5-15-2009

Resilient Average and Distortion Detection in Sensor Networks

Ricardo Aguirre Jurado
University of New Orleans

Resilient Average and Distortion Detection in Sensor Networks

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Ricardo Aguirre Jurado

B.S. University of New Orleans, 2006

May, 2009

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

In this paper a resilient sensor network is built in order to lessen the effects of a small portion of corrupted sensors when an aggregated result such as the average needs to be obtained. By examining the variance in sensor readings, a change in the pattern can be spotted and minimized in order to maintain a stable aggregated reading. Offset in sensors readings are also analyzed and compensated to help reduce a bias change in average. These two analytical techniques are later combined in Kalman filter to produce a smooth and resilient average given by the readings of individual sensors. In addition, principal components analysis is used to detect variations in the sensor network. Experiments are held using real sensors called MICAz, which are use to gather light measurements in a small area and display the light average generated in that area.

Sensors, Correlation, Kalman Filter, Principal Component Analysis, PCA

INTRODUCTION

Redundant sensors are mainly used in applications where critical information needs to be present at all times. If a sensor fails to send that critical data then other sensors can still send the same valuable data, in this case it can be inferred that the availability of that information has been compromised. Sometimes reliability is increased when different kinds of sensors are used to measured the same phenomena, in this case , one sensor can be more accurate that the other but more prone to failure in certain situations. In contrast the other sensor may not be too accurate, but it may have a more robust implementation that will allow it to keep streaming information in harsh conditions. One example of this redundant behavior can be seen in aircraft navigation systems where a global positioning system (GPS) and an Inertia navigation system is (INS) are used in conjunction to accurately obtain the position of the aircraft. The reliability issue of those systems consist in that a GPS signal can be lost due to environmental conditions or satellite outage, the INS, in the other hand, does not depend on those external sources since the position is estimated in function of the relative past position of the aircraft. The combinations of both systems increase the accuracy in determining the exact position since GPS is mainly use to update INS results. However, in the case a GPS signal is lost, the position can still be predicted by using only INS. If the same type of sensors with the same accuracy, reliability and limitation are used as a redundant means to measure data, more sensors are required. By increasing the number of sensors measuring the same event we are assuring that the data will be available even if the probability of failure of any sensor is high. This is one of the main reasons wireless sensors are used by the military to gather information of the surroundings. Most of the time sensors need to be deployed in wide enemy areas to measure possible threats such as toxic gases or just to measure the temperature, humidity or pressure of the place. Multiple sensors can be deployed and those that are close to each other usually provide the same redundant measurement of the environmental condition. By increasing redundancy, the sensors do not have to be too reliable or even too accurate. Lack of reliability and a high probability of failure of the sensors can be overcome with the fact that hundreds of those devices will be used. If a random number of sensors fail, the information can still be captured and analyzed from the rest of the sensor population. In addition, accuracy is sometimes degraded by the use of numerous inexpensive and unreliable sensors, but by combining information of several redundant sensors a good estimate of the real value can be obtained. The redundant data can be passed through aggregated functions such as the average, the median or the mode, to get a unique value that represents what is being measured by the redundant sensors. This aggregated function depends on all the values provided by the redundant sensors, those values are assumed to be correlated between each other since the sensors are affected by the same physical event. As a result, the final output of the aggregated function will vary if that correlation assumption is lost. If a sensor is not well calibrated or if it is showing a random behavior different than the other sensors, then a method to maintain an aggregated result as close as the true value needs to be implemented.   Sensors that contribute corrupted information can be easily spotted by comparing their value with the rest of the sensors and making a decision based on majority consensus on which values should be part of the aggregated function. However, if hundred of sensors are being used, then comparing each sensor value would result in an increase in overhead and time delay. A resilient average with network detection is proposed to ease the effect of corrupted data in an aggregated function such as the average.

These wireless sensors area characterized for being inexpensive and not too accurate, thus readings from all the sensors can be used collectively to describe their environment. Sometimes this outcome does not represent a correct aggregated result as the average when a sensor starts to send faulty readings that can really impact a final output. In order to correct this problem sensors readings are analyzed and filtered using a sliding window of twenty average readings. In this way, any distortion caused by a corrupted sensor can be compensated.

The analysis of the twenty average readings is done by calculating and storing the variance of each reading, where a change in variance could mean a change in a sensor pattern. Even if all sensors follow the same pattern, it will not give the right average if one of those has a bias error. That error might be caused by an offset in a corrupted sensor reading and it is most likely that the sensor is not properly calibrated. By knowing the true mean of the sensors, this bias can be minimized and adjusted to match the sensor average. Since the true mean is not known before hand, each individual sensor is compared to the mean of the rest in order to penalize readings that are farther from the network mean.

After the readings have been analyzed, a Kalman filter[1] is used to estimate the value that should be produced in case the average is distorted by a corrupted sensor. The Kalman filter is a mathematical tool that can be used to remove undesired noise in measurements that are being estimated. It is based on the idea of favoring the estimated value if the measured value contains noise.

---

[1] http://cs.unc.edu/~welch/media/pdf/kalman_intro.pdf
Eli Brookner .Tracking and Kalman Filtering Made Easy.Copyright # 1998 John Wiley & Sons, Inc.
Zarachan, Paul and Howard Musoff. Fundamentuls of Kalman Filtering:A P racfical Approach. Progress in Astronautics and Aeronautics, American Institute of Aeronautics and Astronautics, Inc., 2000.

For example, an equation that gives an updated estimate of a state as weighted sum of two other estimates can be defined as:

$$x_{n+1} = x_n(1 - g_n) + y_n g_n$$

Where x_n is an estimate based on measurement made a time n and y_n is the current measurement made at time n. It can be noticed that in the above equation depending on the selection of g_n the estimate x_{n+1} will be closer to x_n or y_n. If g_n is close to one, the estimate would be equal to the actual measurement y_n and if a small value of g_n is used, the previous estimate will have more weight than the current measurement. The last equation can be written as

$$x_{n+1} = x_n + g_n(y_n - x_n)$$

And the current measurement is given by:

$$y_n = x_n + v_n$$

Where v_n is the measurement error of the n^{th} observation and x_n is the actual measurement.

The Kalman filter uses previous estimates in order to obtain a new estimate that has a minimum variance. The following equation shows minimum variance estimate.

$$x_{n+1} = \left[\frac{x_n}{VAR(x_n)} + \frac{y_n}{VAR(y_n)}\right]\frac{1}{1/VAR(x_n) + 1/VAR(y_n)}$$

In the case where the two estimates y_n and x_n are equally truthful, the last equation can be written as:

$$x_{n+1} = \frac{x_n + y_n}{2}$$

Thus, the result of the combine weights is just their average. In the case were the accuracy of x_n is much greater than y_n the new estimate can be approximated as:

$$x_{n+1} = x_n$$

The above examples show the important role that the variances of the measurements play in order to favor the accuracy of the current measurements or past predictions.

Hence, the minimum variance equation can be rewritten as:

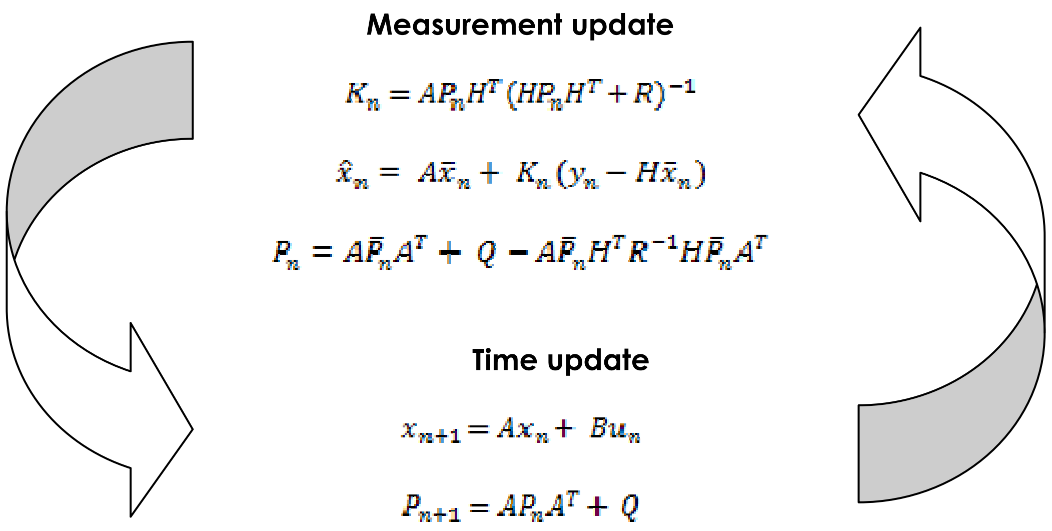
$$x_{n+1} = x_n + \frac{VAR(x_n)}{VAR(y_n)}(y_n - x_n)$$

Where $\frac{VAR(x_n)}{VAR(y_n)}$ is the gain that favors the actual measurement or the estimate. In a Kalman filter, this gain is named *K* and depends on the noise and error covariance.

A Kalman filter combines present and future measurement to predict the state of a process. It can be applied sequentially to a discrete set of measurements. The estimation of measurement is achieved by using a feedback control or noisy measurements and outputting an estimate measurement. This procedure can be divided in two groups: time update equations and measurement update equations.

The first group of equations is in charge of predicting the current state estimate ahead in time. And the second group of equations corrects the projected estimate by an actual measurement at that time, thus minimizing a distortion.

The two groups of equations used in a Kalman filter are the following



Measurement update

$$K_n = AP_nH^T(HP_nH^T+R)^{-1}$$

$$\hat{x}_n = A\bar{x}_n + K_n(y_n - H\bar{x}_n)$$

$$P_n = A\bar{P}_nA^T + Q - A\bar{P}_nH^TR^{-1}H\bar{P}_nA^T$$

Time update

$$x_{n+1} = Ax_n + Bu_n$$

$$P_{n+1} = AP_nA^T + Q$$

Where *x* is the estimate measurement, *A* is called the state transition matrix, *P* is the error covariance, *R* is the measurement noise covariance, *H* is the observation matrix and *Q* is the process noise covariance.

The transition matrix A can be defined as:

$$A = \begin{bmatrix} 1 & Ts \\ 0 & 1 \end{bmatrix}$$

This is the transition matrix for a first order Kalman where measurements are taken every *Ts* seconds.

The observation matrix H is given by:

$$H = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

This is the observation matrix for a first order Kalman where measurements are taken every *Ts* seconds.

The measurement noise covariance can be defined as:

$$R = E[zz^t]$$

Where *z* is the noise vector added to the measurement.

The process noise covariance can be defined as:

$$Q = E[uu^t]$$

Where *u* is white noise vector added in order to tune the filter.

The distortions caused in the sensor network can be detected by using another mathematical tool called principal component analysis[1]. This statistical technique analyzes uncorrelated sets of related data and transforming it to low-dimension projection of orthogonal variables. These new variables are called principal components and are a linear combination representation of the original data. The coefficients of the linear transformation provide the relative contribution to the variance of the principal component. Principal components analysis has been mostly used to solve dimension reduction and visualization, blind source separation problems. Dimension reduction in data compression or pattern recognition is usually accomplished by eliminating the lower principal components since they account for the less variance. However, the upper principal components can also be interpreted and valuable information about the relationship between data sets can be inferred. If data sets are assumed to be correlated then any variation in the data could be explained by analyzing the different principal components.

The fundamental idea behind the method of principal components is to reduce a covariance matrix **S** to a diagonal matrix **L** by premultiplying and postmultiplying by particular orthonormal matrix **U** such that:

[1] J. E. Jackson. A User's Guide To Principal Components. John Wiley & Sons, 1991.
T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer-Verlag, 2001.

$$U'SU = L$$

Where **S** is composed by the covariance between centered data sets **x_1,x_2…x_p,** that corresponds to each column of matrix **X** such that:

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots x_{1p} \\ x_{21} & x_{22} & \dots x_{2p} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \dots x_{np} \end{bmatrix}$$

From this,

$$S = X'X/(n-1)$$

$$S = \begin{bmatrix} s_1{}^2 & s_{12} & \dots s_{1p} \\ s_{21} & s_2{}^2 & \dots s_{2p} \\ \vdots & \vdots & \vdots \\ s_{p1} & s_{p2} & \dots s_{pp}{}^2 \end{bmatrix}$$

Thus, s_i^2 is the variance of the ith random variables , **x_i** , and s_{ij} is the covariance between the ith and the jth random variables.

The columns of U, u_1,u_2,….u_p are the eigenvectors of **S** and satisfy the condition:

$$u_i u_j' = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

The diagonal elements o f **L ,** l_1, l_2,…,l_p are called the eigenvalues of **S** and can be obtained from the solution of:

$$|S - lI| = 0$$

In the same way the eigenvectors can be obtained by:

$$[\boldsymbol{S} - l\boldsymbol{I}]\boldsymbol{t}_i = 0$$

and

$$\boldsymbol{u}_i = \frac{\boldsymbol{t}_i}{\sqrt{\boldsymbol{t}_i'.\boldsymbol{t}_i}}$$

Hence, the principal components are represented by:

$$\boldsymbol{z}_{i=}\boldsymbol{u}_i'\boldsymbol{X}, \qquad (\boldsymbol{i} = \boldsymbol{1}, \boldsymbol{2} \ldots \boldsymbol{p})$$

Meaning that the original vector was then reconstructed by a linear combination of the orthogonal basis vectors of **U,** and resulting in uncorrelated data sets of the original data.

The same procedure can be used with a correlation matrix **R** instead of the covariance **S** matrix in the case where variances of data sets widely differ between each other.

Where,

$$\boldsymbol{R} = \boldsymbol{D}^{-1}\boldsymbol{S}\boldsymbol{D}^{-1}$$

And, D is the diagonal matrix of standard deviations:

$$\boldsymbol{D} = \begin{bmatrix} s_1 & 0 & \ldots 0 \\ 0 & s_2 & \ldots 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & \ldots s_p \end{bmatrix}$$

The procedure of finding eigenvectors and eigenvalues can be generalized by the singular value decomposition (SVD) where **X** can be written as

$$X = U^* L^{1/2} U'$$

Where **U'** are the eigenvectors of the (*n* x *p*) dispersion matrix **X'X** , **U*** are the eigenvectors of the (*p* x *n*) dispersion matrix **XX'** and the diagonal of **L**, that is a (*p* x *p*) matrix, gives the square root of the eigenvalues that are the same for **X'X** and **XX'.**

A different algorithm needs to be applied to obtain a SVD solution; it differs from the simple previous method of finding the eigenvector and eigenvalues, but is computationally suitable and efficient for larger data sets.

Form the SVD algorithm, a linear combination of the original data can be obtained resulting in *p* principal components.

$$z_1 = u_1' X$$

$$z_2 = u_2' X$$

$$\vdots$$

$$z_p = u_p' X$$

RELATED WORK

There has been a lot research related to wireless sensor networks. Algorithms for localization, routing, time synchronization and energy saving are among the most popular topics that can be found in this field. In a sensor network there is hardly any interaction with external users, so sensors need to take advantage of their capability for sharing information between each other. This exchange of data in a network with numerous sensors is the based for most of the sensor protocols that exists today.

In order to implement a localization algorithm, some schemes utilize a few anchors, which are nodes with high power transmitters and GPS that provides a known position. The idea is that by placing these locations equipped devices among an abundant sensor population, it will help find a sensor within a small area[1]. Using an area based approach by partitioning the sensors' region into triangles (where the vertices are the different combinations of anchors that can be used to form triangles) the area in which a certain sensor is located can be narrowed down. A node that wants to be found chooses three anchors from the ones that can be reached by the sensor, and then a test is performed to see if the node is inside of one of the possible combination of triangles. If it is found in one or more triangles, the best accurate triangle is chosen and a center of gravity is calculated in that triangle to estimate the position of the node.

[1] T. He, C. Huang, B. M. Blum, J. A. Stankovic and T. F. Abdelzaher, Range-Free Localization Schemes in Large Scale Sensor Networks, In *Proc. MOBICOM*, 2003.

In routing protocols, all the sensors play the same role by collaborating between each other to achieve a certain task. Due to the great amount of sensors used in any sensing task, assigning a global identifier to each node is impractical. Some schemes propose data centric routing, where a base station only communicate to specific sensors' regions. Some implementation of centric routing can be used to save energy by elimination of redundant data. Protocols that follow this idea[1] propagate all information assuming that all nodes can be used as based station for querying purposes. However, not all data is distributed uniformly since is more likely that sensors with close proximity with each other will carry the same information.

Synchronization is an important aspect of wireless senor networks that helps coordinate their operations to achieve accurate results. Some synchronization schemes estimate the offset between sensor clocks at previous time in order to predict the timestamps of two nodes at the time an event occurs[2]. Transformation of timestamps between nodes has been also suggested instead of clock adjustments. However, this scheme is limited by bounded drift assumptions. In addition, precision in synchronization have been achieved by enabling time stamping at the instant the message is actually sent at the MAC layer, eliminating uncertainties due to the sender node.

Sensor networks are constrained by their limited power. For that reason some schemes are proposed to maximize the lifetime of the network sensor. This lifetime is extended by dividing the sensor nodes into a number of sets[3], such that each set is assumed to be sensing the same

[1] F. Ye et al., A Scalable Solution to Minimum Cost Forwarding in Large Sensor Networks, Proc. 10th Int'l. Conf.Comp. Commun. and Networks, 2001, pp. 304–09.

[2] J. Elson, L. Girod, and D. Estrin. Fine-Grained Network Time Synchronization using Reference Broadcasts. In Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, USA., Dec 2002.

[3] M. Cardei et al., Energy-efficient target coverage in wireless sensor networks,in IEEE Infocom, 2005.

event. The sets are activated successively, in this way only one set is in an active state and all the other sets are in the sleep state, this will result in an overall increase of lifetime in the network.

In all these proposed methods that enhance sensor functionality, communication and sensing information in nodes are fundamental for a distributed implementation. In this thesis, mathematical tools like the Kalman filter and principal components analysis are going to take advantage of that kind of capabilities in sensors. Related work regarding these mathematical tools has also influenced this thesis and is described below.

Kalman filter is mainly use to reduce noise in real time applications, where measurements are collected recursively and the estimation is immediately generated. The down side of this is that the statistics of the noise must be estimated depending on the real world model to be used. This filter is used mainly in navigations systems to estimate different states of an object, such as position, velocity and acceleration, when the position is being measured by a noisy device. There have been many implementations and improvements in Kalman filter, for example, one of those improvements is a modified Kalman filter that performs outlier detection, without the need for manual parameter tuning by the user[1]. This robust implementation needs prior knowledge about the data that is being observed as any real time Kalman filter implementation, but it also assigns weights to each data sample probabilistically by introducing a weight w in the covariance noise R. In this thesis, a slightly different implementation of the Kalman filter is developed. The observed data samples are buffered in blocks, so there are no critical real time processing involved and the applied noise is substituted by random changes in sensor behavior. As in the robust implementation, the noise covariance R is also manipulated to minimize the distortion

[1] Jo-Anne Ting, Evangelos Theodorouand Stefan Schaal, "Learning an Outlier-Robust Kalman Filter" *Machine Learning: ECML 2007.*

produced. This kind of variability produced by uncorrelated sensors is also unpredictable, but at the same time can be better corrected by examining the change of variance produced by the corrupted sensor when the observations are buffered. Correlations between data samples of different sensors have been used to detect corrupted samples, by performing a pairwise comparison between all the sensors and applying a consensus algorithm to ignore the corrupted sensor as proposed in several papers[1]. In this thesis, the sensors are used to obtain an aggregated result, like the average, data from sensors that are not correlated with the rest of the sensors are going to caused a corrupted result in the average. Instead of comparing each sensor data between each other in order to detect if there has been an anomaly in the average, principal components analysis is performed to show the instant moment where uncorrelated sensors have caused a distortion in the average and the Kalman filter has been used to fix that distortion. Principal components analysis is mostly used to reduce dimension or correlated information by discarding the last principal components which account for lower variability. For sensor aggregation purposes, PCA is focused on the second principal, which shows peaks of disagreements between correlated sensors and uncorrelated ones.

[1] X. Xiao, W. Peng, C. Hung, and W. Lee, "Using SensorRanks for In-Network Detection of Faulty Readings in Wireless Sensor Networks."*in MobiDE*, 2007.

METHODOLOGY

The algorithm starts by continuously buffering blocks of 20 readings from each sensor. Each block is analyzed and filtered to reduce any distortion. Since the outcome of the measured data is an average of the readings, it can be shown that some sensors are affecting the aggregated result more that the others when their reading contains an offset (bias). By isolating one sensor, and calculating the mean of the rest of the working sensors, this sensor reading can then be compared to that mean. If there is a big change between those two measurements, then it is possible that the sensor that was left out is adding significant offset to the overall result. The procedure can be done iteratively by leaving one sensor out and comparing that single sensor reading with the mean of the rest of the sensors. Later, every sensor that was left out is assigned a weight depending on its closeness to the mean of the rest.

This equation shows how sensor readings are weighted:

$$y^*_i = \left| \frac{1}{\mu_{(-y_i)}} \left(y_i - \mu_{(y_i)} \right) \right| \qquad i = 1,2,\ldots p$$

$$y_i = (1 - y^*_i) y_i$$

Where y^*_i is the weighted reading that is the normalized difference between the current reading y_i , p is the number of sensors and the mean $\mu_{(-y_i)}$ is the mean calculated without y_i. This means that the sensor values that agree with the average of the other sensors will have a weight close to one.

Kalman filter uses these values to make the aggregated result resilient to a sensor failure. This filter estimates measurements based on the following predictor equations.

(1)$K_n = A\bar{P}_n H^T (H\bar{P}_n H^T + R)^{-1}$

(2)$\hat{x}_n = A\bar{x}_n + K_n(y_n - H\bar{x}_n)$

(3)$P_n = A\bar{P}_n A^T - A\bar{P}_n H^T R^{-1} H\bar{P}_n A^T$

By examining the state estimate equation (2), it can be seen that the estimate x is projected ahead by using past estimations, the measured output y and the Kalman gain K which adjust the estimate due to the actual measurement. By looking at equation (1) shows that K depends on the estimation error covariance P (3) and the measurement covariance noise R. For this purpose, R is going to be the distortion caused by a corrupted sensor. R will not have a covariance fixed value, but it will use the values of the squared variance of the sensor readings in each time n. If the distortion value is large then K will be small and the measurement y is not going to be trusted. If the variance of the measurements is close to zero, meaning that all the sensors all correlated, the distortion is small and the actual measurement y is going to be trusted. In order to use the

Kalman filter to minimize distortion the process must be modeled by linear system. Although, it would seem that a constant or a zero order Kalman filter should suffice this criteria, simulations showed that a first order polynomial gives better estimations. The following equation describes the linear system used:

$$x_{n+1} = Ax_n$$

$$y_n = x_n + z$$

Where the vector *x* is the state of the system and A is the transition matrix $\begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}$ where T is the sampling period of the readings. The vector estimate *x* cannot be directly measured, so the actual measurement *y* that is a function of *x* corrupted by noise *z* is used to assist in the estimate calculation. The system model sometimes requires a parameter called process noise which is used as a tuning factor when uncertainties in modeling the real world are present. Since the sensor model is relatively simple, this parameter is not taken into account.

At the same time, network reliability can be measured by detecting the distortion in the average readings. To achieve this, principal component analysis of the 20 readings block is performed. Principal components of that data set can be computed by a procedure of finding eigenvectors and eigenvalues that can be generalized by singular value decomposition (SVD) where X can be written as

$$X = U^{*} L^{1/2} U'$$

From the SVD algorithm, a linear combination of the original data can be obtained resulting in p principal components.

$$z_1 = u_1' X$$

$$z_2 = u_2' X$$

$$\vdots$$

$$z_p = u_p' X$$

These equations show that the original vector was then reconstructed by a linear combination of the orthogonal basis vectors of U, and resulting in uncorrelated data sets of the original sensor readings. The first principal component represents the combination of the data sets or their average. The second principal component represents most of the change in the difference between data sets, showing less variation than the first principal component. This is the principal component that determines if the sensors reading start to be uncorrelated. The principal components that follow capture the rest of the difference with even less variability.

The following block diagram shows the steps taken to compensate and detect a distortion in the average.

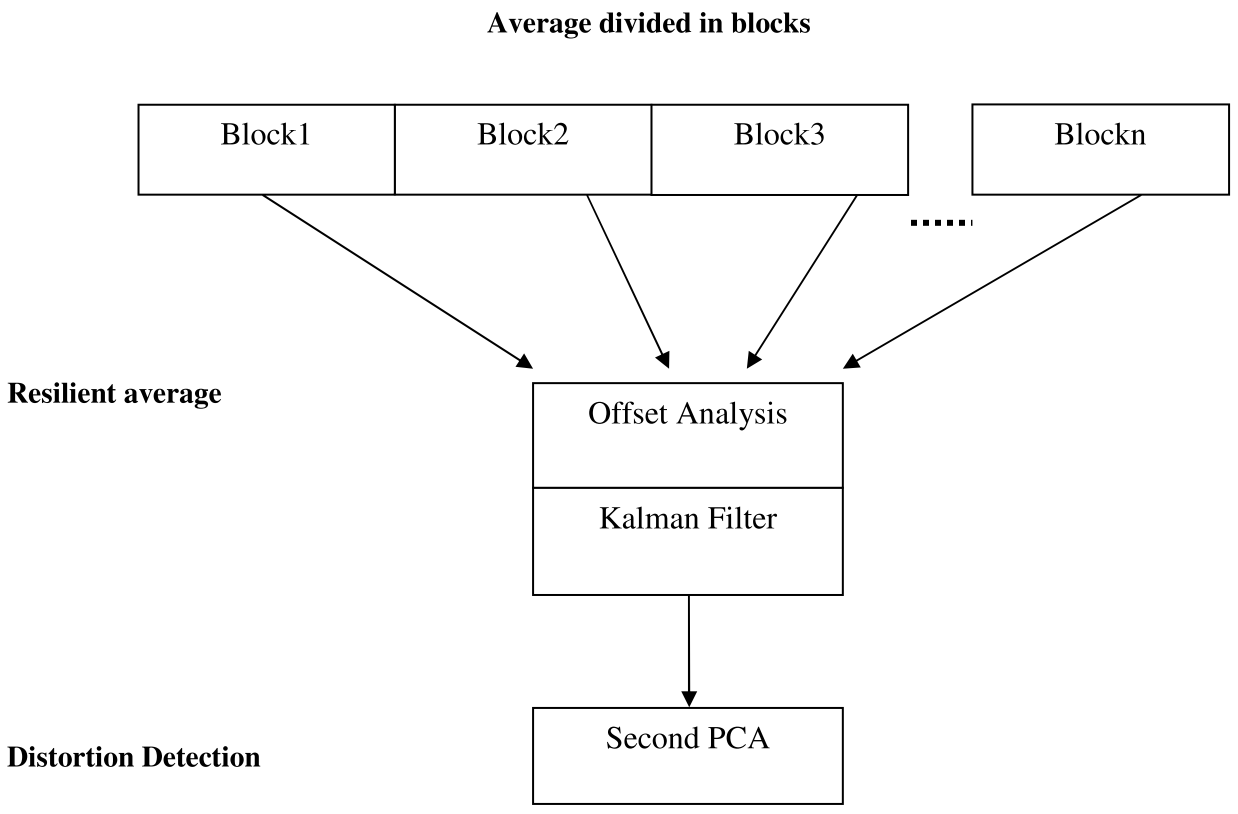
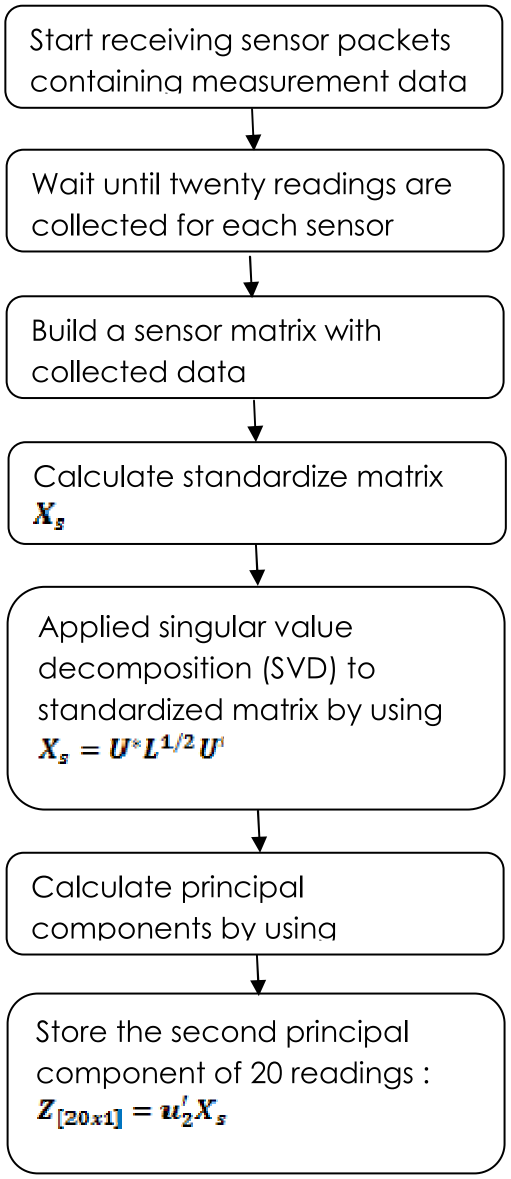
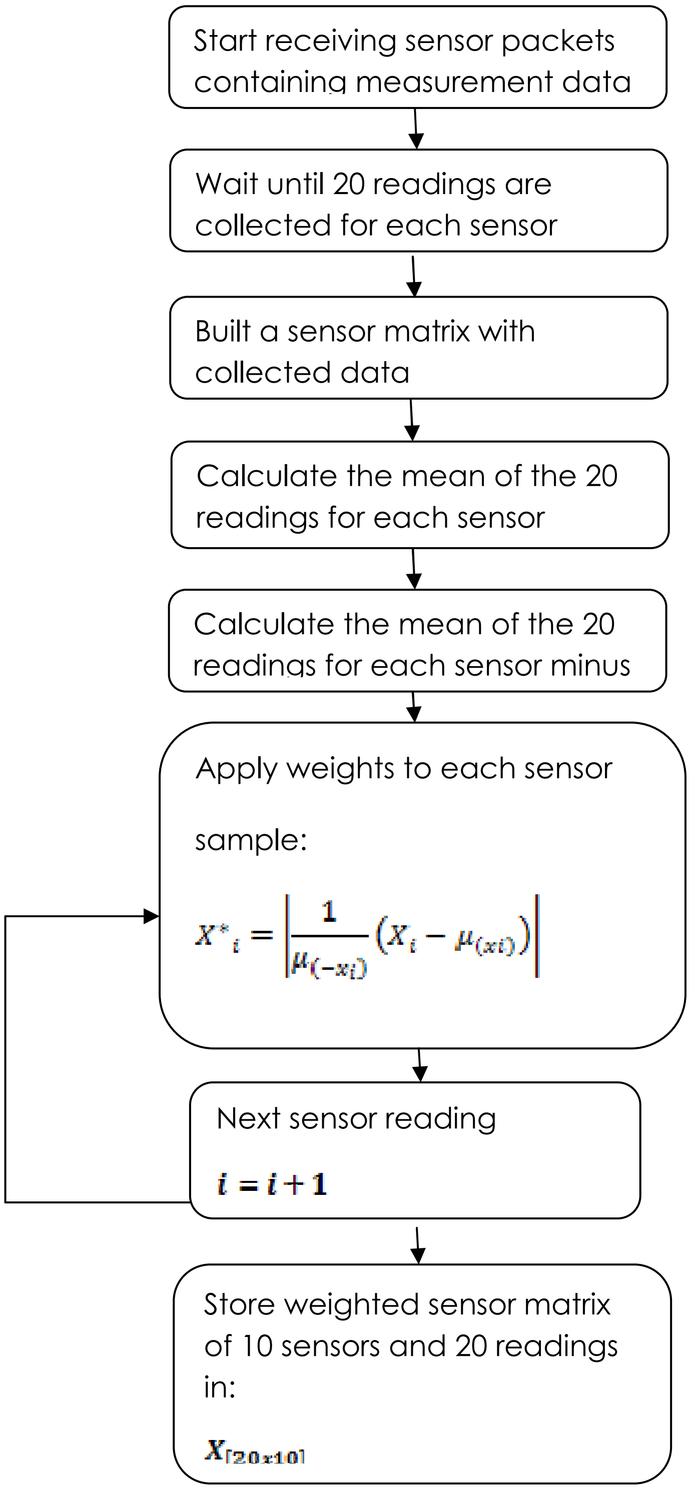


Figure 1. Resilient average and distortion detection block diagram

PCA algorithm implemented in base station when 20 readings sensed by 10 sensors:

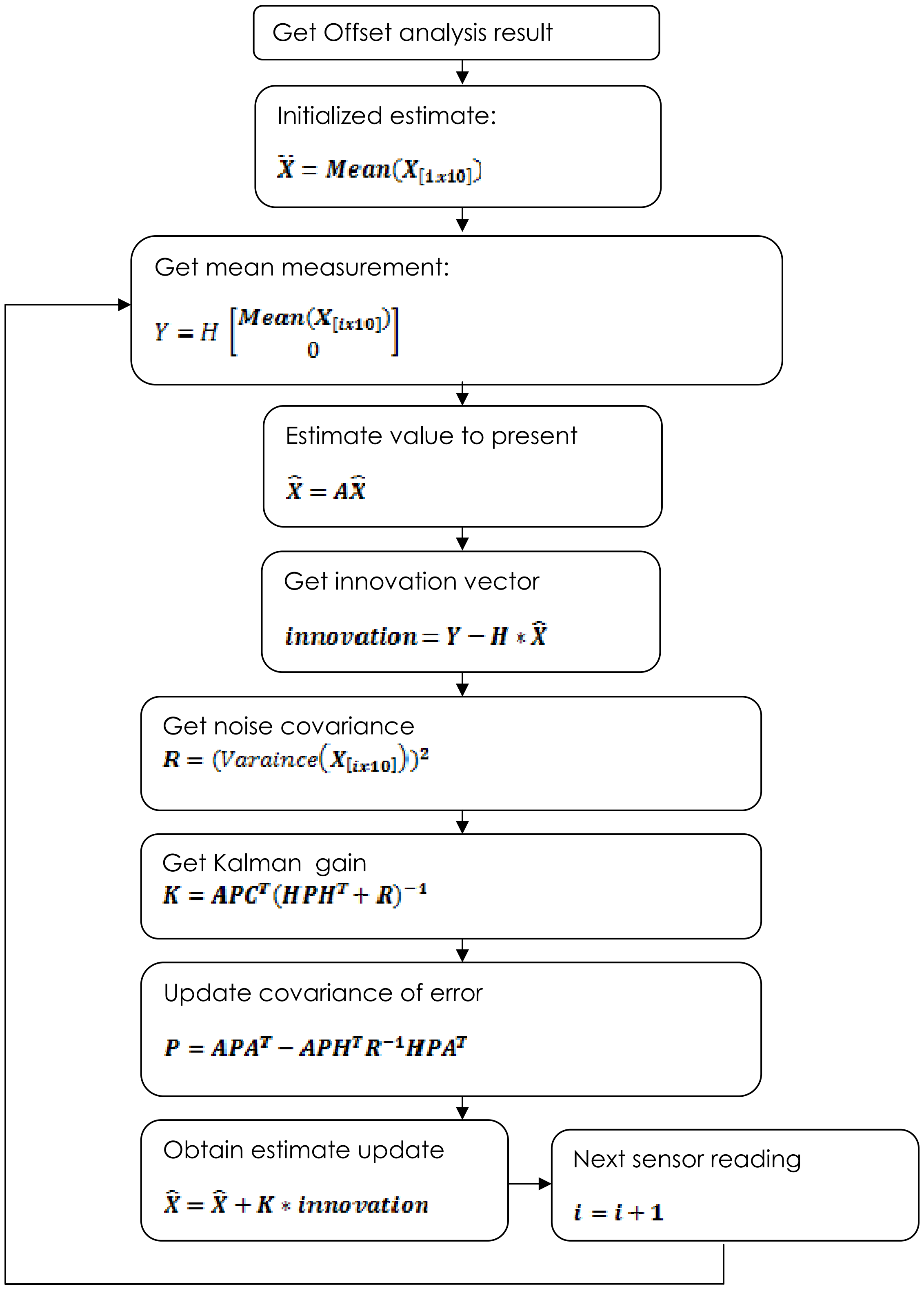

Start receiving sensor packets containing measurement data
Wait until twenty readings are collected for each sensor
Build a sensor matrix with collected data
Calculate standardize matrix X_s
Applied singular value decomposition (SVD) to standardized matrix by using $X_s = U^* L^{1/2} U'$
Calculate principal components by using
Store the second principal component of 20 readings : $Z_{[20x1]} = u_2' X_s$

Offset analysis algorithm implemented in base station:



Start receiving sensor packets containing measurement data
Wait until 20 readings are collected for each sensor
Built a sensor matrix with collected data
Calculate the mean of the 20 readings for each sensor
Calculate the mean of the 20 readings for each sensor minus
Apply weights to each sensor sample:
$X^*_i = \left| \frac{1}{\mu_{(-x_i)}} (X_i - \mu_{(xi)}) \right|$
Next sensor reading
$i = i + 1$
Store weighted sensor matrix of 10 sensors and 20 readings in:
$X_{[20x10]}$

Kalman filter algorithm implemented in base station for twenty data samples and ten sensors:


Get Offset analysis result
Initialized estimate:
$\hat{X} = Mean(X_{[1x10]})$
Get mean measurement:
$Y = H \begin{bmatrix} Mean(X_{[ix10]}) \\ 0 \end{bmatrix}$
Estimate value to present
$\hat{X} = A\hat{X}$
Get innovation vector
$innovation = Y - H * \hat{X}$
Get noise covariance
$R = (Varaince(X_{[ix10]}))^2$
Get Kalman gain
$K = APC^T(HPH^T + R)^{-1}$
Update covariance of error
$P = APA^T - APH^T R^{-1} HPA^T$
Obtain estimate update
$\hat{X} = \hat{X} + K * innovation$
Next sensor reading
$i = i + 1$

EXPERIMENTS

A wireless sensor network is used to gather and transfer information to a computer. The data collected is ambient light intensity readings produced in a room which are captured by the sensors and sent in blocks to a computer for later processing using a Java program. Since sensors are measuring light, distortion is easy to simulate just by blocking light in the sensor's photo resistors. The nodes used in the network are MICAz, each device is a mote module used for enabling low-power wireless sensor networks. Its radio communication capabilities make it suitable for wireless sensor networks, since data information can be exchanged between motes. In addition, a sensor board (mts310) is integrated to the MICAz to retrieve light intensity data. The MICAz motes runs the TinyOS operating system which handles RF communication, power management, I/O expansion, and secondary storage. Applications for this operating system are written in NesC programming language which are built of software components connected to each other using interfaces. Some interfaces and components are provided by TinyOS for common abstractions such as packet communication, routing, sensing and storage.

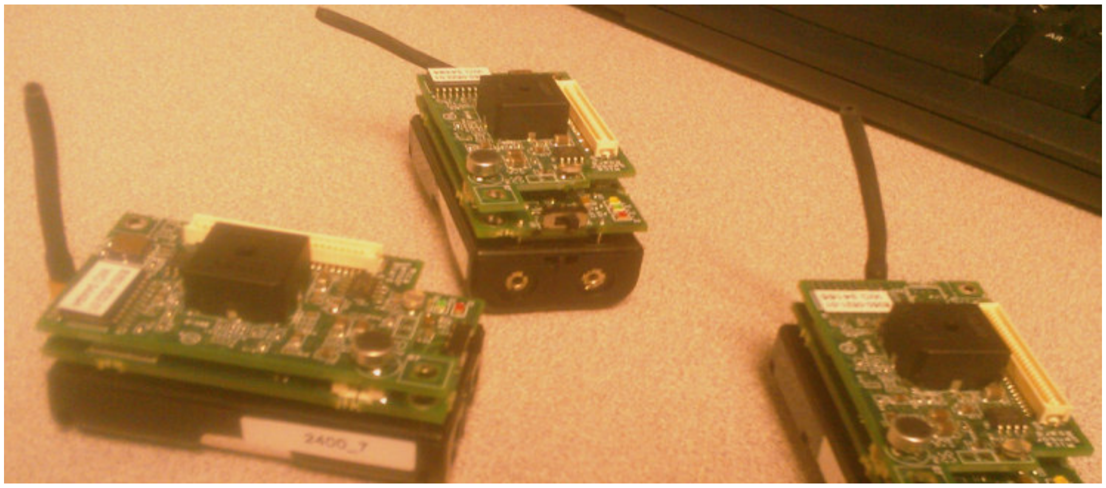


Figure 2. MICAz motes

A brief description of those hardware components is explained as follows:

The MICAz module has been designed for embedded sensor networks. In addition, a sensor board can be connected to the MICAz in order to be use as a measurement system. The sensor board is a collection of different sensing modalities. These include a dual-axis accelerometer, light, temperature, speaker and microphone. A base station is used as a programming board to install nesC programs in the MICAz and allows aggregation of sensor network data onto a PC or other computer platform. The performance of the MICAz is limited. It contains a 128K bytes of flash memory and 4K bytes of configuration EEPROM. A 10 bit analog to digital converter and UART for Serial communication is provided. One of the most important features on the MICAz is the radio. It has an outdoor range of 75 meters to 100 meters with a data rate of 250 kbps.

The following picture shows the hardware used for the experiment:

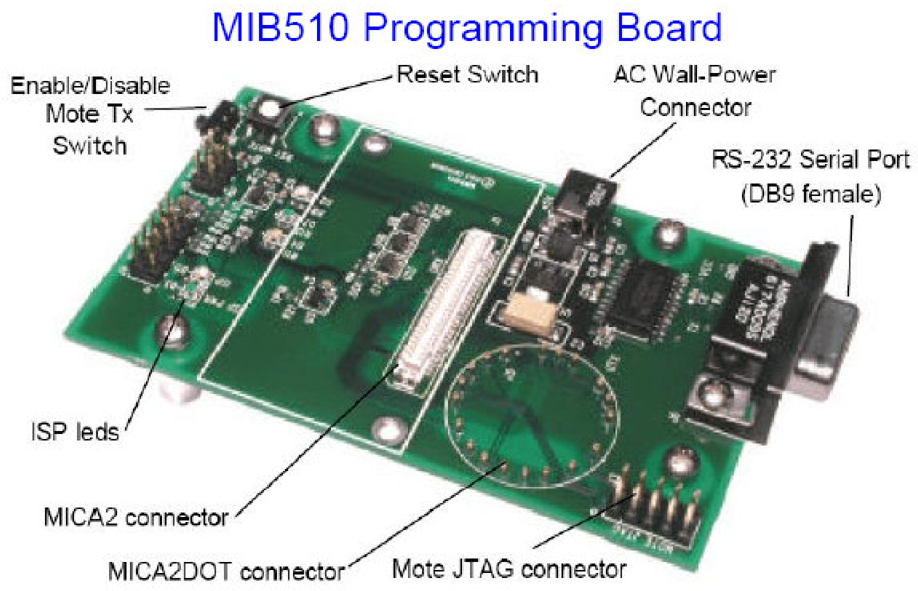


Figure 3. Programming Board

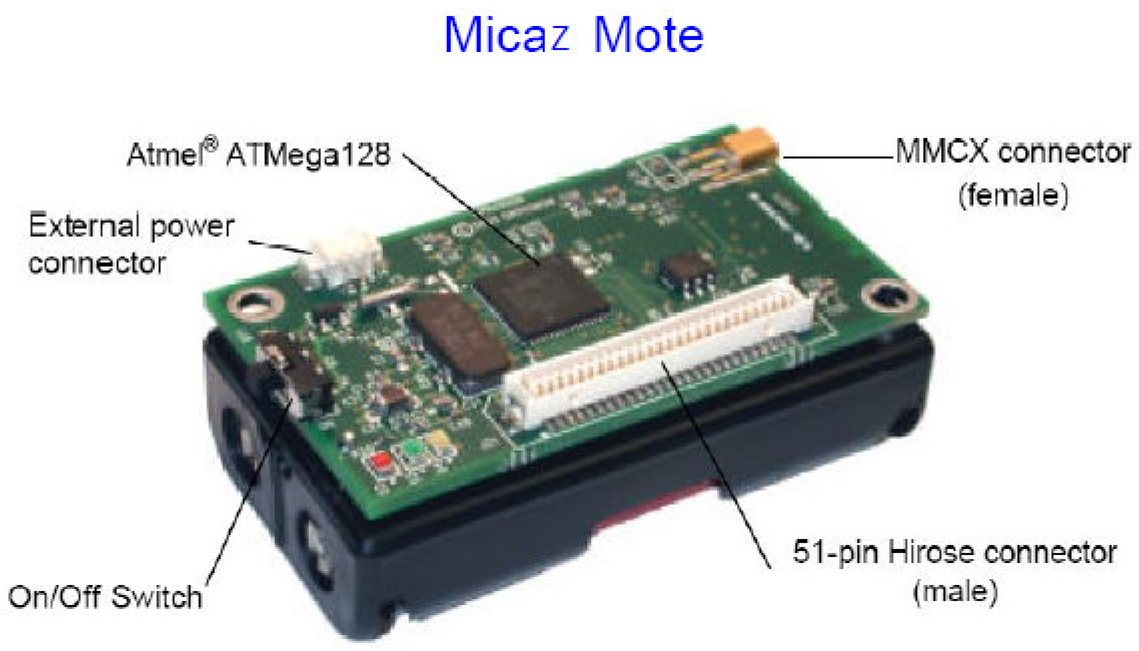


Figure 4. MICAz Mote

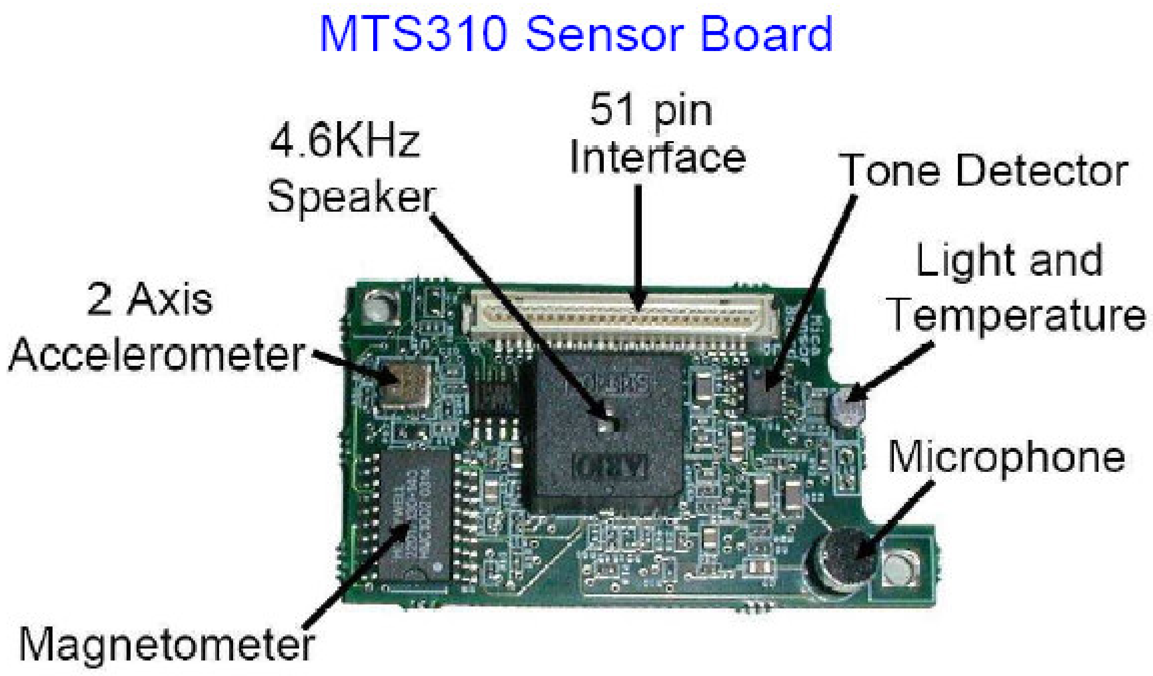


Figure 5. Sensor Board

The nesC program is used to enable radio communication, serial communication and light sensors of the MICAz. Since NesC is an event driven programming language, a timer is used to trigger sensing the data and transmitting the data. Every second, the data sample is gathered from the light sensor A/D converter. This information is appended as payload in a packet and broadcasted through the network. All the nodes receive the packet but only one node handles this packet and sends it throughout the serial port. The packet that is being exchanged contains two important fields, sensor number and payload, needed to identify each node with their respective data.

In Nesc programming, components can be compared to Java objects since they encapsulate functions and variables in its implementation. Components have a local namespace, so different components can call the same named function, but that function may have a different implementation for each component which most of the time are implemented via interfaces. As a result, a component that represents an abstraction of service provides a certain interface, which can be used by other components. A nesC program can have a collection of components that wire between each other. This type of communication is called wiring. This connection is done at compile time, so runtime allocation is not required. This static approach is offered because of the unattended behavior of embedded system were user has no interaction with the device. Two kinds of components are used by a nesC program: configurations and modules. Configurations connect declarations of different components and enable components to call each other. Components wiring are the implementation of a configuration producing an abstraction of composed modules. The advantage of encapsulating modules as an abstraction is to later be able to wire a single module to all the resources contained in the abstraction. Modules define functions and allocate state. Those modules must implement all the commands of the interfaces it provides and all the events used by the interface. Interfaces are bidirectional; they specify a set of functions to be implemented by the interface's provider (commands) and a set to be implemented by the interface's user (events). Commands typically call downwards (from application components to components closer to hardware) while events call upwards. This is achieved by a split phase operation, where a completion of a request is a call back. For example in order to send a message the split phase operation is divided in two phases, the first phase, declares a buffer storage in a frame, name the handler and request transmission. In the second phase a done completion signal is performed.

These programming concepts were applied to acquire and transfer data from the MICAz sensors. The components used are SenseC and SenseAppC. The first one is a module, in its signature, it is specified that this module needs to use several interfaces. Since the module is only using and not providing interfaces, then only events from the interfaces needs to be implement. The events generate callback signal or interrupts when a hardware procedure has been completed. In this module several signal from events are needed to capture and transfer data: a signal that the MICAz has finishing booting, a signal that indicates that the radio has been successfully started and stopped, a signal that fired a timer interrupted for data sampling purposes, a signal that indicates that a data has been captured and read, and signal that shows that a message has been sent or received by other MICAz sensor. The booting signal is going to implement a call to start the radio. In the same way, after the radio has finishing starting up, a signal is issued. This event implements the starting of a timer for data sampling. While data is being captured by the photoresistor, a signal for each sample is used to trigger a packet assembly and transfer procedure. Each data sample is put into a packet with its node identification number and broadcasted to the other MICAz. When the packet is received by the other sensor a signal of that event is produced. The implementation for the “packet received” event includes transferring data through the serial port. Obviously, only the sensor that is connected to the base station will be able to transfer data.

The second component is the configuration component. This component wires the resources used by the SenseC module. The components are first listed in the specification section of the configuration and then wired in the implementation section. The components and its services are as follows: The MainC component is used to provide booting services, TimerMilliC component provide the timer for sampling,i PhotoC component provides the callbacks when the light sensor has read a value, and SerialActiveMessageC and ActiveMessageC control serial port and radio communication respectively.

The nesC code to achieve sensor communication and transfer of data is shown in appendix A:

A Java program is used to implement a GUI for a graphical representation of the node’s position and light intensity, also it displays a chart of the reading being captured, a chart representing the resilient average produced by the Kalman filter and a chart showing the second principal component. The Java program gets the packet information from the serial port, strips the needed fields (sensor number and payload) and start the algorithm after buffering 20 readings from the sensors. After the data has been filtered and analyzed, the results are displayed in charts every time 20 data samples from each sensor are collected and sent to the serial port.

The Java code for obtaining principal components is shown in appendix B:

The following screenshot shows the GUI that was developed. The white squares in the grid represent working sensors, which change to a darker color when the light intensity is lowered. The next three charts shows the light readings being measured, the resilient average and the second principal component

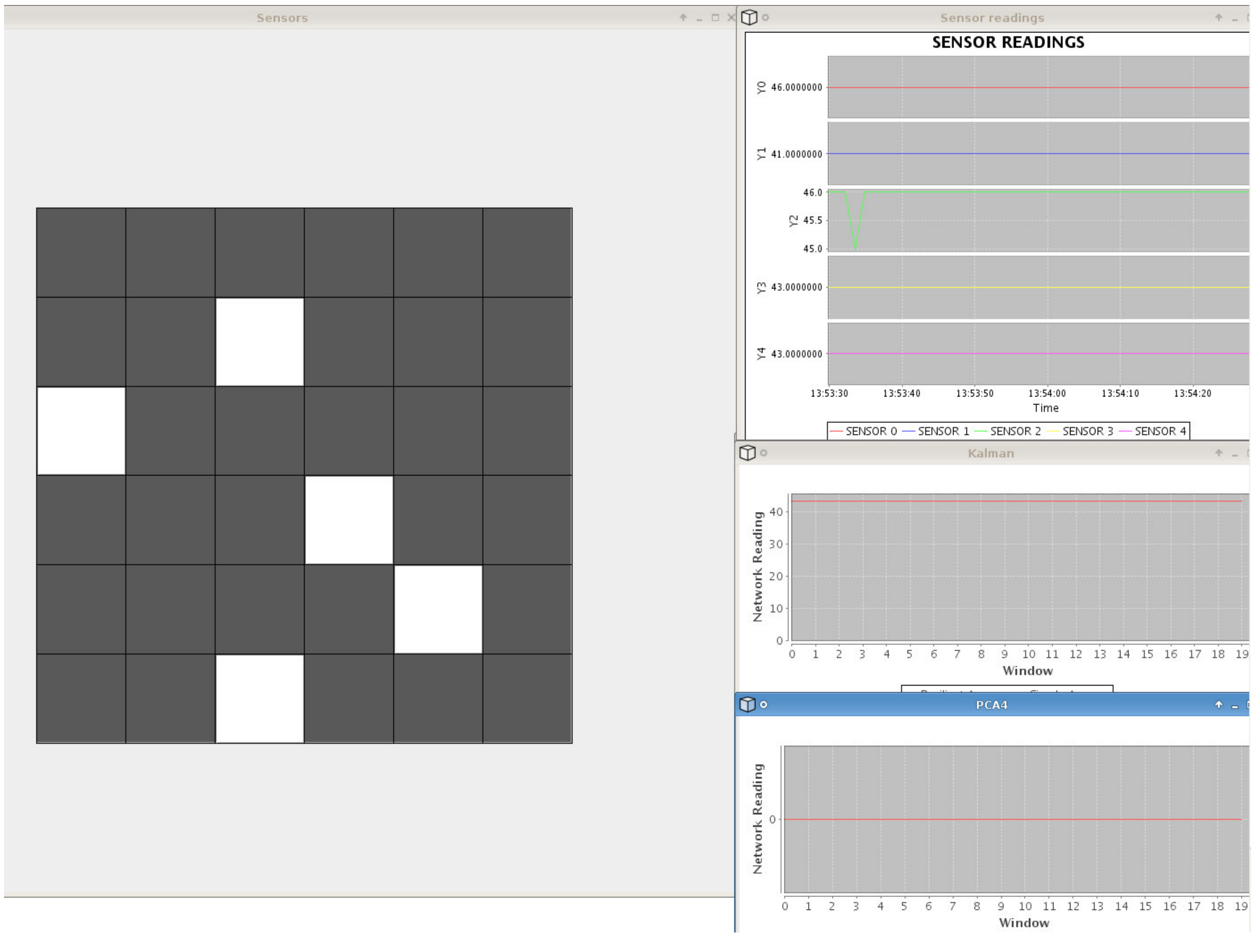


Figure 6. Java GUI showing working sensors.

RESULTS

By gathering a block of twenty readings and creating a distortion in one of the sensors while data is being captured, the following result is obtained:

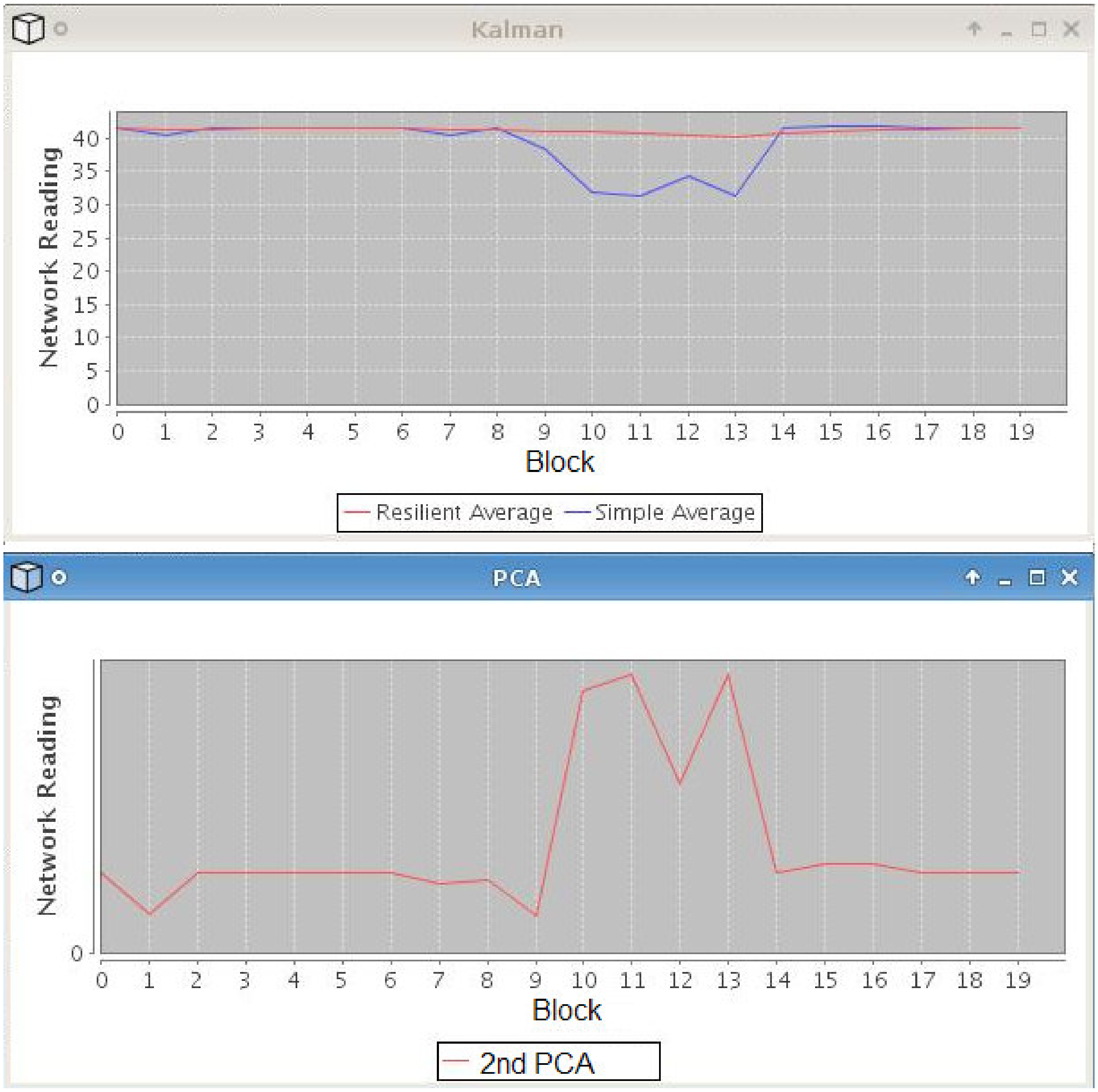


Figure 7. Resilient average using Kalman filter followed by second PCA

Figure 4 shows how the Kalman Filter produces a resilient average. The distortion generated by one sensor is hardly noticeable when the Kalman filter is applied. In the next graph the second principal component shows evident peaks where the distortion has occurred. Although, 5 principal components can be obtained, only the second one is used, since it contains the information that represents when the biggest distortion has occurred. The following graphs show the five principal components obtained when there is a change in the sensor correlation readings. The second principal component is marked with a dashed circle. It can be seen that the information contained in that component give the best indication that there is a distortion present in the sensors.

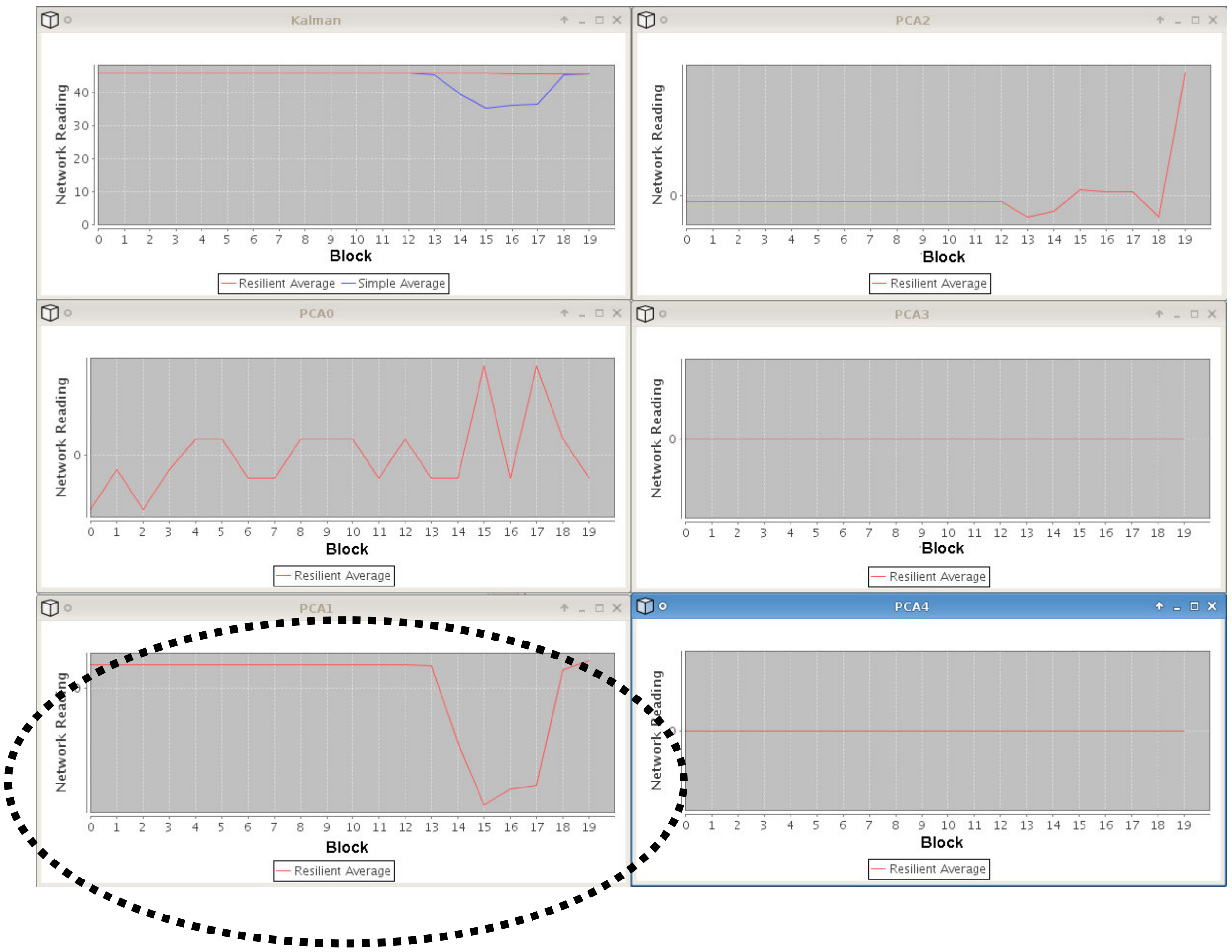


Figure 8. Principal components of the sensor correlation matrix.

In the case where more than one sensor is adding corrupted values to the average the Kalman filter starts to degrade .The following charts shows the cases for 2, 4 and 5 are distorted respectively from a population of 5 sensors.

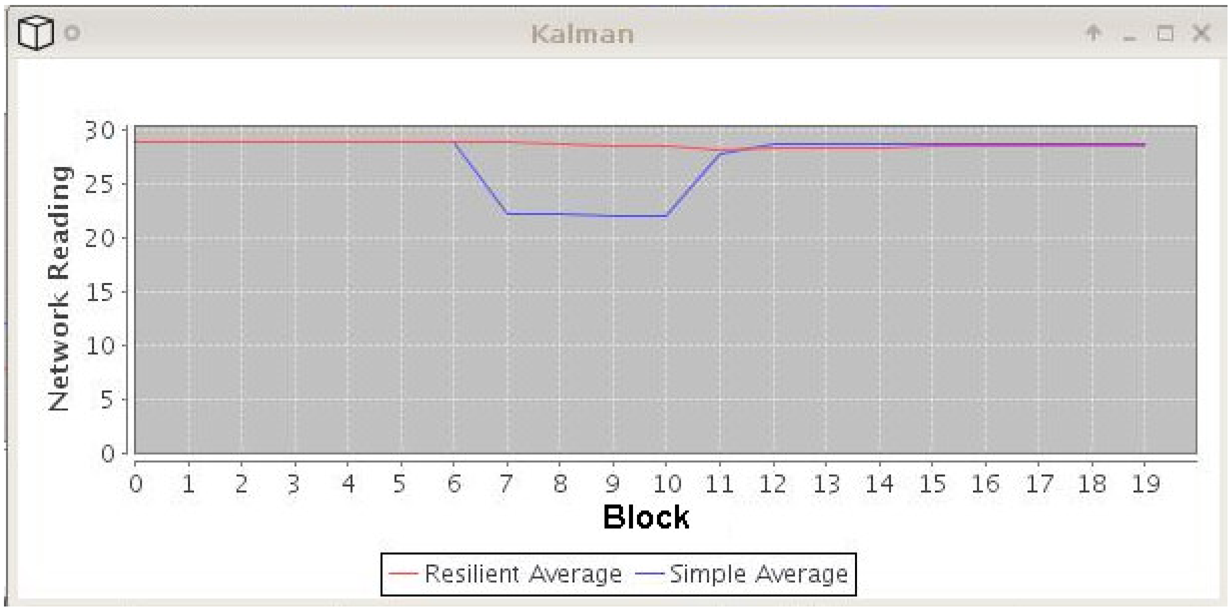


Figure 9. Resilient average when two sensors are corrupted.

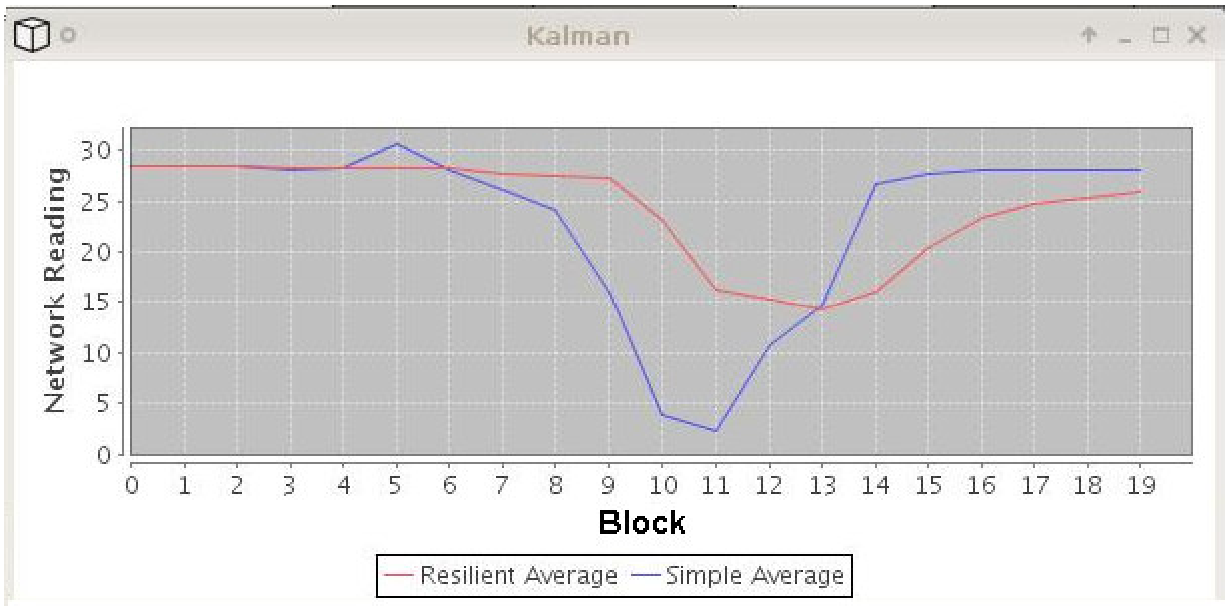


Figure 10. Resilient average when four sensors are corrupted.

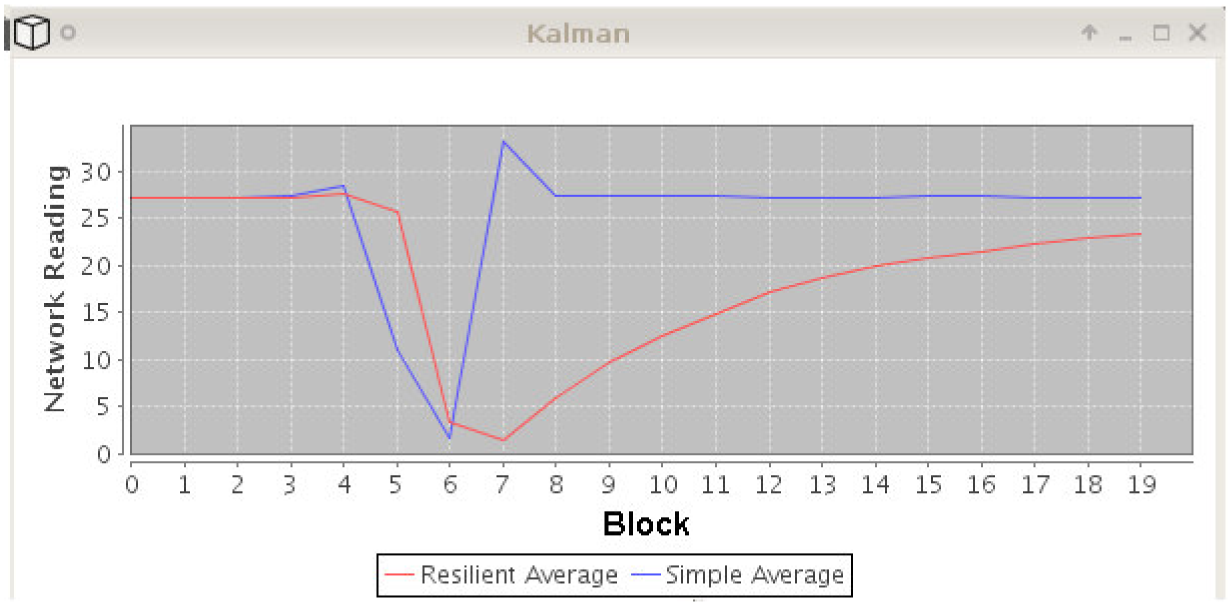


Figure 11. Resilient average when all the sensors (five) are corrupted.

CONCLUSION

Distribution and redundancy of information are common characteristics that are found in sensors networks and can be used to increase efficiency and performance of data acquisition. This raw data obtained by sensors can be manipulated by using mathematical tools in order to verified or validate the information. In this thesis, advantage of the redundancy feature has been taken in order to analyze and change the received data. Kalman filter and principal components analysis implementation were used to compensate and detect average distortion due to corrupted sensors. In order to prove this proposed scheme, MICAz sensors were used in the experiments, since their hardware and software functionality allowed real time communication and acquisition of data for later processing. A NesC program was developed in the sensors to exchange light intensity readings to the base station. Likewise, a Java program was developed to validate and display the data received in the base station. Evidently, as more sensors started sending corrupted data, the resilient average started to resemble a simple average. Although, the Kalman filter worked very well when few sensors were corrupted, as more sensor started to be uncorrelated with the rest, the aggregated result could not be improved. The second principal component was used to determine if corrupt measurements have produced uncorrelated data, which means that the aggregated result may not be one hundred percent accurate. For this reason, it can be assumed that the second principal component capture the strongest correlation among sensor discrepancies. Although most of this variability is represented by the second principal component, still there is some small information related to data disagreements captured by the subsequent principal components.

REFERENCES:

- J. E. Jackson. A User's Guide To Principal Components. John Wiley & Sons, 1991.

- L. Buttyan, P. Schaffer, and I. Vajda. Resilient aggregation with attack detection in sensor networks. In Proc. of 2nd IEEE Workshop on Sensor Networks and Systems for Pervasive Computing, 2006.

- P. Zarchan and H. Musoff. Fundamentals of Kalman filtering: a practical approach. Progress in astronautics and aeronautics. American Institute of Aeronautics and Astronautics, Inc., 2000.

- Jo-Anne Ting, Evangelos Theodorouand Stefan Schaal, Learning an Outlier-Robust Kalman Filter Machine Learning: ECML 2007.

- Wagner, D. 2004. Resilient aggregation in sensor networks. In Proceedings of the ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN).

- X. Xiao, W. Peng, C. Hung, and W. Lee, Using SensorRanks for In-Network Detection of Faulty Readings in Wireless Sensor Networks. MobiDE, 2007.

- Zhuang, Y., Chen, L., Wang, X. S. and Lian, J, A weighted moving average-based approach for cleaning sensor data, Proceedings of IEEE International Conference on Distributed Computing System, 2007

- Sheng, B., Li, Q., Mao, W. and Jin, W. (2007) Outlier detection in sensor networks, Proceedings of ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp.219-228.

- PRZYDATEK, B., SONG, D., AND PERRIG, A. 2003. SIA: Secure information aggregation in sensor networks. In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems

- Kerschen, G, De Boe, P, Golinval, J, and Worden, K, 2005. Sensor Validation using Principal Component Analysis, Smart Materials and Structures, 14, 36-42.

APPENDIX A

SenseC.nc

```
#include <Timer.h>
#include "SenseTest.h"

module SenseC
{
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Packet;
    interface Packet as RadioPacket;
    interface AMSend as RadioAMSend;
    interface AMPacket;
    interface AMPacket as RadioAMPacket;
    interface AMSend;
    interface Receive;
    interface SplitControl as AMControl;

  }
}
implementation
{


  message_t pkt;
  uint16_t counter;

  event void Boot.booted() {
    call AMControl.start();// initialize radio


  }

  event void AMControl.startDone(error_t err){
    if (err == SUCCESS) {
      call Timer.startPeriodic(SAMPLING_FREQUENCY);
    }
    else {
      call AMControl.start();
    }
  }
  event void AMControl.stopDone(error_t err){}
  event void Timer.fired()
  {
    counter++;

  call Read.read(); //get light intensity reading
  }

  event void Read.readDone(error_t result, uint16_t data)
  {
//append sensor number and data to packet

    if (result == SUCCESS){
    Msg* sensepkt= (Msg*)(call Packet.getPayload(&pkt, NULL));
   sensepkt->data=data;
   sensepkt->nodeid = TOS_NODE_ID;
   if (call RadioAMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(Msg)) ==
SUCCESS){

   }
 // sensor 1 sends data to serial port
   if (TOS_NODE_ID == 1){
     call AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(Msg));}

    }
  }

  event void AMSend.sendDone(message_t* msg,error_t err) {
}
  event void RadioAMSend.sendDone(message_t* msg,error_t err) {


}

  event message_t* Receive.receive(message_t* msg, void* payload, uint8_t
len){
    Msg* sensepkt = (Msg*)payload;

    // broadcast data to all sensors
        call AMSend.send(AM_BROADCAST_ADDR, msg, sizeof(Msg));

      return msg;
  }

}
```

SenseTest.h

```
#ifndef SENSETEST_H
#define SENSETEST_H

enum {
  AM_SENSEMSG = 6,
  SAMPLING_FREQUENCY = 1000
};

typedef nx_struct Msg {
 nx_uint16_t nodeid;
  nx_uint16_t data;

} Msg;

#endif
```

SenseAppC.nc

```
#include "SenseTest.h"
configuration SenseAppC
{
}
implementation {

  components SenseC, MainC, LedsC, new TimerMilliC(), new PhotoC() as
Sensor;
  components SerialActiveMessageC;
  components ActiveMessageC;
  components new SerialAMSenderC(7);
components new AMSenderC(AM_SENSEMSG);


 components new AMReceiverC(AM_SENSEMSG);

  SenseC.Boot -> MainC;
  SenseC.Leds -> LedsC;
  SenseC.Timer -> TimerMilliC;
  SenseC.Read -> Sensor;
  SenseC.Packet -> SerialAMSenderC;
  SenseC.AMPacket ->SerialAMSenderC;
  SenseC.AMControl->SerialActiveMessageC;
  SenseC.AMControl->ActiveMessageC;
  SenseC.AMSend->SerialAMSenderC;
  SenseC.RadioAMSend->AMSenderC;
  SenseC.RadioPacket ->AMSenderC;
  SenseC.RadioAMPacket -> AMSenderC;
  SenseC.Receive -> AMReceiverC;

}
```

APPENDIX B

```
package lightsensor;

import java.util.*;



// This class converts the raw data into integers and
//get the reading data from the sensor packet
public class BufferSensor {


//private static byte[] packet=new byte[2];
private  byte number;
private  byte data;
private  byte data2;


public  int getSensor_number(){
    return (int)(number) & 0xFF;

}



public int getSensor_data(){
    if(((int)(data2) & 0xFF)>0){
        return ((int)(data) & 0xFF)+256;
    }
    return (int)(data) & 0xFF;

}

    public  void setSensor(byte[] pkt){
    data=pkt[11];
    data2=pkt[10];
    number=pkt[9];
}

}
```

```
package lightsensor;

import java.io.*;
import java.util.*;
import java.text.*;
import Jama.*;

//This class returns the principal components of a given correlation matrix
public class PCA{

    public static double[][] principalComponents(double[][] X,int column){
        Matrix x = new Matrix(X);
     SingularValueDecomposition svd=new  SingularValueDecomposition(x);

     Matrix V = svd.getV();
       Matrix rowproj = x.times(V);
            double[][] Z=rowproj.getArray();
        double[][] Zabs =new double[20][5] ;

      for (int j=0; j< 20; j++){

          Zabs[j][1]=(Z[j][column]);


       }

       return Zabs; //return the second principal component



    }

  //Method to standardize and center the mean
    public static double[][] Standardize(int nrow, int ncol, double[][] A)
    {
    double[] colmeans = new double[ncol];
    double[] colstdevs = new double[ncol];
    double[][] Adat = new double[nrow][ncol];
    double[] tempcol = new double[nrow];
    double tot;

    // Get means and standard deviations
    for (int j=0; j<ncol; j++)
        {
         tot = 0.0;
         for (int i=0; i<nrow; i++)
             {
                tempcol[i] = A[i][j];
                tot += tempcol[i];
             }
         colmeans[j] = tot/(double)nrow;
         for (int i=0; i<nrow; i++) {
                colstdevs[j] += Math.pow(tempcol[i]-colmeans[j], 2.0);
             }
             colstdevs[j] = Math.sqrt(colstdevs[j]/((double)nrow));
             if (colstdevs[j] < 0.0001) { colstdevs[j] = 1.0; }
      }


    for (int j=0; j<ncol; j++)
        {
         for (int i=0; i<nrow; i++)
             {
                  Adat[i][j] = (A[i][j] - colmeans[j])/
                 (Math.sqrt((double)nrow)*colstdevs[j]);
             }
        }
    return Adat;
    }
}
```

```
package lightsensor;


//This class performs an offset analysis between sensors.

public class OffsetAnalysis {

    public static double[][] Offset(int nrow, int ncol, double[][] A)
    {
    double[] rowmeans = new double[nrow];
    double[] rowmeansminus = new double[nrow];

    double[][] weight = new double[nrow][ncol];
    double[] temprow = new double[ncol];
    double[] temprowminus = new double[ncol-1];
    double tot;
      double totminus;
        int t=0;


    for (int i=0; i<nrow; i++)
        {
         tot = 0.0;

         for (int j=0; j<ncol; j++)
             {
                temprow[j] = A[i][j];
                tot += temprow[j];
             }


         rowmeans[i] = tot/(double)ncol;
        for (int k=0; k<ncol; k++){
             totminus = 0.0;
         for (int j=0; j<ncol; j++)
             {
             if (j!=k){
                temprowminus[t] = A[i][j];
                totminus += temprowminus[t];
                t++;
             }
             }
         rowmeansminus[i] = totminus/(double)(ncol-1);
         weight[i][k]=1-Math.abs(rowmeans[i]-rowmeansminus[i])/rowmeans[i];
         t=0;
        }

      }
    return weight; //return weight that is going to be used as a confident
value
    }
}



package lightsensor;

import Jama.Matrix;
import java.io.*;
import net.tinyos.packet.*;
import net.tinyos.util.*;
import net.tinyos.message.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.*;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.*;
import org.jfree.ui.RefineryUtilities;

//this class displays the sensor GUI and implements the kalman filter

public class ShapesDemo2D {

    private static int end = 0;

    public static void main(String[] args) throws IOException {
        int i = 0;
        JFrame jf = new JFrame("Sensors");
        Container cp = jf.getContentPane();
        BufferSensor[] buffersensor = new BufferSensor[800];
        for (int j = 0; j < 800; j++) {
            buffersensor[j] = new BufferSensor();
        }
        MyCanvas tl = new MyCanvas(buffersensor);
        cp.add(tl);
        jf.setSize(1000, 1000);
        jf.setVisible(true);


        String source = null;
        PacketSource reader;
        if (args.length == 2 && args[0].equals("-comm")) {
            source = args[1];
        } else if (args.length > 0) {
            System.err.println("usage: java net.tinyos.tools.Listen [-comm
PACKETSOURCE]");
            System.err.println("       (default packet source from MOTECOM
environment variable)");
            System.exit(2);
        }
        if (source == null) {
            reader = BuildSource.makePacketSource();
        } else {
            reader = BuildSource.makePacketSource(source);
        }
        if (reader == null) {
            System.err.println("Invalid packet source (check your MOTECOM
environment variable)");
            System.exit(2);
        }

        try {
            reader.open(PrintStreamMessenger.err);
            for (;;) {
                byte[] packet = reader.readPacket();
                buffersensor[i].setSensor(packet);
                i++;
                if (i == 10) {
                    tl.getData();
                    i = 0;
                }


                System.out.flush();

            }
        } catch (IOException e) {
            System.err.println("Error on " + reader.getName() + ": " + e);
        }



    }
}

class MyCanvas extends JPanel {

    private Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    private int sensor_radius1[][] = new int[6][6];
    private GridElement[][] element = new GridElement[7][7];
    private ArrayList occupied_elements = new ArrayList();
    private int data;
    private int number;
    private String dat;
    private final int rows=20;
    private Timer animationTimer;
    private boolean received1 = false;
    private boolean received2 = false;
    private boolean received3 = false;
    private boolean received4 = false;
    private boolean receivedall = false;
    private BufferedReader input;
    private BufferSensor[] buffer;
    private DynamicDataDemo3 demo;
    private DynamicDataDemo3 demo2;
    private LineChartDemo6 demo9;
    private PCA pca;
    private double[][] X = new double[rows][5];
    private int a,b,c,d,e = 0;
    private boolean received5=false;
    private Matrix Xw=new Matrix(X);
    private double[][] xi = new double[2][1];
    private double[][] xh = new double[2][1];
    private double[][] ci = new double[1][2];
    private double[][] yi1 = new double[1][1];
    private double[][] yi2 = new double[1][1];
    private double[][] yi3 = new double[1][1];
    private double[][] s = new double[1][1];
    private double[][] s1 = new double[1][1];
    private double[][] ai = new double[2][2];
    private double[][] innovation = new double[1][1];
    private double[][] p = new double[2][2];
    private double[][] wi= new double[1][1];
    private double[][] k = new double[2][1];
    private double[][] FilterX = new double[2][1];
    private double[][] FilterX2 = new double[rows][2];
    private double[][] AverageX= new double[rows][2];
    private double[] meanz = new double[5];
    private boolean initialize = false;


    public MyCanvas(BufferSensor[] buf) {

        for (int i =0; i<rows;i++){
            for(int j=0; j<5; j++){
                X[i][j]=100.00;
            }
        }

           pca =new PCA();
          demo = new DynamicDataDemo3("Sensor readings",5);
        demo.pack();
        RefineryUtilities.positionFrameOnScreen(demo, 1, 0);
        demo.setVisible(true);

        buffer = buf;

    while( (data = input.read()) != -1) {
    repaint();
    try {
    Thread.sleep(1000);
    } catch (InterruptedException ex) {
    ex.printStackTrace();
    }
    }
    } catch (IOException ex) {
    ex.printStackTrace();
    }
```

```
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        double average=0.0;
        g.setColor(Color.gray);
        g.fill3DRect(200, 200, 600, 600, false);

        drawBoard(g);

        intitaray();
        double[] meanzTemp=new double[5];
        int sensornumber=0;;

//start filling the matrix with real sensor data
        for (int i = 0; i < 10; i++) {
            data = buffer[i].getSensor_data();
            number = buffer[i].getSensor_number();

            switch(number){
                case 2:
                    if (a<rows){
                    X[a][0]=(double)(data/10);
                    System.out.println("a "+a);
                    a++;}
                    else
                        received1 =true;
                    // System.out.println("a "+a);
                    break;
                case 3:
                     if (b<rows){
                    X[b][1]=(double)(data/10);
                    b++;}
                     else
                          received2 =true;
                     //System.out.println("b "+b);
                    break;
                case 4:
                     if (c<rows){
                    X[c][2]=(double)(data/10);
                    c++;}
                     else
                          received3 =true;

                    //System.out.println("c "+c);
                    break;
                case 6:
                     if (d<rows){
                    X[d][3]=(double)(data/10);
                    d++;}
                     else
                          received4 =true;
                    // System.out.println("d "+d);
                    break;
                case 7:
                     if (e<rows){
```

```
                    X[e][4]=(double)(data/10);
                    e++;
                     }
                     else
                       received5 =true;
                     //System.out.println("e "+e);
                    break;

            }
            if (received1 == true && received2 == true && received3 == true
&& received4 == true && received5 == true) {



                X=(double[][])X;
                Matrix X1 = new Matrix(X);

                double[][] Xs= PCA.Standardize(rows,5 , X);
                Matrix X2 = new Matrix(Xs);






               double[][] Z0= PCA.principalComponents(Xs,0);
               double[][] Z1= PCA.principalComponents(Xs,1);
               double[][] Z2= PCA.principalComponents(Xs,2);
               double[][] Z3= PCA.principalComponents(Xs,3);
               double[][] Z4= PCA.principalComponents(Xs,4);

                LineChartDemo6 demo3= new LineChartDemo6("PCA0", Z0);
                demo3.pack();
                RefineryUtilities.positionFrameOnScreen(demo3, 1,0.9 );
                demo3.setVisible(true);

                                LineChartDemo6 demo5= new
LineChartDemo6("PCA1", Z1);
                demo5.pack();
                RefineryUtilities.positionFrameOnScreen(demo5, 1,0.9 );
                demo5.setVisible(true);

                                LineChartDemo6 demo9= new
LineChartDemo6("PCA2", Z2);
                demo9.pack();
                RefineryUtilities.positionFrameOnScreen(demo9, 1,0.9 );
                demo9.setVisible(true);

                                LineChartDemo6 demo7= new
LineChartDemo6("PCA3", Z3);
                demo7.pack();
                RefineryUtilities.positionFrameOnScreen(demo7, 1,0.9 );
                demo7.setVisible(true);

                                LineChartDemo6 demo8= new
LineChartDemo6("PCA4", Z4);
                demo8.pack();
```

```
        RefineryUtilities.positionFrameOnScreen(demo8, 1,0.9 );
        demo8.setVisible(true);


      Matrix W1 = new Matrix(OffsetAnalysis.Offset(rows, 5, X));

        Xw=X1.arrayTimesEquals(W1);

        if (initialize==false){
        xi[0][0]=meanrow(Xw,rows,5,0);
        xi[1][0]=0.0;

        k[0][0]=1;
        k[1][0]=0.0;

        ci[0][0]=1;
        ci[0][1]=0;

        ai[0][0]=1;
        ai[0][1]=1;//1;
        ai[1][0]=0;
        ai[1][1]=1;//1;

        p[0][0]=1;
        p[0][1]=1;
        p[1][0]=1;
        p[1][1]=1;

        s[0][0]=1;
        initialize=true;

        }
        Matrix P=new Matrix(p);
        Matrix K=new Matrix(k);
        Matrix S=new Matrix(s);
        Matrix S1=new Matrix(s1);
        Matrix A= new Matrix(ai);
        Matrix C= new Matrix(ci);
        Matrix x= new Matrix(xi);
        Matrix Xh= new Matrix(xh);
        Matrix Innovation=new Matrix(innovation);
        Xh=x;

        double[][] z= new double[rows][1];



        Matrix Y1= new Matrix(yi1);

        Matrix Y= new Matrix(yi3);
        Matrix Y2= new Matrix(yi2);


        for (int r=0; r<rows;r++){
// Kalman filter implementation

        xi[0][0]=meanrow(Xw,rows,5,r);
```

```
                Y1=C.times(x);
                yi2[0][0]=(varrow(X1,rows,5,r));
                wi[0][0]=Math.pow(yi2[0][0],2);






                Matrix W=new Matrix(wi);
                Y=Y1;
                Xh= A.times(Xh);
                Innovation=Y.minus(C.times(Xh));
                S1=C.times(P).times(C.transpose());
                S=S1.plus(W);
                // get kalman gain
                K=A.times(P).times(C.transpose()).times(S.inverse());

                Xh=Xh.plus(K.times(Innovation));

                FilterX=Xh.getArray();
                FilterX2[r][1]=FilterX[0][0];
                AverageX[r][1]=xi[0][0];


P=A.times(P).times(A.transpose()).minus(A.times(P).times(C.transpose()).times
(S.inverse()).times(C).times(P).times(A.transpose()));
                }

               LineChartDemo6 demo4= new LineChartDemo6("Kalman", FilterX2,
AverageX);
        demo4.pack();
        RefineryUtilities.positionFrameOnScreen(demo4, 1,0.6 );
        demo4.setVisible(true);


            received1 = false;
            received2 = false;
            received3 = false;
            received4 = false;
            a=0; b=0; c=0; d=0; e=0;
            }
            set_sensor_radius(number, data, g);


            if (number==2)
            demo.setData1((data/10));
            average=average+data/10;
            if (number==3)
            demo.setData2((data/10));
            if (number==4)
            demo.setData3((data/10));
            if (number==6)
            demo.setData4((data/10));
            if (number==7)
            demo.setData5((data/10));
        }


        }
    }

    public void getData() {

          repaint();
        animationTimer = new Timer(1, new LightHandler());
        animationTimer.start();

    }

    private class LightHandler implements ActionListener {

        public void actionPerformed(ActionEvent actionEvent) {
            int dataint;
            char datastring;

        }
    }
    public double meanrow(Matrix A, double nrow,double  ncol, int index){

        double[][] Atemp= A.getArrayCopy();
        double tempcol=0;
        double tot=0;

        for (int i=0; i<(int)ncol; i++)
             {
                tempcol = Atemp[index][i];
                tot += tempcol;
             }


         return tot / ncol;

    }

//Method to obtain the variance of rows
    public double varrow(Matrix A, double nrow,double  ncol, int index){
        double[][] Atemp= A.getArrayCopy();
        double tempcol=0;
        double colstdevs=0;


         for (int i=0; i<(int)ncol; i++) {
              tempcol = Atemp[index][i];
                colstdevs += Math.pow(tempcol-meanrow( A,nrow,ncol,index),
2.0);
             }
             colstdevs = Math.sqrt(colstdevs/ncol);
             if (colstdevs < 0.0001) { colstdevs = 1.0; }
             return colstdevs;
      }
```

```
        public double varcol(Matrix A, int nrow,int  ncol){
double[][] Atemp= A.getArrayCopy();
double[] colmeans = new double[ncol];
double[] colstdevs = new double[ncol];
double[] tempcol = new double[nrow];
double tot;
for (int j=0; j<ncol; j++)
    {
     tot = 0.0;
     for (int i=0; i<nrow; i++)
         {
            tempcol[i] = Atemp[i][j];
            tot += tempcol[i];
         }

           colmeans[j] = tot/(double)nrow;
     for (int i=0; i<nrow; i++) {
            colstdevs[j] += Math.pow(tempcol[i]-colmeans[j], 2.0);
         }
         colstdevs[j] = Math.sqrt(colstdevs[j]/((double)nrow));
         if (colstdevs[j] < 0.0001) { colstdevs[j] = 1.0; }
  }

return colstdevs[0];
  }

public double meancol(double[][] Atemp, int nrow,int  ncol){
double[] colmeans = new double[ncol];
double[] tempcol = new double[nrow];
double tot;
for (int j=0; j<ncol; j++)
    {
     tot = 0.0;
     for (int i=0; i<nrow; i++)
         {
            tempcol[i] = Atemp[i][j];
            tot += Math.abs(tempcol[i]);
         }

           colmeans[j] = tot/(double)nrow;
  }

return colmeans[1];
  }


public void set_sensor_radius(int sensor_num, int reading, Graphics g) {
    int distance = 0;
    int data_hop = 2;

    switch (sensor_num) {
        case 4:
                drawLightShade(2, 3, g, reading);

            break;
        case 2:
            drawLightShade(3, 1, g, reading);
```

```
                break;
            case 6:

                drawLightShade(4, 4, g, reading);
                break;

             case 3:
                distance=reading;

                drawLightShade(5, 5, g, reading);

                break;
            case 7:


                distance=reading;

                    drawLightShade(6, 3, g, reading);
                break;

        }

    }

    public void drawBoard(Graphics g) {


        g.setColor(Color.black);
        g.drawLine(200, 200, 800, 200);
        g.drawLine(200, 200, 200, 800);
        g.drawLine(200, 800, 800, 800);
        g.drawLine(800, 200, 800, 800);
        for (int i = 1; i < 6; i++) {
            g.drawLine((500 * i / 5) + 200, 200, 500 * i / 5 + 200, 800);
            g.drawLine(200, 500 * i / 5 + 200, 800, 500 * i / 5 + 200);
        }
    }

        public void drawLightShade(int row, int column, Graphics g, int rgb )
{
        int r;
                r=rgb-100;
                    if (r > 255){
                r=255;}
            else if(r <0){
                r=0;}


        g.setColor(new Color(r,r,r));

        g.fill3DRect((column + 1) * 500 / 5 + 2, (row + 1) * 500 / 5 + 2, 500
/ 5 - 2, 500 / 5 - 2, true);
    }
    public void intitaray() {
        for (int j = 0; j <= 6; j++) {
            for (int i = 0; i <= 6; i++) {
                element[i][j] = new GridElement();
            }
        }
    }
}
```

```
package lightsensor;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Timer;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JPanel;

import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.axis.ValueAxis;
import org.jfree.chart.plot.CombinedDomainXYPlot;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.StandardXYItemRenderer;
import org.jfree.data.time.Millisecond;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
//import org.jfree.ui.Spacer;

//this class displays the sensor readings being captured in real time

public class DynamicDataDemo3 extends ApplicationFrame implements
ActionListener {

    public  int SUBPLOT_COUNT;

    private TimeSeriesCollection[] datasets;
    private XYSeriesCollection[] datasets2;
     private Timer animationTimer;
    private double[] lastValue;
private int data;
    private int data1;
    private int data2;
    private int data3;
    private int data4;
    private int data5;
    private double[][] data6;
    private double average;
    private int sum=0;
    /**
     * Constructs a new demonstration application.
     *
     * @param title  the frame title.
     */
    public DynamicDataDemo3(final String title, int subplotnum) {


        super(title);
        SUBPLOT_COUNT=subplotnum;
         lastValue = new double[subplotnum];
                final CombinedDomainXYPlot plot = new
CombinedDomainXYPlot(new DateAxis("Time"));
        this.datasets = new TimeSeriesCollection[SUBPLOT_COUNT];
            String Y= new String("Y");
        for (int i = 0; i < SUBPLOT_COUNT; i++) {
            this.lastValue[i] = 100.0;
            final TimeSeries series = new TimeSeries("SENSOR " + i,
Millisecond.class);
                        this.datasets[i] = new TimeSeriesCollection(series);
            final NumberAxis rangeAxis = new NumberAxis(Y + i);
            rangeAxis.setAutoRangeIncludesZero(false);
            final XYPlot subplot = new XYPlot(
                    this.datasets[i], null, rangeAxis, new
StandardXYItemRenderer()
            );
            subplot.setBackgroundPaint(Color.lightGray);
            subplot.setDomainGridlinePaint(Color.white);
            subplot.setRangeGridlinePaint(Color.white);
            plot.add(subplot);
        }

        final JFreeChart chart = new JFreeChart("SENSOR READINGS", plot);
              chart.setBorderPaint(Color.black);
        chart.setBorderVisible(true);
        chart.setBackgroundPaint(Color.white);

        plot.setBackgroundPaint(Color.lightGray);
        plot.setDomainGridlinePaint(Color.white);
        plot.setRangeGridlinePaint(Color.white);
          final ValueAxis axis = plot.getDomainAxis();
        axis.setAutoRange(true);
        axis.setFixedAutoRange(60000.0);  // 60 seconds

        final JPanel content = new JPanel(new BorderLayout());

        final ChartPanel chartPanel = new ChartPanel(chart);
        content.add(chartPanel);

        final JPanel buttonPanel = new JPanel(new FlowLayout());


        final JButton buttonAll = new JButton("START");
        buttonAll.setActionCommand("START");
        buttonAll.addActionListener(this);
        buttonPanel.add(buttonAll);



        content.add(buttonPanel, BorderLayout.SOUTH);
        chartPanel.setPreferredSize(new java.awt.Dimension(600, 470));
        chartPanel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
        setContentPane(content);
    animationTimer= new Timer(1000, new ActionListener(){
        public void actionPerformed(final ActionEvent e) {

        for (int i = 0; i < SUBPLOT_COUNT; i++) {
                final Millisecond now = new Millisecond();

                lastValue[i] = lastValue[i] * (0.90 + 0.2 * Math.random());

lastValue[i]);
                if (i<5)
                 datasets[i].getSeries(0).add(new Millisecond(), getData(i));
                else
                    datasets[i].getSeries(0).add(new Millisecond(),
getData());

        }



    }
    });


    }

    public void setData1(int dat){


        data1=dat;

    }



       public void setData2(int dat){


        data2=dat;

    }
```

```
   public void setData3(int dat){


    data3=dat;

}


   public void setData4(int dat){


    data4=dat;

}


   public void setData5(int dat){


    data5=dat;

}


public int getData(int i){
    if (i==0){
        sum=sum+data1;
    return data1;}
    else if (i==1){
         sum=sum+data2;

    return data2;}
    else if (i==2){

     sum=sum+data3;
    return data3;}
    else if (i==3){

     sum=sum+data4;
    return data4;}
    else {
        sum=sum+data5;
    return data5;
    }
}


public double getData(){

    average=(double)sum/5;
    sum=0;
return average;
}


public void actionPerformed(final ActionEvent e) {
        if (e.getActionCommand().equals("START")) {
            animationTimer.start();
            }
        }


}
```

```
package lightsensor;

import java.awt.Color;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.xy.XYDataset;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;

// this class displays  a chart of the kalman filter and  PCA
public class LineChartDemo6 extends ApplicationFrame {
    private XYSeries series2;
    private XYSeriesCollection dataset;
    private XYSeries series1;



 public LineChartDemo6(final String title, double[][] data) {

    super(title);
    series1 = new XYSeries("Resilient Average");

    for (int i = 0; i < 20; i++) {
        series1.add(i,data[i][1]);

    }
    dataset = new XYSeriesCollection();
    dataset.addSeries(series1);


    final JFreeChart chart = createChart(dataset);
    final ChartPanel chartPanel = new ChartPanel(chart);
    chartPanel.setPreferredSize(new java.awt.Dimension(600, 270));
    setContentPane(chartPanel);

    }

     public LineChartDemo6(final String title, double[][] data, double[][]
average) {
        this(title,data);
        series2 = new XYSeries("Simple Average");
        for (int i = 0; i < 20; i++) {
                                series1.add(i,data[i][1]);
                series2.add(i,average[i][1]);
            }

        dataset.addSeries(series2);

    }

    private JFreeChart createChart(final XYDataset dataset) {

                final JFreeChart chart = ChartFactory.createXYLineChart(
            " ",      // chart title
            "Window",                     // x axis label
            "Network Reading",                     // y axis label
            dataset,                 // data
            PlotOrientation.VERTICAL,
            true,                    // include legend
            false,
            false
        );

        chart.setBackgroundPaint(Color.white);


        final XYPlot plot = chart.getXYPlot();
        plot.setBackgroundPaint(Color.lightGray);
        plot.setDomainGridlinePaint(Color.white);
        plot.setRangeGridlinePaint(Color.white);

        final XYLineAndShapeRenderer renderer = new
XYLineAndShapeRenderer(true,false);
        plot.setRenderer(renderer);

        final NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
        rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());

        return chart;

    }

}
```

VITA

Ricardo Aguirre was born in Guayaquil, Ecuador. He began his Bachelor degree in Escuela Superior Politecnica del Litoral (ESPOL) and finished his degree at UNO where he obtained his Bachelor degree in Electrical Engineering in 2006. His interests in high level programming languages made him pursued a Master degree in Computer Science.