57 60 63 67 71 72 72 72 72 64 60 55 54 53
51 50 47 49 45 45 42 40 40 40 41 48 57 61 62 65 66 67 68 65 61 59 55 49
49 50 47]';

A=0;
B=1;
C=1;
n=73;
randn('seed',0)

Q = 2 ; R = 1;
w=0*u;%w = sqrt(Q)*randn(n,1);
v = sqrt(R)*randn(n,1);

Plant = ss(0,[1 0],1,0,-1,'inputname',{'u' 'w'},'outputname','y');
[kalmf,L,P,M] = kalman(Plant,Q,R);
kalmf = kalmf(1,:);
kalmf
a = A;
b = [B 0 0*B];
c = [C;C];
d = [0 0 0;0 0 1];
P = ss(a,b,c,d,-1,'inputname',{'u' 'w' 'v'},'outputname',{'y' 'yv'});

sys = parallel(P,kalmf,1,1,[],[])

%Close loop around input #4 and output #2
SimModel = feedback(sys,1,4,2,1)

%Delete yv from I/O list
SimModel = SimModel([1 3],[1 2 3])

SimModel.inputname

SimModel.outputname

[out,x] = lsim(SimModel,[w,v,u]);

y = out(:,1);   % true response
ye = out(:,2);  % filtered response
yv = y + v;     % measured response

ye(1)=s1(1);

%and compare the true and filtered responses graphically.
%subplot(211), plot(t,y,'--',t,ye,'-'),
%xlabel('No. of samples'), ylabel('Output')
%title('Kalman filter response')
%subplot(212), plot(t,y-yv,'-.',t,y-ye,'-'),
%xlabel('No. of samples'), ylabel('Error')
for i=1:73
    if abs(fs1(i)-s1(i))>2
        disp('error in sensor at hour')
```

```
        disp(i)
        break
    end
end
subplot(211),
plot(s1,'b');
hold on
plot(fs1,'r');
```

BIBLIOGRAPHY

[1] Crossbow, http://www.xbow.com

[2] Xiao Di, B. K. Gosh, Xi Ning, and T. J. Tarn. "Sensor-based hybrid position/force control of a robot manipulator in an uncalibrated environment," *IEEE Transactions on Control Systems Technology.* vol. 8, no. 4, pp. 635-645, July 2000.

[3] S. E. Lysheyski, "Smart flight control surfaces with microelectromechanical systems," *Aerospace and Electronic Systems, IEEE Transactions on Control Systems Technology*, vol. 38, no. 2, pp. 543-552, April 2002.

[4] S. Katsura, Y. Matsumoto, and K. Ohnishi. "Analysis and experimental validation of force bandwidth for force control," *2003 IEEE International Conference on Control Systems Technology*, December 2003, pp. 796-801.

[5] Pouliezos A. D. and G. S. Stavrakankis, *Real Time Fault Monitoring of Industrial Processes*, Kulwer Academic Publishers, 1994.

[6] Zhirabok and O. V. Preobragenskaya, "Instrument fault detection in nonlinear dynamic systems," *in Proceedings on Systems, Man and Cybernetics*, 1993.

[7] C. B Weinstock, W. L Heirnerdinger, H. Ihara, S. B. Johnson, H. D. Kirrmann, J. J. Stiffler, and L. Yount, "The state of the practice in fault tolerant systems," *22nd international Symposium on Fault-Tolerant Computing*, July 1992.

[8] S. C. Lee, "Sensor Value Validation Based on Systematic Exploration of the sensor Redundancy for Fault Diagnosis", *IEEE Transactions on systems, Man and Cybernetics,* vol.24, no. 4, pp. 594-605, April 1994.

[9] M. L. Leushen, J. R. Cavallaro, and I. D. Walker, "Robotic fault detection using analytical redundancy", *in Proceedings IEEE Conference on Robotics and Automation*, 2002, pp. 456-463.

[10] K. Narendra and P. Gallman, "An iterative method for the identification of nonlinear systems using a Hammerstein model", *IEEE Transactions on Automatic Control,* vol. 11, no. 3, pp. 546-550, July 1966.

[11] R. Haber, "Parametric identification of nonlinear dynamic systems based on nonlinear crosscorrelation functions", *Control Theory and Applications, IEEE Proceedings*, vol. 135, no. 6, November 1988, pp. 405-420.

[12] A. Lo Shiavo and A. M. Luciano, "Powerful and flexible fuzzy algorithm for nonlinear dynamic system identification", *IEEE Transactions on Fuzzy Systems*, vol. 9, no. 6, pp. 828-835, December 2001.

[13] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks", *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4-27, March. 1990.

[14] S. Haykin, *Neural Networks – A Comprehensive Foundation* (second ed.), Prentice-Hall, Upper Saddle River, NJ, 1998.

[15] F. L. Lewis, S. Jagannathan, and A. Yesilidrek, *Neural Network Control of Robot Manipulators and Nonlinear Systems*, Taylor and Francis, Philadelphia, PA, 1999.

[16] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, pp. 303- 314, 1989.

[17] K. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural Networks*, 2. 183-192.

[18] H. K. Khalil, *Nonlinear Systems*, 3rd edition, Prentice Hall, Upper Saddle River, New Jersey, 2002.

[19] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli, "Fault tolerance techniques for wireless ad hoc sensor networks," *in Proceedings IEEE Conference on Sensors*, 2002, pp. 1491-1496.

[20] Dennis S. Bernstein, "Sensor Performance Specifications", *IEEE Control Systems Magazine*, pp 9-18, August 2001.

[21] A. Bernieri, M. D'Apuzzo, L. Sansone, and M. Savastano, "A neural network approach for identification and fault diagnosis on dynamic systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 43, no. 6, pp. 867-873, December 1994.

[22] A. Bernieri, G. Betta, A. Pietrosanto, and C. Sansone, "A neural network approach to instrument fault detection and isolation", *in Proceedings IEEE Conference on Instrumentation and Measurement*, 1994, pp. 139-144.

[23] R. S. Ahmed, "Identification of nonlinear dynamic systems using a rapid neural network", *the 27th Annual Conference of the IEEE,* vol. 3, December 2001, pp. 1734-1739.

[24] S. Straub and D. Shroder, "Identification of nonlinear dynamic systems with recurrent neural networks and Kalman filter methods", *1996 IEEE International Symposium,* 12-15, May 1996, pp. 341-344.

[25] Harvard University. "Code Blue: Wireless Sensor Networks for Medical Care." http://www.eecs.harvard.edu/~mdw/proj/codeblue.

[26] M. Holler Ed. "Camalie Net: Wireless Sensor Network at Camalie Vineyards Mt. Veeder, Napa Valley, California." http://camalie.com/WirelessSensing/WirelessSensors.

[27] UC Berkeley. "Structural Health Monitoring of the Golden Gate Bridge". http://www.cs.berkeley.edu/~binetude/ggb.

[28] S. C. Lee, "Sensor Value Validation Based On Systematic Exploration of the Sensor Redundancy For Fault Diagnosis", *IEEE Transactions on Systems, Man and Cybernetics,* vol.24, no. 4, pp. 594-605, April 1994.

[29] M. L. Leushen, J. R. Cavallaro, and I. D. Walker, "Robotic Fault Detection Using Analytical Redundancy*", in Proceedings IEEE Conference on Robotics and Automation*, 2002, pp. 456-463.

[30] Moteiv Corporation. "Tmote Sky Datasheet." V1.04. November 2006. http://www.moteiv.com.

[31] Chipcon. "SmartRF CC2420 datasheet." V1.2. June 2004. http://www.chipcon.com.

[32] Sensirion. SHT1x/SHT7x Datasheet. V2.0. March 2003 www.sensirion.com.

[33] UC Berkley, www.TinyOS.net.

[34] P. Levis."TinyOS Programming." June 2006. www.TinyOS.net.

[35] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo. "The Collection Tree Protocol (CTP)". *TinyOS Extension Proposal 123. V1.8.* August 2006.

[36] Y Chraibi. "Localization in Wireless Sensor Networks." *KTH Signal Sensors and Systems.* Stockholm, Sweden. 2005.

[37] Gelb A., "Applied optimal estimation*". MIT Press*, 1974.

[38] Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME - Journal of Basic Engineering* Vol. 82: pp. 35-45 (1960).

[39] Kalman, R. E. and Bucy R. S., "New Results in Linear Filtering and Prediction Theory", *Transactions of the ASME - Journal of Basic Engineering* Vol. 83: pp. 95-107 (1961).

[40] Julier, Simon J. and Jeffery K. Uhlmann. "A New Extension of the Kalman Filter to nonlinear Systems." In *The Proceedings of AeroSense: The 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls, Multi Sensor Fusion, Tracking and Resource Management* II, SPIE, 1997.

[41] Harvey, "A.C. Forecasting, Structural Time Series Models and the Kalman Filter". *Cambridge University Press, Cambridge*, 1989.