Delft University of Technology
Master's Thesis in Computer Science

Distributed Kalman Filtering using Broadcast Gossip

A.C. Kodde

Distributed Kalman Filtering using Broadcast Gossip

Master's Thesis in Computer Science

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

A.C. Kodde
a.c.kodde@student.tudelft.nl

August 27, 2012

Author
A.C. Kodde (a.c.kodde@student.tudelft.nl)

Title
Distributed Kalman Filtering using Broadcast Gossip

MSc presentation
October 1, 2012

Graduation Committee

Prof. dr. K.G. Langendoen	Delft University of Technology
Dr. S. Dulman	Delft University of Technology
Dr. J. Sijs	TNO Technical Sciences
H.-G. Gross, PhD	Delft University of Technology

Abstract

For many processes it is required to have a reliable view of an environment of interest. One way to achieve this is by performing a distributed Kalman filter. In this thesis, three distribution methods from different research backgrounds are implemented and evaluated using multiple metrics for use in a Gossip based wireless sensor network: consensus, weighted consensus and covariance intersection. The modular solution makes it possible to easily switch between the different implemented distribution methods and Gossip algorithms. From the evaluation based on metrics like the correctness of the estimate and the agreement among the different nodes, it follows that the naive consensus algorithm does not perform well. The weighted consensus and covariance intersection perform both with errors smaller than two degrees Celsius. However, the weighted consensus does require assumptions that covariance intersection does not.

Acknowledgements

I would like to express my very great appreciation to my daily supervisor at TNO, Dr. Sijs for his valuable and constructive suggestions and all his support throughout this research work. I would also like to thank Dr. Dulman for his useful critiques and patience.

I would like to offer my special thanks to Martin Kodde whose insight and ongoing motivation proved invaluable.

Finally, I wish to thank my parents, family and friends for their support throughout my study.

A.C. Kodde

Delft, The Netherlands
August 27, 2012

Contents

Chapter 1

Introduction

In many processes, it is required to know as much as possible about a given environment of interest. For example, in a greenhouse knowing the temperature as fine grained as possible throughout the building is very important for the plants to grow optimally. For such a process, it is for example needed to have early warnings when temperatures get to low and to have a history of measurements to prevent liabilities.

However, a fine grained measurement setup will result in a large amount of data that requires processing and communication. When using a central processing unit to process all data, this will quickly become a bottleneck in the network of sensors. By switching to a decentralized setup, these kind of bottlenecks can be alleviated. For example, by processing information at the sensor location itself, data does not have to be transmitted and can be shared with neighbors to increase the reliability of the estimates. Furthermore, by using sensors that do not require cables, so called wireless sensor nodes, deployment of a fine grained network can be achieved with minimal effort.

Wireless sensor network are composed out of nodes that are small and cheap to manufacture, have limited processing power, are able to sense their environment and transmit this information. The use of small sensor nodes with network capabilities allow for the gathering of information from any environment, without the need of fixed or expensive infrastructures. Wireless Sensor Networks provide huge opportunities for monitoring environments. Due to the distributed nature of wireless sensor networks, measurement devices can easily be deployed over large areas and provide real-time data about the state of the object under consideration [2].

In the last couple of years, it has become possible to use wireless sensor networks for real life situations. Many Wireless Sensor Networks applications are already implemented and tested in the field, like Structural Monitoring [7], Volcano Monitoring [39] and Habitat Monitoring [35]. With the ongoing research into wireless sensor networks and the minification of the hardware of individual nodes, making the platforms smaller and power friendlier, future applications of this like smart dust are envisioned in all kinds of environments [37], like intelligent vehicles, climate control or public health monitoring.

Sensors obtain noisy measurements and it is the information embedded in these measurements that is needed for real-world applications [9]. Without any form of processing, the measured data will show fluctuations as if a random variable is added to the signal. One of the methods that is available to remove noise and smooth the data from sensors, is the Kalman filter developed by Kalman in 1960 [16]. This filter is able to remove noise from measured data coming for different sensors, only using the previous measured data. This fact makes it very interesting for wireless sensor nodes, where memory is scarce.

In many of the wireless sensor network applications, including this thesis work, it is not enough to only forward the sensor data to a central gathering point, but rather online processing needs to be performed on the sensor data. Because wireless sensor networks have limited capabilities, a centralized estimation solution is suboptimal. In many wireless sensor networks, cooperation between the different nodes is a must [10]. In this kind of networks, the communication is the most expensive operation a node can make [27]. Minimizing the amount of data that needs to be transferred is a must. Using an adaptation of the Kalman filter, the distributed Kalman filter [25], the traffic between sensor nodes can be minimized. Therefore, it seems that Distributed Kalman Filters and Wireless Sensor Networks are a natural fit to each other. Such has been proposed and proved in literature before [26], [33], [24]. However, the different ways of distributing the Kalman filter has been split into two main groups: the “synchronization” group on one side and the “fusion” group on the other side. As these two ideas come from two different research communities, a survey comparing both methods using wireless sensor networks has not been conducted.

In this thesis work, the goal is to estimate distributed parameters using distributed Kalman filtering and current wireless sensor network techniques and assess the implication of different distribution methods using Gossiping. Gossiping is a way of communicating between two or more nodes over an unreliable medium to exchange information. As a test case, the temperature profile of a room will be measured using multiple wireless sensor nodes at different locations in a closed room. All nodes will continuously monitor the temperature in the room in such a way, that information is available for various spatial locations in the testing environment at all times. In short, the solution as proposed in this work will be able to perform a distributed estimation of the temperature profile with the following properties.

- Frequent updates of the temperature information;

- Coverage over the entire room;

- Reliable for failures;
- Cheap.

Given the current level of computing power available in sensor nodes and their communication abilities, the combination of Gossip and distributed Kalman Filtering will result in a feasible solution abiding to aforementioned specifications.

In the next paragraph the problem statement and research questions are stated.

1.1 Research Questions

The problem statement of this thesis is: What are the different methods of implementing Gossip Based Distributed Kalman Filtering and how does it perform.

From this, the following research questions can be distilled.

1. What form of Gossip is best suited for Gossip based distributed Kalman Filter?

   There are several forms of gossiping available in practice and literature [5, 32], each with their own strengths and weaknesses. A proper choice of a gossiping implementation is important as this influences all aspects of the distribution of a distributed Kalman filter. In Chapter 2, we present a survey of applicable gossiping methods and select the method most appropriate for distributed Kalman filtering, while chapter 3 shows the implementation details.

2. What are the different ways of distributing information in a distributed Kalman filter and how do they compare?

   It is possible to share information between different nodes in a network in many different ways. The two important methods are consensus and covariance intersection, which differ in the way the information is merged into the local state. As the distribution method is an important factor in the way the distributed Kalman filter works, it must be taken into account. In chapter 2 the two methods are explained and compared extensively. Chapter 3 will discuss the way the two methods are implemented and the differences between them. Experimental results will be presented in chapter 4.

3. What limitations are encountered when implementing Gossip based Distributed Kalman Filter and how can one cope with those? *In this thesis we present an implementation of a modular Gossip based distributed Kalman filter for wireless sensor nodes to experimentally verify*

the fitness of these technologies. However, several limitations of the platform can influence the end result. Chapters 3 and 4 will discuss these.

1.2 Organization

This Master's is organized as follows. In chapter 2 the background and related work of wireless sensor networks, Kalman filtering and the distribution of the Kalman filter will be discussed. In chapter 3 the implementation of the distributed Kalman filter based on gossiping will be discussed in detail along with the problems that are encountered during this implementation. Chapter 4 will show the results of the distributed Kalman filter using different distribution methods for the distributed Kalman filter and will perform a survey on the differences between them. The final chapter 5 will answer the research questions and propose future work.

Chapter 2

Related Work

In this thesis, the goal is to run a distributed Kalman filter using measurements from multiple sensors spread out in a grid. In the practical sense, a distributed Kalman filter running in a Wireless Sensor Network will estimate the temperature distribution in a room by exploiting measurements obtained by the deployed sensor nodes. This chapter will outline the techniques that are used to attain this goal. it will describe the distributed system and how the networked system can be setup. Furthermore, the different algorithms that can be used to distribute the Kalman filter will be discussed. The Kalman filter itself will also be explained along with the model that is used in this filter.

2.1 Gossiping in Decentralized Networks

Centralized networks have been in use for several decades now. In this type of networks a central node is in charge of all decisions in a network. For a central node to make these decisions, it needs to acquires all information in the network, i.e. it knows the global state. On small networks, having this central node is not a problem. When trying to upscale the network, such a node can become a bottleneck [29]. as communication requirements become impractical, especially for wireless sensor networks. In addition, having a single central node is also a single point of failure. If this node fails for whatever reason, the network has to cope and maybe even reorganize itself, reducing the availability of the network.

To alleviate the burden on a central node, several options exist. One of these options is to divide the network in several clusters, splitting the network essentially in smaller portions [1]. While this works, it still brings a burden on the cluster head. Especially in wireless sensor networks, having one node do more work than other nodes, is suboptimal [5]. For example, having it do more work drains its battery faster than surrounding nodes, making the network die out unbalanced and losing coverage or connectivity.

Several ways to reduce this imbalance have been proposed, like switching cluster head regularly [12], but this requires again communication, which is again suboptimal. Furthermore, having clusters will increase the number of nodes that can fail, but still reorganization is required after node failure.

The idea behind decentralized networks is to have no central node. Instead, all nodes perform the same operation, making the speed of draining of the battery more even across the network [2].

Examples of applications of decentralized networks are a network of nodes with the goal to calculate the size of the network, either in number of nodes or in spatial extent by calculating the convex hull [28]. Another example would be to achieve consensus within a network. In consensus, all nodes in a network must more or less agree on a specific value [30]. An example of this is temperature, where every node in a network knows its local temperature. By exchanging messages, the nodes can reach a consensus on the (average) room temperature.

With the ongoing minification of hardware and progress in wireless communication, wireless sensor networks have become off the shelve products. A wireless sensor network is made out of many inexpensive nodes, where each node has processing power, sensing capabilities - like temperature or light - and wireless communication [27]. Where previously nodes had to have a static connection, now ad-hoc networks intersensor communication is possible. This greatly reduces the effort required for installation and deployment. For instance, it is now possible to drop the sensors out of a plane and, once landed, start sensing their environment and send their results to a base station.

Wireless sensor networks have many advantages over regular networks. Since the nodes are small and portable, it is relatively easy to deploy the network. Furthermore, most wireless sensor networks are designed in such a way, that the network can organize itself. For example, the nodes can cooperate to find the shortest path to the base station. Having many small nodes also increases the coverage of a system. As in [39], having only a few expensive measurement systems does not give the fine grained results that a wireless sensor network can deliver. Since the nodes are small, usually battery powered and possess only a little processing power, they are cheap. Hence, it is possible to deploy many sensors at minimal costs. Of course, wireless sensor networks do have some disadvantages as it comes with certain drawbacks and constraints. Most of these constraints are a direct result from the limited power on board a single node.

- Sending a single message is orders of magnitude more power consuming then calculations. One of the goals of a wireless sensor network design is therefore to limit the number of messages transmitted. An additional benefit of reducing the number of messages is that with less messages, the chance of message collisions is smaller and network reliability increases.

- With wireless sensor networks, the density of the network is also a problem. When many nodes are in each others communication range, which can happen for a fine grained measurement setup, the chance of message collision or contention increases again. This of course is a consideration that must be made before deployment.

- A final drawback of the lack of power is that positioning is very hard. For instance, the power requirements of GPS make it unusable in wireless sensor networks [2].

In this work we are interested in distributed Kalman filtering. To do this there is a need for a method to exchange information efficiently between nodes. Over time, different methods have been developed for communication, one of which is gossiping [32]. The main benefit in gossiping is the lack of network organization, lack of required global state and relatively low message count. Hence, gossiping is ideal for use in decentralized networks [5].

The main idea of gossiping is based on the natural phenomenon of rumor spreading in a community, hence the name. Gossiping is generally used for two objectives. The first objective is dissemination of various types of information across a network. The second objective that gossiping can achieve is consensus. In consensus, every node already has the information, but it wants to agree on it with the entire network [18].

For the two objectives mentioned earlier, different gossiping schemes have been invented. Among other, these implementations differ in their network coverage, their energy efficiency, latency and overhead.

- *Network coverage* is the effectiveness of the algorithm to spread its information to as many nodes as possible, preferably to all nodes in the network.

- The *network efficiency* metric is mostly concerned with the number of messages that need to be exchanged, since communication is the main power drain.

- The *overhead* metric is about the number of messages that is not effectively transferring data, but instead shares information about the algorithm itself.

- In consensus gossiping another important metric is the *convergence rate*, which is the rate in which all nodes (approach) the network wide average.

Metric	Network Coverage	Energy Efficiency	Latency	Convergence rate	Overhead
Gossip	+	0	-		0
Push&Pull	++	-	−		−
Randomized Gossip	+	0	−	+	+
Broadcast Gossip	++	+	+	++	++
Geographic Gossip	+	−	-	-	-

Table 2.1: A comparison of the different Gossip algorithms with their respective properties.

The gossiping strategies currently proposed differ in their criteria for selecting to which neighboring node data should be sent

The main way to influence these metrics, is by tuning the selection criteria which select the nodes who will receive a message. For example, by sending a message to all nodes in range instead of to a single node, the local values among the different nodes in the network might be able to convergence quicker. See Table 2.1 for an overview of the different metrics for the different gossip algorithms. The different gossip algorithms will be discussed in depth next.

For data dissemination, the original "Gossip" algorithm is the most used, along with its extensions. Another important gossip algorithm is Push&Pull. Both dissemination algorithms (with their extensions) will be discussed in the subsequent sections. For consensus gossiping, there are several other algorithms. Three of these consensus algorithms - Randomized Gossip, Geographic Gossip and Broadcast Gossip which are popular in literature, will be discussed in subsequent sections.

The first algorithm, **Gossip** [13], can be used to spread one piece of information across a network in such a way that most of the network will receive this message. The goal of this algorithm therefore is to spread a message in an efficient matter to most of the nodes. Gossip is able to decrease the number of messages send by up to 35%, while still reaching almost all nodes.

In Gossip, the algorithms starts with flooding, i.e. always forwarding messages, for the first few time steps to make sure the gossiping is started up correctly. After this initialization phase, the node will retransmit any incoming message with probability p_1 after waiting for a predefined time period of T. Messages that have been received during the waiting time T, will be ignored. In practice, this means that the Gossip algorithm is nothing more than a flood algorithm with a way to reduce the flooding. If $p_1 = 1$, then the algorithm is just a flood algorithm [18].

An extension to Gossip is proposed in [13]. This extension not mentioned in table 2.1 adds a small enhancement to maximize the coverage. If a node has a small number of neighbors, it will increase its probability, making the probability to send a message higher. This makes the coverage of Gossip2 slightly better than the original Gossip. The coverage can still not be guaranteed as only the probability is increased. To make the probability even higher, a second extension is presented in [13].

Another extension to Gossip from [13] includes a method for better coverage. When a node has decided not to transmit, it will listen to the messages spread by its neighbors, waiting for at least m copies of the message. If the message is not overheard at least m times, the node will transmit the message anyway. In this way, the Gossip3 algorithm is actively expanding the coverage by sending its message when the coverage is measured to be low.

Push&Pull gossiping [17] builds further on the ideas from the previous gossip algorithm. Developed at Berkeley, its goal is to try and minimize the number of update rounds required to spread a distributed database over the entire network. Push&Pull is developed to be more robust for failures and for it to handle node failures in a sparse network.

The Push&Pull algorithm is split into two phases. In the first phase, a message along with an age counter is spread out to a random neighbor, i.e. pushing a message to a random neighbor. In this phase, also called the exponential growth phase, every iteration doubles to number of nodes that known the message. When the message is of a certain age, the node switches to the pull phase and only spreads its message when it is called upon by a neighbor. This phase called the quadratic shrinking phase spreads the message even faster since the number of nodes that are uninformed will decrease in a squared fashion.

Randomized gossip [6] is an algorithm that is used for consensus, where every node already has a value and the network only has to agree on a global value. The name randomized Gossip comes from the fact that the algorithm will choose a receiving node at random.

The procedure is as follows. In every time step, every node will wake up with probability 0.5. If a node -j - is active, it will contact at most one neighbor with probability $1/d$, where d is the number of neighbors. The neighbor of choice will be selected at random, while active nodes ignore any request. If an inactive node k gets a request from and only from j, then node k sends back its values, after which both j and k take the pair wise average of the received values.

Geographic Gossip [8] is an extension on randomized gossip with the addition of geographical information. With this addition, a basic assumption is made that the nodes know the location of all other nodes. As noted in [6], randomized gossip computes pair wise averages with their one hop neighbors, making the convergence speed in typical wireless sensor network layouts slower than necessary. Geographic gossip therefore combines geographic routing with gossiping. In each gossip round, a node selects a location (y) in the network to send a message to. The location picked can be anywhere in the network, requiring multiple hops to reach. The sender node picks a neighbor closest to y and sends the message. This node will forward to message to the next node closest to y, unless itself is closest to y. If the receiving node accepts the package, the receiving node will return a message with its local values. With their values shared among them, both nodes will calculate their new values by taking the pair wise average of the received value and its local values.

Broadcast gossip [4, 3] is specifically designed for wireless sensor networks that want to achieve consensus. It keeps in mind that the medium used in such a network is the air, and that every message send, can be received by all neighbors, whether they were addressed or not. Furthermore, the required two way of gossip (i to j, j to i) means that the chances for negative effects, e.g. packet loss, of wireless transmissions are doubled. In broadcast gossip, a node wakes up with a certain probability and when awoken, it broadcasts its state to all its neighbors. All the neighbors that receive the message, update their interval values with a mixing parameter and the gossip algorithm stops. In [4] it is shown that the values do convergence with probability one, although the final calculated value is not exactly the network wide average.

The gossip algorithms discussed in the previous sections will be used to distribute a Kalman filter over a network of nodes. In the next section, the distribution process of the Kalman filter will be discussed.

2.2 Distributed Kalman Filtering

Kalman filtering is a technique to get reliable information from noisy inputs, like sensors that have results with noise. By applying the Kalman filter, results from these sensors can be used for all kinds of applications like navigation, climate control and motion control. Usually, Kalman filtering a central node that performs all calculations [38] is used. In this thesis, the goal is to eliminate this central node and distribute the Kalman filter over all nodes.

In distributed Kalman Filtering, it is necessary to have communication between the different nodes in a network. An issue in wireless sensor network is that communication is expensive in terms of battery power and life time of the network [5]. The number of messages depends on the communication protocol, the topology and connectivity of the network, but also on the amount of information that needs to be spread.

The following sections will discuss the Kalman filter algorithm, along with the different ways to distribute this algorithm in a efficient way suitable for wireless sensor networks.

2.2.1 General Principles of Kalman Filtering

Sensors, especially those on wireless sensor network nodes like the TelosB, give noisy measurement values. The measurement values behave as if a stochastic random variable is added to the measured value. To alleviate this problem, some sort of filtering can be performed to remove this stochastic noise.

To perform this filtering for static processes, least squares estimations needs to be performed which is able to estimate unknown parameters with stationary random variables [19].
To estimate n unknown parameters with l measurements, the basic equation to characterize the measurement is:

$$z = H \times x + v \tag{2.1}$$

Where the vector x is the n-dimensional vector representing the unknown parameters or the state. The l-dimensional vectors v and z represent the observation noise and the measurement respectively. The matrix H of size $l \times n$ is the design matrix and describes the relationship between the measurements and the parameters. The noise vector v is a stochastic random variable characterized by a Gaussian probability density function. Equation 2.1 will most likely be inconsistent due to the noise added by the vector v. Therefore, the stochastically best estimate $\hat{x}$ is the unbiased best estimate representing x.

As it is not possible to model every aspect of a process, the way forward is to find a best estimate of $\hat{x}$, along with the correctness of this estimation represented as the covariance matrix P. This matrix gives information about the covariance of the estimation of $\hat{x}$, i.e. the "correctness" of the estimation. The minimization problem to find $\hat{x}$ can be written down in equation form as seen in equation 2.2.

$$v^T R^{-1} v = (z - H\hat{x})^T R^{-1} (z - H\hat{x}) \tag{2.2}$$

In equation 2.2, R is a matrix representing the weight given to the observations, as defined in equation 2.3. Reducing this equation results in the equation as can be seen in equation 2.4.

$$R = cov(v) \tag{2.3}$$

$$\hat{x} = (H^T R^{-1} H)^{-1} H^T R^{-1} z \tag{2.4}$$

To get the covariance matrix P describing the error in this estimation, the equation in 2.5 can be used:

$$P = (H^T R^{-1} H)^{-1} \tag{2.5}$$

It can be proven that the matrix P is the optimal covariance matrix, meaning that $\hat{x}$ is the best linear unbiased estimation of the parameters given by z [21].

It is possible to find an estimation for $\hat{x}$ that is the best given the parameters. However, for this estimation, it is required for the least squares estimator to have access to all measurement values, past and current. On a wireless sensor node, this becomes a problem with regards to storage space and computational power: for every new measurement, all previous measurements need to be accessed and incorporated.

One way to solve this problem, is by converting the least squares estimator into a recursive algorithm. In [19], it is shown that this is indeed possible. The equations for such a recursive estimator, are shown in equations 2.6 to 2.8.

$$K_j = P_{j-1} H_j^T (H_j P_{j-1} H_j^T + R_j)^T \tag{2.6}$$

This K, called the gain matrix describes the gain achieved between the new measurement (z_j) with respect to the previous estimation (x_{j-1}) in sample instant $j-1$. Using this gain, it is possible to calculate the new states without using all previous measurements directly.

The equation for the covariance matrix P and estimation vector x are given in equations 2.8 and 2.7.

$$\hat{x}_j = \hat{x}_{j-1} + K_j (z_j - H_j \hat{x}_{j-1}) \tag{2.7}$$

$$P_j = (I - K_j H_j) P_{j-1} \tag{2.8}$$

Starting with initial estimation vector $\hat{x}_0$ and covariance matrix P_0, this recursive algorithm is able to find the best possible linear estimation for every $j > 0$ [19].

For the least squares estimator to work using a non-stationary random process, i.e. the state vector and stochastic behavior depends on time, the least squares estimator needs to be extended. The Kalman filter is such an extension, where each measurement is taken at discrete times denoted as k. The Kalman filter is able to use multiple (noisy) input sensors, while still able to calculate the optimal linear estimation. Kalman filtering uses a matrix that describes the relationship between two consecutive state vectors, i.e. the transition matrix. The equation for this relationship is given in equation 2.9.

$$x_k = Ax_{k-1} + w_{k-1} \tag{2.9}$$

The transition matrix A here is a $n \times n$ matrix and w is a n-dimensional vector representing the noise. This vector follows a Gaussian distribution with zero mean and has an $n \times n$ covariance matrix Q. This vector and matrix describe the noise caused in the system by modeling uncertainties.

The gain calculation, or Kalman Weight, remains mostly the same equation as the gain calculation in the recursive least squares, as can be seen in equation 2.10.

$$K_k = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} \tag{2.10}$$

However, the vector x and matrix P cannot be adopted from the recursive least squares algorithm, since these two matrices change over time. Instead, the vector x and matrix P need to be predicted using the dynamic model given earlier in equation 2.9. These new equations are given in equations 2.11 to 2.14.

Time update equations of the Kalman filter are given in equations 2.11 and 2.12

$$\tilde{x}_k = A\hat{x}_{k-1} \tag{2.11}$$

$$\tilde{P}_k = AP_{k-1}A^T + Q \tag{2.12}$$

Measurement Updates equations are given in equations 2.13 and 2.14

$$\hat{x}_k = \tilde{x}_k + K_k(z_k - H_k\tilde{x}_k) \tag{2.13}$$

$$P_k = (I - K_k H_k)\tilde{P}_k \tag{2.14}$$

As can be seen in the equations, the Kalman filter is split up into two separate steps. These two steps alternate, giving the recursive nature of the Kalman filter. In the first step, the time update (a priori) is performed to make a prediction of both the state vector and the covariance matrix based on the previous state. In the next step, the measurement update (a posteriori), this prediction is updated with the current value from the sensors. This process is shown in Figure 2.1.

The measurement update uses the Kalman weight to determine the correctness of the prediction. This Kalman weight, represented as the matrix K, is the weight that determines how much a prediction can be trusted. If the measurements are very accurate ($R \approx 0$) or the predicted state is of very low accuracy (for example because of a high Q), the matrix K will become the identity matrix I. From equation 2.13, it follows that the new measurements will be highly weighted, yielding that the estimated state is mainly determined by the current measurements rather than the predictions from the process model. However, when the measurements are not accurate or the process noise is very low, the K matrix will become 0. The result of

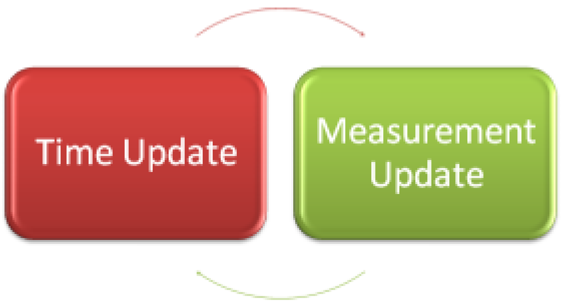


Figure 2.1: The Kalman process, an alternating sequence of measurement and time updates.

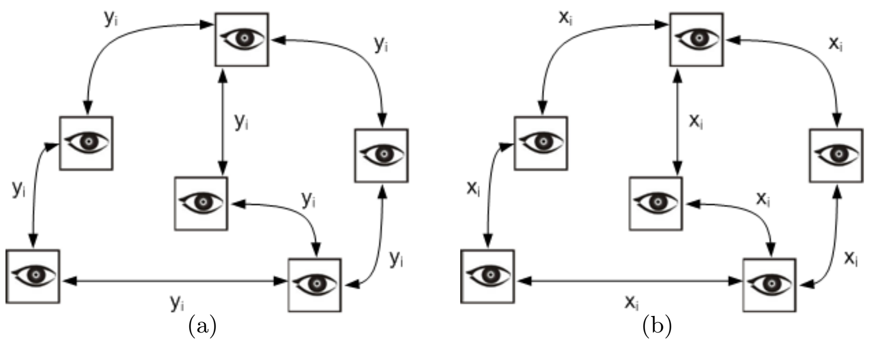


Figure 2.2: A wireless sensor network exchanging information. (a) Measurement exchange; (b) Estimation exchange.

this weighing is that the prediction is weighted at a high level, ignoring the measurement completely.

In the following sections, the centralized Kalman filter will be adapted to a decentralized Kalman filter.

2.2.2 Adaptation to Distributed System

A central system requires a single node to perform all work. As discussed earlier, this approach has its drawbacks since it creates a single point of failure and places a bigger burden on a single node.

The same holds for Kalman filtering, where in a centralized implementation, one node will perform all the filtering. If all nodes need information produced by the Kalman filter, the results should then be transmitted back to the all other nodes. This is suboptimal and a distributed Kalman can cope with these problems. In figure 2.2, two wireless sensor networks are depicted performing a distributed Kalman filer. In distributed Kalman fil-

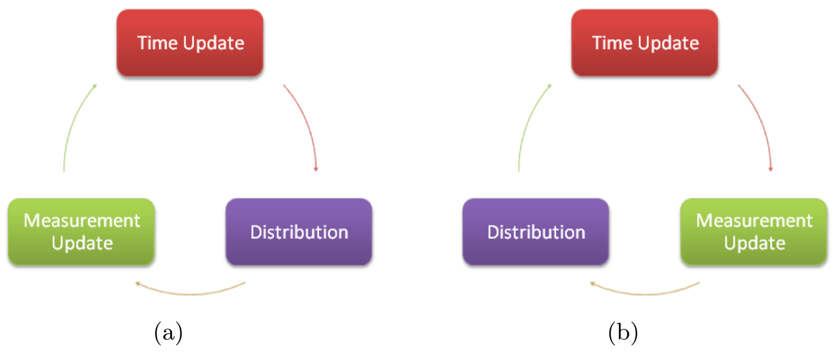


Figure 2.3: The distributed Kalman process, including the same steps as shown in figure 2.1, now including a distribution step. (a) Measurement exchange; (b) Estimation exchange.

tering, an extra step in the Kalman process is added, called the Distribution step. In this step, the distribution of the Kalman filter takes place. The extra step is shown, along with the usual Kalman filtering steps in figure 2.3. There are different ways to perform a distributed Kalman filter, each with their own advantages and drawbacks. This section will outline this different approaches to distributed Kalman filtering.

The most straightforward method to distribute a Kalman filter is by using **Measurement Exchange**, depicted in figure 2.3(a). In this distribution model, the measurements y_i of every node is transmitted to its neighbors. This received measurement is then used in the receiving node to calculate the new estimation x using the standard Kalman filtering equations. This mechanism however has several drawbacks. One of these drawbacks is in the fact that the measurement y_i will be processed multiple times by different nodes. Furthermore, for a node to process the information contained in y_i, the model describing the linear process needs to be exchanged as well. The final major drawback is that y_i does not necessarily contain all information needed to process y_i [34]. Because off all these drawbacks, measurement exchange will not be further explored in this work.

The logical step from exchanging measurements, is **Estimation Exchange**, depicted in 2.3(b). In this method, the measurements y_i are not exchanged, but instead the processed estimates x_i are, solving the problems of measurement exchange: the local measurement y_i is only processed by a single node which has to proper knowledge to do so.

In estimation exchange, two mechanism exists in the way estimates are merged into the local state: synchronization and fusion.

Usually, synchronization using estimation exchange is achieved via **consensus**, where the algorithm tries to minimize the differences between the different estimates x_i for all nodes in the network [24]. In principle, consensus is achieved by performing several rounds of weighted averaging to come to a point where $x_i - x_j$ is minimal. In 2.15 this algorithm is shown. In [14] it is shown that this weighted averaging is able to calculate a value that is the average of all initial values of the individual nodes in the network.

$$\hat{x}_i = (1 - w)x_i + wx_j \tag{2.15}$$

The primary objective from which consensus strategies were designed was the need for clock synchronization across a network. From this, it followed that it can also be used to synchronize other values, such as local estimates x. The downside of using synchronization for distributed Kalman filter is however, that the error of the estimates are synchronized as well, instead of reducing it [34]. Local estimates therefore become less accurate because of the synchronization. In addition, local covariances are not synchronized. As a result of this, the Kalman gain cannot benefit from the exchanged information.

The second form of estimation exchange is **fusion**. In fusion, the local covariance matrices are exchanged as well when merging the estimates. Fusion therefore can use this information to remove uncertain areas of the received information. The main fusion method is **covariance intersection** [15]. Covariance intersection makes use of a weight that determines the way two estimates are merged. To be able to cope with unknown covariances, several algorithms for these weights have been developed. The algorithm as described in [23] is one of the most popular algorithms.

Covariance intersection performs the fusion in three steps. These three steps are:

1. Determining the mix weight;
2. Calculating the new covariance matrix P;
3. Calculating the new estimation vector x.

The first equation, responsible for calculating the weight is given in Equation 2.16.

$$W_m = \frac{\mathrm{Tr}(P_m)^{-1}}{\sum_{i=0}^{n} \mathrm{Tr}(P_i)^{-1}} \tag{2.16}$$

The result of equation 2.16 gives for every arbitrary covariance matrix a value $0 \leq W_m \leq 1$ and $\sum_{m=0}^{n} W_m = 1$.

The next step is applying this weight to both the covariance matrix P_m and estimation vector x_m. The equations for this are given in equations 2.17 and 2.18.

$$P_m^{-1} = \sum_{i=1}^{n} W_i P_i^{-1} \tag{2.17}$$

Property	Value [unit]
Air density (p)	$1.205[kg/m^3]$
Thermal conductivity (k)	$0.0257[W/mK]$
Volumetric capacity (C_v)	$1210[J/m^3K]$

Table 2.2: Properties of air at 20 degrees Celsius

Property	Value [unit]
Air density (p)	$1.205[kg/m^3]$
Thermal conductivity (k)	$1[W/mK]$
Volumetric capacity (C_v)	$100[J/m^3K]$

Table 2.3: Model properties of air at 20 degrees Celsius

$$x = P_m \sum_{i=1}^{n} W_i P_i^{-1} x_i \tag{2.18}$$

2.3 Temperature Model

As described in the section 2.2.1, a model is required to describe the state changes between two consecutive time steps. This transition matrix is depended on the effects that can happen between these two time steps. In this thesis, the natural effect monitored is the air temperature of a closed room. Therefore, all relevant properties of this effect need to be expressed in the model. The model is derived from the model for diffusion developed at TNO [34].

In table 2.2, all relevant parameters of the model are shown. However, temperature is both a convection and a conduction process which results in problems when modeling temperature as a linear process: convection is a process that is very non-linear. To alleviate this, the convection part of the process will be modeled using conduction. The actual values that are used in the temperature model are therefore different than the actual values in nature. In table 2.3, the used values are shown.

In this thesis, a grid of 9 by 9 meters is used, with each grid point having a size of one meter (d_l) by one meter (d_w). It is assumed that no heat is lost in the z direction.

At each grid point q, the equation shown in 2.19 is the time continuous model of the temperature, were F^q is the new temperature of a cell, and a being the transferred heat from one cell to another. All cells have four neighbors, denoted as north, south, west and east.

$$\tilde{F}^q = aF^q + a_n F^{qn} + a_s F^{qs} + a_w F^{qw} + a_e F^{qe} \tag{2.19}$$

The temperature F^q is determined using the following equation

$$F = pC_v d_l d_w [kgJ/Km^4] \tag{2.20}$$

Given there is no wind, it is assumed all cells spread ($a_{[n,s,w,e]}$) $0.3kg/m^3$ of air to each neighboring cell. This spreading can be modeled using the following equation:

$$G = 0.3k\frac{d_l}{d_w}[kgJ/m^4Kt] \tag{2.21}$$

Filling in the parameters of air, the continuous model of temperature in a room can be written as seen in equation 2.22.

$$\begin{aligned}\tilde{F^q} = \frac{-1.2 \times 0.0257}{1.205}F^q + \frac{0.3 \times 0.0257}{1.205}F^{qn} + \frac{0.3 \times 0.0257}{1.205}F^{qs} \\ + \frac{0.3 \times 0.0257}{1.205}F^{qw} + \frac{0.3 \times 0.0257}{1.205}F^{qe}\end{aligned} \tag{2.22}$$

To be to use this continuous model in a Kalman Filter, the model needs to be converted to a discrete model. This last step results in the transition matrix A that will be used in the Kalman Filter.

$$A = (\frac{F}{t_s} - G)^{-1}\frac{F}{t_s}[unitless] \tag{2.23}$$

Chapter 3

Implementation

In this work, the main aim is to assess the feasibility, performance, reliability and functioning of different Kalman filter distribution methods on a wireless sensor network that communicates via gossiping. To test the feasibility of a distributed Kalman Filter with gossiping, an implementation needs to be developed and tested. In this chapter, an overview will be given of the architecture as used in the implementation of the distributed Kalman Filter.

3.1 TelosB Wireless Sensor Node

For this research TNO has provided TelosB nodes which will be the platform for the implementation. These nodes are created from an open design that may be implemented by everyone. These specific nodes, that are often used for wireless sensor network research, are created by MEMSIC. As any wireless sensor node, the node is composed out of several chips, together forming the node.

The main controller of the chips, the microcontroller, is a 16 bit TI MSP430 RISC microcontroller. This microcontroller will run the actual code. To store this code a memory of 48 KB is provided, while the variables can be stored in a RAM of 10 KB. [20]

Besides the microcontroller, a wireless sensor network needs to communicate. The communication platform on the TelosB is the CC2420 radio, an IEEE 802.15.4 compliant radio. This radio can send data with a speed of 250 kbps using the antenna integrated in the circuit. Sensors are provided for battery voltage, light, humidity and temperature [31]. The power required for the node is provided by two AA batteries that can be placed in a holder on top of the circuit, providing the energy required for functioning.

To provide the necessary tools and hardware abstraction, an operation system is required. The operation system of choice for this work is TinyOS, discussed further in the section 3.2. In Table 3.1, an overview is given of the TelosB node with the most important specifications.

Current drawn by processor in active mode	23 mA
Current drawn by processor in sleep mode	5.1 μA
Current drawn by radio in receive mode	23mA
Current drawn by radio in idle mode	21 μA
Current drawn by radio in transmit mode	17 mA (@ 0dBm)
Frequency band	2400 MHz to 2483.5 MHz
RF power	-24 dBm to 0 dBm
Outdoor Range	75 m to 100 m
Indoor Range	20 m to 30 m
Visible Light Sensor Range	320 nm to 730 nm
Temperature Sensor Range	-40° C to 123.8° C
Temperature Sensor Accuracy	± 0.5° C @ 25° C
Temperature Sensor Resolution	0.01° C
Size (mm)	65 x 31 x 6 (without battery pack)

Table 3.1: Overview of important specifications of the TelosB wireless sensor node.

3.2 TinyOS

As described in section 3.1, an operating system is required for ease of development and hardware abstraction. The operation system of choice is TinyOS, an operating system that supports several commercially available wireless sensor nodes, including the TelosB [36].

TinyOS, currently at version 2.1, is specifically designed for wireless sensor node applications. TinyOS is programmed in a special version of the programming language C, called nesC, providing additional language features like components, concurrency and events [11]. A typical nesC application is made out of several of these components wired together to form a single application. Every component in TinyOS can provide interfaces that other components can use. An interface defines a contract which commands and events need to be implemented. By setting up TinyOS using these interfaces, the code is independent from the actual implementation, required for the platform independence that TinyOS tries to achieve. For example, every platform has its own Timer implementation that implements the same Timer interface to cope with the different hardware. In the consuming component however, only a single interface has to be used.

The events provided by TinyOS can range from hardware interrupts to timers to callbacks from earlier command invocations: TinyOS used the so called split-phase operations, where long tasks are split up into a “start” signal and an event signaling when the task is done. This provides a non-blocking way of performing long tasks, keeping the system responsive at all times. An example of such an operation is shown in figure 3.1.

```
// start phase
send();

//completion phase
void sendDone(error_t err) {
  if (err == SUCCESS) {
    sendCount++;
  }
}
```

Figure 3.1: An example of the split-phase technique used in TinyOS

3.3 Architectural Overview

As discussed in the previous section, a typical TinyOS application is composed out of multiple components, together forming a single application. The Kalman Filter implementation from this work also follows this pattern: it is composed out of several system blocks and custom created blocks. The blocks specifically created for this work, are components to perform the Kalman filter, gossip, store relevant parameters in a log and transmit this log when required, matrix calculations and a fixed point library. In figure 3.2, a figure is shown depicting the blocks and their interconnections. In figure 3.3, a flow chart is given of the control logic through the application. The different colors in the two figures give an impression of which logic is performed by which block. In the next sections, the blocks from figure 3.2 and the steps from figure 3.3 are further discussed.

In figure 3.2, all (primary) components used are shown, with every arrow showing its connection to the other blocks, stating the implemented interface provided to the block that is pointed to. The depicted gray blocks are blocks provided by the TinyOS platform, while all other blocks are written for this thesis. For example, LedsC is the platform independent implementation of the LED controller; the ActiveMessageC component provides communication and RandomC is a pseudo random number generator.

As stated earlier, these blocks work together and are tied together by the main processing unit KalmanFilterNodeC to perform the needed work. In figure 3.3, for clarity, the logging function provided by HistoryC is not depicted. Furthermore, the described split-phase, event based model from TinyOS is modeled in figure 3.3 as a queue that is polled at set times. In reality, events are generated that initiate the flow.

From the flowchart in figure 3.3, it can be seen that the timer is the start of the whole process. When the timer goes off - it is set at the sampling frequency - the Kalman Filter process is started. The first step in this process is to read the current temperature from the sensors. This temperature

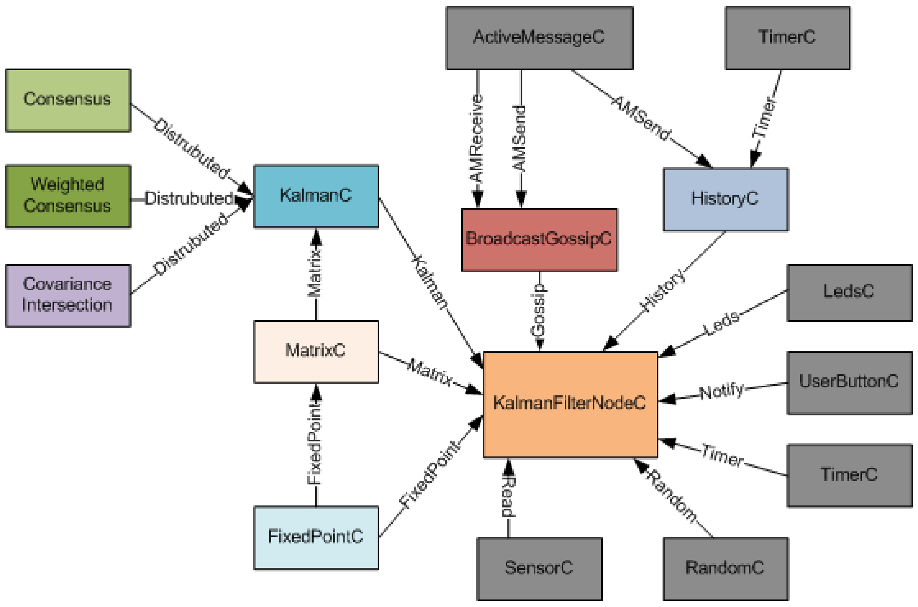


Figure 3.2: The components used in the implementation of the Kalman Filter

will be used later during the Kalman Filter Measurement update. Following this reading of the sensor, the Kalman filter performs its prediction step, as described in section 2.2. In this step, the Kalman filter will predict the state according to an estimated mean and a covariance matrix. In the next step, this prediction is fed into the measurement update, that with the sampled temperature will result in a new estimate vector with accompanying covariance matrix. The final step in this process, is the distribution step. In this step, information about the local are spread out to its neighbors, depending on the gossip and distribution algorithm.

At times that the timer did not signal, the node is in its receive state. In this state, messages from its neighbors can be processed. This processing is again dependant on the distribution algorithm, performing different processing for different algorithms.

3.4 Platform Specific Considerations

The current platform of choice for this thesis is the TelosB wireless sensor node. However, at TNO interest has been shown for other wireless sensor platforms, like the TMote platform [22]. This platform is comparable with the TelosB platform, but differs in its radio, microprocessor and other components.

Since TinyOS has build in support for platform independent development via components and interface, it has been decided to write a fixed point

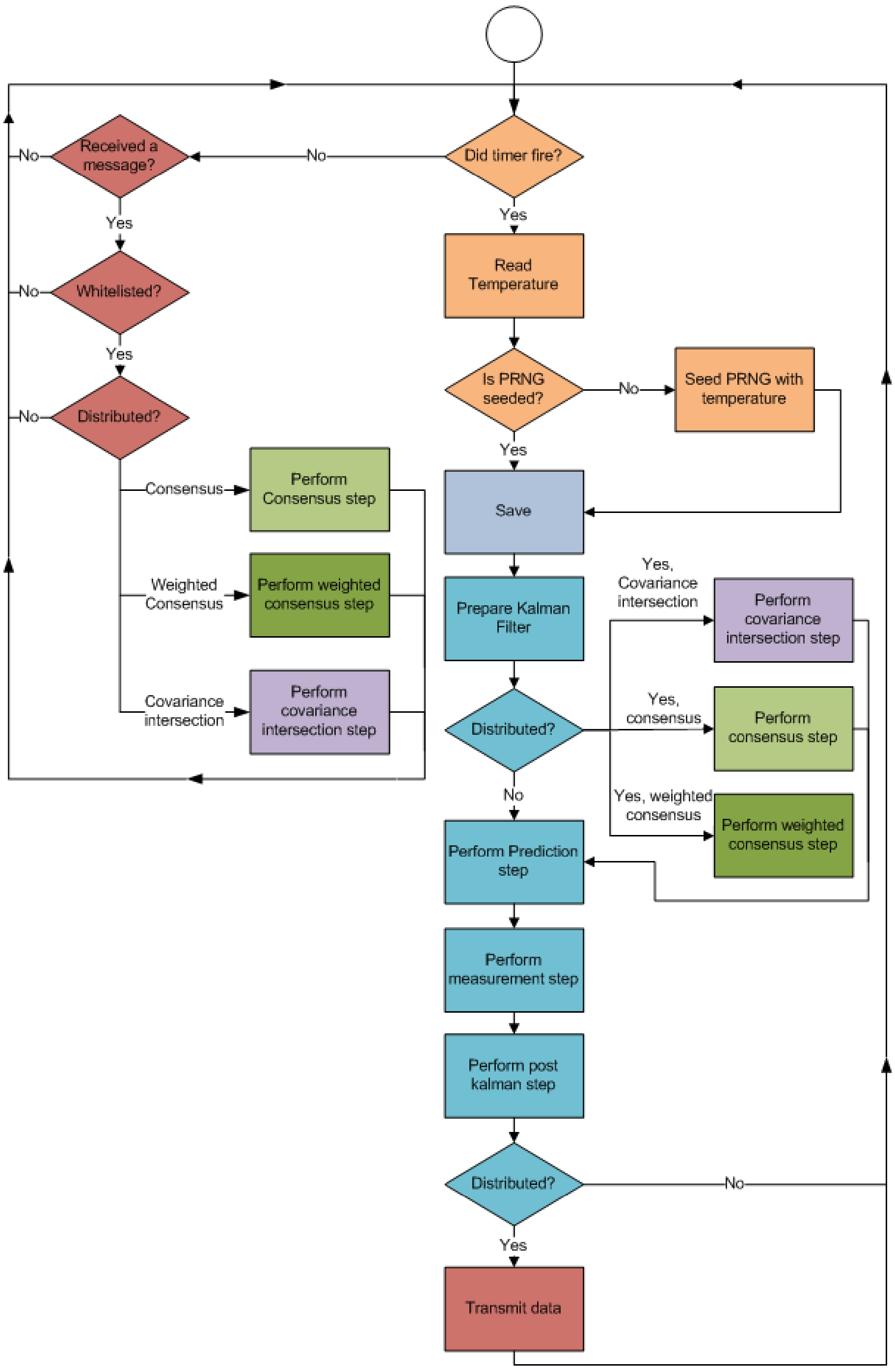


Figure 3.3: Flowchart depicting the flow through the distributed Kalman algorithm

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1

Table 3.2: The number 35 as represented in integer arithmetic

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1

Table 3.3: The same binary representation as table 3.2, using the Q10.6 fixed point notation

component and a matrix calculation component in nesC. The current implementation is platform independent: it does not use any microcontroller specific code. However, future implementation that do use microcontroller specific code can be plugged into the module, without any need for refactoring. The matrix component uses this fixed point component for fixed point matrix calculations.

In the following sections, these two components will be described in further detail.

3.4.1 Fixed Point Component

The TelosB platform has, just like most microcontrollers, no support for floating point arithmetic, i.e. support real numbers. While this can be simulated in a software environment, this is generally bad for performance. As a solution to this, fixed point arithmetic can be used. In this method, calculations are performed using native integer arithmetic, directly in the hardware. The real part of the number is stored starting at a certain bit, as explained further on. By default, TinyOS has no component or library for fixed point calculation. Therefore, a custom component that can be used throughout the application has been developed for this thesis work.

In fixed point arithmetic, the weight of every bit is different than in classic integer arithmetic. In the table 3.2, a typical 16 bit integer is shown, representing the number 35. To retrieve the number represented in this diagram, the weights of all bits that are one need to be added. This results in the following equation: $2^0 + 2^1 + 2^5 = 35$.

In this work, two fixed point formats will be used, one of which is Q10.6. In this format, 6 bits will be used to represent the fractional part of the number, while 10 bits will be used for the integer part and sign. This is achieved by shifting the 2^0 bit to the left with 6 positions. If the same binary representation as the Table 3.2 is used in this Q10.6 format, the table as shown in Table 3.3 is formed.

Again, just as in the previous diagram, the number can be retrieved by adding all weights for bits that are one: $2^{-1} + 2^{-5} + 2^{-6} = 0.546875$.

A big advantage of using this method, is that the simple arithmetic functions like adding and subtracting, can be performed using integer arithmetic that is available in the hardware and thus providing the expected speed and efficiency. Multiplication and division however does require some extra steps in the process to get results in the same Q10.6 format as the input. An example for multiplication is given in equation 3.1

$$x = (a \times b) \gg pointlocation \tag{3.1}$$

Since a part of the fixed point number is used for the fractal part, the range is limited. For example, the maximal value a number can have in Q10.6 is 15 one bits in a row, representing the number 511.9844. Likewise, the smallest precision that can be reached is the value of a single one in the lowest position, representing 2^{-6}, or 0.015625. Any number that required more precision to function, cannot use this representation.

As described earlier, this work uses two fixed point representations. The first one is Q10.6 as discussed in the previous section. The second representation is Q21.11, with both more space for the integral part and for the fractal part. This representation will be used for all calculation in the matrix component described in the next section, since it can provide more precision as well as less chance of overflow: the largest possible number that can be expressed in this format is 1048575.9995, with a precision of 0.00048828125. As can be seen from these numbers, using this format will result in results with outcomes closer to the outcome when using real numbers.

In the fixed point component, a method is provided that converts between the two formats, losing precision when going to the smaller format. Values that will result in overflow are clamped to the maximal and minimal value automatically.

3.4.2 Matrix Component

The Kalman filter uses matrix calculations for most of its computations. Therefore, a library specifically designed for this work is required. For this, a TinyOS component is written to perform operations on matrices. By setting it up as a TinyOS component, future implementation can be plugged in without any changes to the rest of the code. The module is setup in such a way that it uses the fixed point component, making it possible to calculate matrices with fractals.

The implemented Matrix library is able to perform matrix comparisons, transposition, addition, subtraction, multiplication with other matrices, vectors or scalars, matrix inversion and calculate the trace. The most stressing matrix operations are multiplication and inversion, both often used in the Kalman filter. In the current implementation, multiplication has the complexity of $\mathcal{O}(N^3)$. Inversion is performed via Gauss-Jordan elimination, an operation of complexity $\mathcal{O}(N^3)$ as well.

Because the matrix library uses the fixed point component for its calculations, the results of intensive calculations, for example a matrix inversion, are not the values that will be obtained when using floating point arithmetic. This is caused by the Q21.11 representation that regardless of its precision in the order of 5×10^{-4}, is less precise than floating point numbers. To see this offset from the actual values, an experiment is setup with the matrix from equation 3.2.

$$x = \begin{pmatrix} 3 & 5 & -1 & -4 \\ 1 & 4 & -0.7 & -3 \\ 0 & -2 & 0 & 1 \\ -2 & 6 & 0 & 0.3 \end{pmatrix} \tag{3.2}$$

When the matrix from equation 3.2 is inverted in *floating point*, the inverted matrix is as shown in equation 3.3.

$$x^{-1} = \begin{pmatrix} 0.6546 & -0.9355 & -0.1930 & 0.0142 \\ 0.1979 & -0.2829 & -0.1036 & 0.1558 \\ 0.3683 & -1.9565 & -4.2643 & -0.4230 \\ 0.3962 & -0.5662 & 0.7924 & 0.3112 \end{pmatrix} \tag{3.3}$$

In the *fixed point* implementation however, this inverse will be calculated as the equation 3.4.

$$x^{-1} = \begin{pmatrix} 0.6544 & -0.9348 & -0.1912 & 0.0142 \\ 0.1983 & -0.2832 & -0.1034 & 0.1558 \\ 0.3683 & -1.9547 & -4.2635 & -0.4249 \\ 0.3966 & -0.5666 & 0.7932 & 0.3116 \end{pmatrix} \tag{3.4}$$

When comparing these two results, the accuracy can be measured to be at two digits. However, every step in the calculation will continue to add to this error, resulting in a bigger error at the end of the calculation step. To show this, the inverted matrix in equation 3.4 is inverted again, resulting in the original matrix from equation 3.2. However, because of the *fixed point* calculus, this matrix will only be approximated. The error of this approximation, on average, is 0.1. As a result of this error, the Kalman filter implementation can only approach a *floating point* implementation.

As previously stated, both multiplication and inversion are operations with a complexity of $\mathcal{O}(N^3)$. As a result of this, the operation can take a lot of time to calculate the results. To test this, a timing experiment is performed using representatives 9×9 matrices. The size of these matrices is based on the experiments that are performed in chapter 4. The results of this timing experiment is shown in table 3.4.

Inversion [ms]	0.30
Multiplication [ms]	0.25

Table 3.4: Time taken for matrix calculations to be completed

VDD [V]	d_1 [C]	SO_t [bit]	d_2 [C]
3	-39.6	14	0.01

Table 3.5: Conversion for the sensor data to actual temperatures.

3.5 Temperature Measurement

In section 3.3, the architectural overview of the system is given. From figure 3.3, it follows that, once the timer has gone off, the first step in the Kalman filtering processes as implemented is the collection of the sensor data. This step is performed using the TinyOS split-phase execution model. In the first step, a request is made to the operating system to get the temperature from the onboard chip. Because of the interface / component setup from TinyOS, this call is platform independent. When the OS has received the temperature from the onboard sensor chip, the OS raises an event. In this event, a sensor dependant value is supplied, leaving further processing to the application.

The TelosB platform at TNO uses the Sensirion SHT1x humidity and ambient temperature sensor for temperature measurements [31]. The value returned by this sensor needs to be converted to degrees Celsius before any other processing can happen. For the SHT1x, this conversion is a linear equation dependent on the system voltage and the number of bits that are read from the sensor. The equation that needs to be executed is shown in equation 3.5.

$$T = d_1 + d_2 \times SO_t \tag{3.5}$$

In this equation, SO_t is the digital temperature readout provided by the sensor. Values d_1, d_2 are the voltage and data width constants respectively. For the TelosB, the data width is set to be 14. In table 3.5, the conversion between voltage, data width, d_1 and d_2 is given.

The equation resulting in degrees Celsius therefore is given in equation 3.6.

$$T = -39.6 + 0.01 \times SO_t] \tag{3.6}$$

For some parts of the implementation, a random number generator is required. In TinyOS, the default Pseudo Random Number Generator uses the ID of the node for its seed. Because the same random numbers would be used during every run, this is suboptimal. By using the first measured temperature alongside the ID of the node to seed the PRNG, this problem is averted.

3.6 Data Processing

In the data processing step, an estimation of the temperatures in the grid is performed using the measured temperature is. When the software is configured in as a distributed Kalman filter, the information from its neighbors will also be processed in this step. In this section, it will be assumed all information has been gathered into the local state.

In this thesis, every single wireless sensor node will try and estimate the temperature of all grid cells in the network. This means that the Kalman Filter will need to keep track of N states, where N is the number of grid cells. Therefore, the resulting estimation vector has N elements. As discussed in section 2.2.1, the covariance matrix keeps track of the error in the estimations. With N states, this matrix will be $N \times N$ in size. Therefore, the bottleneck of the distributed Kalman filter is the covariance matrix. Because of the quadratic size increase, the size of the network has been determined to be a grid of 3×3, resulting in 9 different states. Along with the $\mathcal{O}(N^3)$ multiplication and inversion operations, this was deemed stressing enough to evaluate the performance of the wireless sensor nodes.

For the Kalman filter implementation, two forms of equations have been assessed. In *form 1*, as seen in equation 3.7, the gain K needs to be computed before the measurement update can happen. In *form 2*, as seen in equation 3.8, this gain is not calculated.

$$x_p = Ax \tag{3.7a}$$
$$P_p = APA' + Q \tag{3.7b}$$
$$K = P_pC(CP_pC' + R_s^2)^{-1} \tag{3.7c}$$
$$P = (I - KC)P_p \tag{3.7d}$$
$$x = x_p + K(y - Cx_p) \tag{3.7e}$$

$$x_p = Ax \tag{3.8a}$$
$$P_i = (APA' + Q)^{-1} \tag{3.8b}$$
$$P = (P_i + C'(R_s^2)^{-1}C)^{-1} \tag{3.8c}$$
$$x = P(P_ix_p + C'(R_s^2)^{-1}y) \tag{3.8d}$$

In these equations, A is the transition matrix, Q the process noise, C the observation vector describing the local grid cell, R_s the sensor noise and y the measurement value.

Although *form 2* uses less multiplications (6 in *form 2* versus 11 in *form 1*), the two inverse calculations make *form 2* more expensive than *form 1*. Since C is a vector of size 9, in *form 1* the inversion becomes an inversion of a 1×1 matrix which can be implemented as a single division. The two inversions in *form 2* however, are both a inversion of a 9×9 matrix. As discussed in section 3.4.2, inversion is a costly operation requiring a temporary matrix of 9×18 for the Gauss Jordan algorithm to work. *Form 2* can be more efficient, but only in cases where the state is small with relation to the measurement. As this is not the case in this work, the Kalman filter implementation will use *form 1*.

3.7 Data Storage

To be able to measure the performance of the Kalman filter, some way of extracting run time information from the algorithm is required. At every time step, the required information for performance analysis needs to be persisted.

The information that is meaningful for this evaluation are:

1. The temperature as measured directly by the sensor on the node. This value will be used to evaluate the performance of the Kalman filter;
2. The neighbors that have contributed to the current Kalman filter state in this round;
3. The estimation vector containing all temperature estimates for the current round.

Implementing a way of persisting the data can be done is several ways, of which two are considered.

1. Sending information during the test.
   By sending the information, the base station is able to make real time measurements of the test in progress. The downside of this approach is that the transmitted data can cause interference with the normal test, causing problems like missed messages. Furthermore, the base station needs to be within range of all nodes performing the test.
2. Storing information during the test.
   By storing all interesting information during the test on the node until it is retrieved does not have the same problems as live transmission has. All data is send after the test, causing no additional collisions. Range is also not a problem, since the nodes can be moved within the range of the base station for data extraction. The downside of storing the information is the limited storage space available on the nodes.

To store the wanted information, 22 bytes per log item needs to be allocated. In practice, it means that all remaining storage space will be used by the log. An experimental maximum of 200 log entries was determined, taking almost 4.5 kB of storage space. At a sample interval of 3 seconds, it is only possible to store 10 minutes of information with these 200 entries.

Accessing these two methods, the second option has been selected for its low impact on the running experiment, while it is still able to reliably store all relevant parameters of the experiment. In the implementation, the data log can be retrieved by pressing the “User” button on the TelosB hardware, stopping the Kalman Filtering and entering the data extraction mode. In this mode, all logs are send out one by one at a rate of 10 per second to the base station running a application persisting the log.

3.8 Kalman Distribution

In this work, the wireless sensor node need to cooperate in order for them to estimate the temperature distribution throughout the network. In section 2.2.2, some approaches are discussed that can achieve this goal. Some of these models will be implemented and evaluated in both simulations and experiments. Every implemented distribution model is implemented as a TinyOS component with the same interface. This way, different distribution models can be plugged in at compile time, resulting in an application with different behavior. The implemented models will be discussed in the following sections.

3.8.1 Consensus

The first two distribution models that are implemented are based on consensus as discussed in section 2.2.2. As explained in that section, wireless sensor nodes exchange their estimate vectors with each other. The received vectors are then merged into a single new local estimate vector. The approach at which this merging is performed, can differ. In this work, two different merging algorithms are implemented.

The first merging algorithm for consensus, is a simple **averaging scheme over all the received estimation vectors** from its neighbors. If a node has received three messages from its neighbors, every message - and its local state - will each weigh in for 25 percent (equation 3.9a and 3.9b). This averaging will result in a single new local state, that will be used in the next Kalman Filter round.

$$x_1 = 0.25x_1 + 0.25recv_1^1 + 0.25recv_1^2 + 0.25recv_1^3 \tag{3.9a}$$

$$x_2 = 0.25x_2 + 0.25recv_2^1 + 0.25recv_2^2 + 0.25recv_2^3 \tag{3.9b}$$

50%	12%	5%
12%	12%	5%
5%	5%	5%

Table 3.6: Weights for the estimation vector per sender

The second implemented merging algorithm for consensus is, like the previous implementation, an averaging scheme. However, in this implementation, the weighing is not in direct relation with the number of received messages. Here, **weighing per element of the vector** is performed instead of the entire vector.

The merging algorithm is based on the assumption that a node does not accurately know all the temperatures in the network. However, it is assumed that it does have knowledge about its local temperature and the temperatures of its direct neighbors. The assumption of its knowledge about its local temperature is a straight forward one: it measures this information locally. The assumptions about its neighbors is based on the fact that temperature is a process that is not very local: neighboring grid cells usually have the same temperature profile. Furthermore, a node has more chance of receiving messages from its direct neighbors.

Implementation wise, this merging algorithm is implemented using weights on a per element basis. The used weights in this algorithm are dependent on the sending node. Every node has a local table stating the direct neighbors of the sender, the two hop neighbors and all other neighbors. In table 3.6, a table is shown that is used when a message is received from node 1: this node has high chance of knowing the temperature in grid cell 1, so it is assigned a weight of 50 percent (equation 3.10a). The direct neighbors of node 1, grid cells 2 (equation 3.10b), 4 and 5 are weighted less since it is only an estimate. The rest of the elements are weighted with 5 percent.

$$x_1 = 0.5x_1 + 0.5recv_1 \tag{3.10a}$$

$$x_2 = 0.88x_2 + 0.12recv_2 \tag{3.10b}$$

3.8.2 Covariance Intersection

Besides the estimation exchange implementation as discussed in the previous section, a different Kalman distribution method has been implemented. This method is called covariance intersection and follows the principles as discussed in section 2.2.2. As explained, covariance intersection sends its covariance matrix along with the estimation vector, merging them both into its local state. Since the covariance matrix holds information about the accuracy of the estimation, it can be seen as an extension of the consensus algorithm with sender based weighting. However, the information about the correctness is not assumed, but determined during run time. Covariance intersection also has its drawbacks. Where in the consensus the information is build into the software, in covariance intersection the information needs to be transmitted. In section 3.9.1, methods will be discussed to reduce the computational and communication costs of covariance intersection.

3.9 Distribution Dependent Considerations

With the different models as discussed in the previous section, some considerations need to be made for the implementation. For the other discussed software components, the implementations are the same, independent of the Kalman distribution. In this section, the parts that differ dependant on the distribution method, will be discussed.

3.9.1 Data Transmission

In **consensus**, the data transmission is straightforward. As discussed in section 2.2.2, the only information that needs to be shared between nodes is the estimation vector. Internally, this vector is made out of nine fixed point numbers with the Q21.11 representation. When transmitting this, it will result in 36 bytes of data that needs to be transferred. Because in wireless sensor networks transmitting and receiving data is the most expensive operation, it can be beneficial to reduce the number of bytes that need to be transferred. To reduce the number of bytes, the estimation vector is encoded using the Q10.6 fixed point notation, using two bytes, reducing the data size with 50 percent.

For **covariance intersection**, some extra information needs to be exchanged for the algorithm to work. The entire covariance matrix needs to be transmitted as it is needed for correct information fusion. Luckily, the matrix P is a symmetrical matrix, meaning that $P = P^T$. By only transmitting the unique information of the upper half of the matrix, 45 elements instead of 81 in case of a 9×9 matrix, the amount of communication can be reduced. By applying the same encoding of the elements as in the consensus algorithm, two bytes per element need to be transmitted resulting in 90 bytes.

For optimizations reasons however, the covariance matrix is not transmitted as is. From equations 2.16 to 2.18 in section 2.2.2 it follows that only the trace of the covariance and the inverted covariance matrix is used. To minimize the total amount of inversions in the network, the covariance matrix is transmitted in its inverted form, requiring only a single inversion per transmitted message instead of an inversion per received message. As stated, the trace is also required by the algorithm; this value is transmitted along with the inverted matrix as a trade off between computational power and communication power.

3.9.2 Data Reception

In **consensus**, two additional vectors are used to store the incoming messages. Depending on the consensus merging algorithm, the incoming vector is added without any further processing, or first multiplied with the merging weight. The second vector is used to store the percentages of the weighted consensus, keeping track of the remaining weighing percentage that will be applied to the local state.

In **covariance intersection**, every received message is stored in memory until the communication has finished. This is done because the covariance intersection algorithm is only able to fuse all states into a single state when all external states are known.

Chapter 4

Experimental Results

To test the implementation as discussed in chapter 3, several experiments have been conducted with the implementation of the distributed Kalman Filter that was designed and implemented within this research project.

In section 2.2, three different methods of distributing and merging the Kalman state is described. Each of these methods has been tested in different network layouts to be able to see the effect of the network layout on the used distribution method. After all experiments, it is possible to make informed decisions about which distribution method is best in certain network layouts.

In the next sections, the experimental setup and the evaluation metrics will be discussed followed by experimental results.

4.1 Experiment Case

For the experiments performed for this work, nine wireless sensor nodes were deployed to measure the temperature in a grid of three by three with cell sizes of one meter. A heat source is placed on one side, blowing hot air into the grid. The grid is depicted in figure 4.1.

With this setup, a recording is made of the temperature at every grid point. These recorded measurements are played pack during every experiment allowing for comparison between different experiments as the input data is identical. The sample rate of the recording and the sample rate of the distributed Kalman filter are kept the same.

The measured temperatures are depicted in figure 4.2. The temperature behavior during the experiment is as follows: from $t = 0$ to $t = 220$ the temperature is stable. From $t = 220$, the heat source is enabled, blowing in hot air until $t = 390$. From then, the temperature will slowly stabilize again to its former state, falling back to the temperature when the experiment is started. From this, it is logical that all temperatures seem to converge to a single value, as the temperature in the grid returns to room temperature.

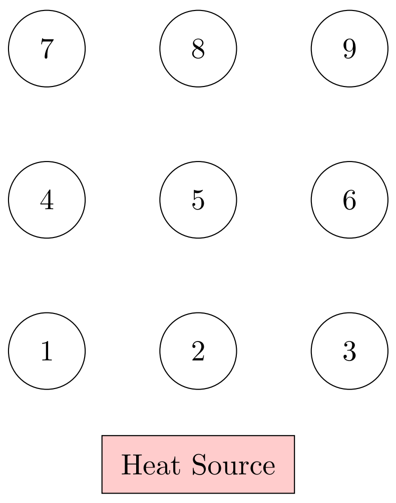


Figure 4.1: Overview of the grid as used in the experiments.

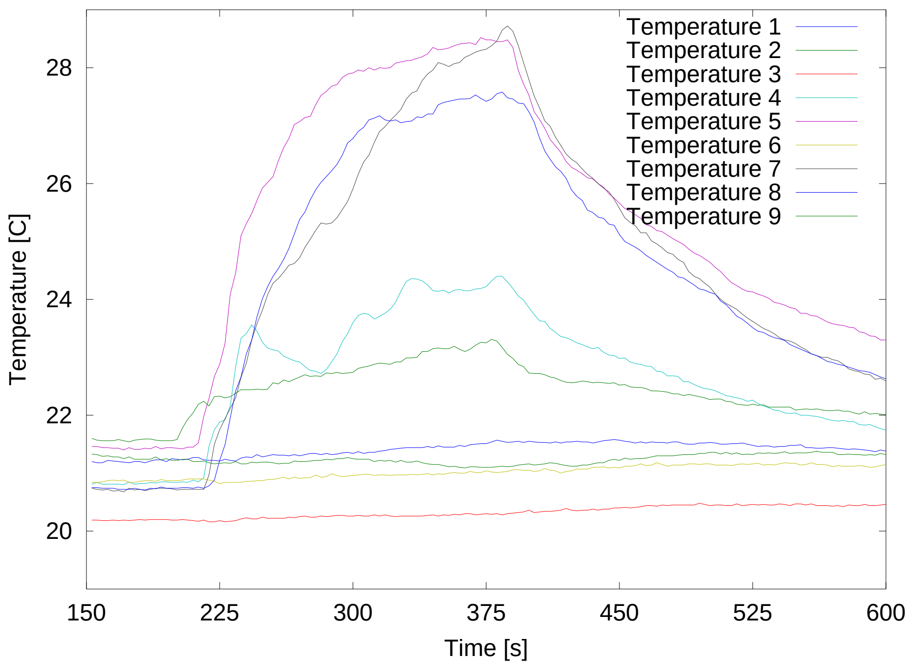
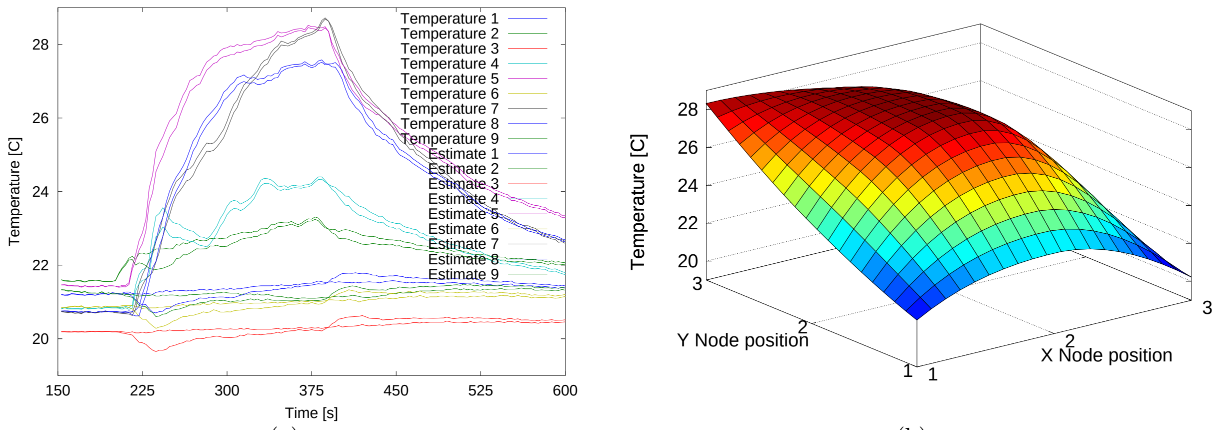


Figure 4.2: Temperatures measured in the grid during the experiments



(a) (b)

Figure 4.3: (a): Temperature changes and estimate results of the central Kalman filter from $t = 150$ to $t = 600$. (b): Estimate results of all grid cells by the central Kalman filter at $t = 375$

Figure 4.2 indicates that nodes 5, 7 and 8 are influenced the most by the heat source, rising five degrees Celsius above room temperature. Node 4 and 2 are only affected to a smaller degree. Nodes 1, 3, 6 and 9 are totally unaffected by the temperature changes created by the heat source. These differences in the way the nodes are affected by the temperature changes are important, as they make it possible to clearly see the differences in the different distribution methods that will be used.

In the following sections, the results from the different implementation will be discussed, along with metrics that describe the effectiveness and correctness of the implementation and allow for comparison between the different implementations.

4.2 Central Kalman Filter

For a good comparison between the different distribution methods, a base line is required for the algorithm used. As a base line for the distributed Kalman filter, a central Kalman filter will be used.

In figure 4.3(a), the estimate for every grid point is shown, alongside the temperature that has been recorded as discussed in section 4.1. From this figure, it can be seen that the temperature model as described in section 2.3 is able to estimate the temperatures in all grid cells accurately: rising with the temperature as the heat source is turned on at $t = 220$ and slowly falling again when the heat source is turned off at $t = 390$.

In figure 4.3(b), a slice of the experiment is shown of the estimation at $t =$

375. This is an interesting point of time in the experiment, as it is a shifting point from rising to falling temperatures. For all coming experiments, this time slice will be used to show the capabilities of the distributed Kalman filter. More precise metrics will be defined in the next section to quantify the results from the different distribution methods along with the different network layouts that are tested.

4.3 Evaluation Metrics

To evaluate the performance of the different implementations, some metrics need to be defined to compare the three implementations with each other in an objective manner. The method that performs best according to these metrics, will be selected as the best implementation for the given environment of temperature measurement. The metrics shown are calculated on a single time during the experiment: $t = 375$.

Since the goal of this thesis is to see which distribution method is best suited for different network layouts, the most important metric is the correctness of the estimate with relation to the temperature estimated by a central Kalman filter implementation. As such, the first metric is the RMS value of the estimation error of every grid point per node. For a single node this metric is calculated using the equation as stated in 4.1. In this equation, x is the estimate result of that node, c the estimate result of the central Kalman filter and n the number of nodes, which is 9 in this experiment.

$$RMS = \sqrt{\frac{(\sum_{i=1}^{n} x_i - c_i)^2}{n}} \tag{4.1}$$

A second metric that is important, is the degree of agreement between the different nodes in the network: if only one node is able to estimate the correct temperatures profile in the entire grid, with all other nodes having a different notion of the temperatures, this is deemed incorrect. In the best case scenario, all nodes are able to make the same estimate for every single grid cell. To test this metric, the RMS is taken per grid cell of the error between the estimate of a node and the average estimate of all the nodes. This metric for a single node is defined in equation 4.2.

$$RMS = \sqrt{(\frac{1}{n}\sum_{i=1}^{n}(x_i - \frac{1}{n}\sum_{j=1}^{n} x_j))^2} \tag{4.2}$$

If an algorithm is able to perform well on the metrics *Agreement*and *Correctness*, it means that all nodes in the network are able to accurately estimate the temperature in the grid with an acceptable error.

Another metric that is important, is the time taken for the algorithm to perform. Since calculation time is in direct proportions to the power consumption, using less time to process is better for the lifetime of the battery. This metric is recorded on the hardware using a timer with a resolution of 32 KHz, or 312.5 μs; making this the resolution of this metric. The definition of the metric *Time*is given in 4.3.

$$\text{Time required for received messages} \tag{4.3}$$

Besides calculation time, data transmission is also a big power consumer. It is required to know how much data is transferred. This way, a selection can be made on the power consumption of an algorithm. Since all three algorithms spread their information at the same rate, the only difference is in the amount of data transferred. This therefore the metric *Memory*, defined in 4.4.

$$\text{Number of bytes transferred per gossip round} \tag{4.4}$$

The third metric concerned with implementation details is metric *Storage*. In this metric, the amount of memory is counted. Since in a wireless sensor node memory is sparse, a more memory efficient algorithm can be important. The definition of metric *Storage*is defined in 4.5.

$$\text{Number of storage bytes required} \tag{4.5}$$

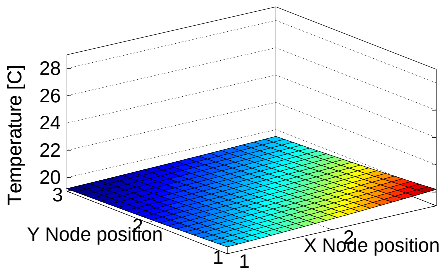
4.4 Local Kalman Filter

The first implementation that is used in an experiment is an implementation of a Kalman Filter without any form of communication between the different nodes in the network. As an effect of this, all nine nodes will have only information about the temperature in its local grid cell, therefore not able to estimate the temperature in the other grid cells precisely. The main purpose of this section therefore is to evaluate whether the implementation is done correctly.

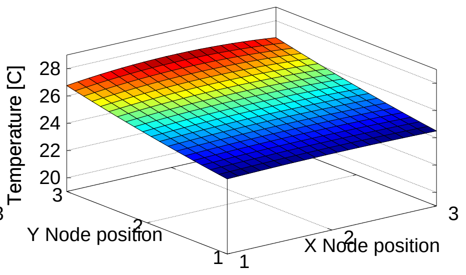
In figures 4.4(a) and 4.4(b), the estimation results of node 3 and 8 are shown of this experiment. From these graphs it can be seen that all nodes independently try to estimate the temperature of the grid. Since there is no communication, the nodes are not able to get a reliable view of the temperature distribution. The only estimate that is correct, is the estimate at its local grid point, with all other grid cells following the temperature of the local grid point.

Metric *Agreement*is not relevant for the local Kalman filter implementation, because of the lack of communication. This metric is therefor not shown.

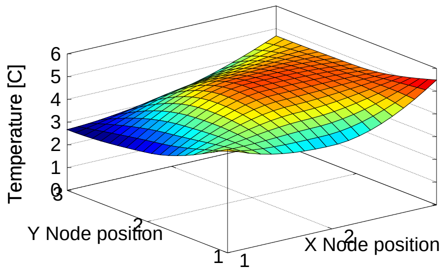
Figure 4.4(c) (metric) shows the offset of the experiment with the central Kalman filter implementation of section 4.2. Every single node in the



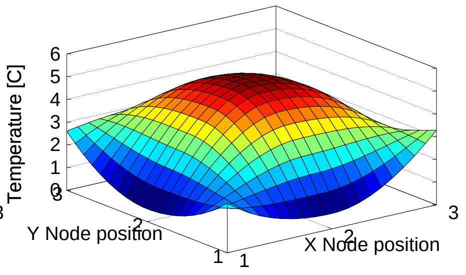
(a)



(b)



(c)



(d)

Figure 4.4: Local Kalman filter results. (a), (b): Estimate results of node 3 and 8 respectively. (c): Offset to central Kalman filter. (d): Metric *Agreement*.

Metric *Correctness*	Minimum [^{o}C]	2.67
	Maximum [^{o}C]	5.49
	Mean [^{o}C]	4.11
Metric *Agreement*	Minimum [^{o}C]	0.56
	Maximum [^{o}C]	4.98
	Mean [^{o}C]	2.44

Table 4.1: Results from local Kalman filter implementation

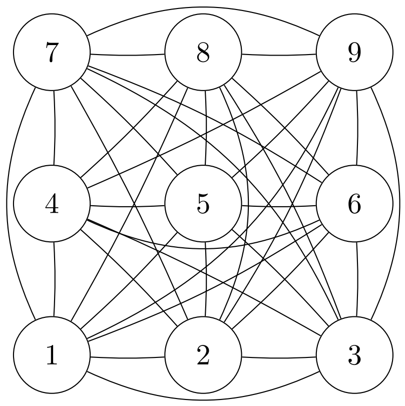


Figure 4.5: The fully connected network as used in the experiments.

network has a large error, ranging from 2.5 degrees Celsius in node 7 to 5.5 in node 3.

Metric *Memory*and *Time*can be handled quickly for the local implementation. Since there is no communication, all time taken in processing is purely in the Kalman Filter, $0.38\mu s$. The amount of data transferred therefore is 0 bytes. Metric *Correctness*and metric *Agreement*are summarized in table 4.1.

The goal of this thesis is to have a working decentralized Kalman Filter, so in the next sections, communication and data merging will be added.

4.5 Fully Connected Network

In the first real experiment with communication between nodes, the nodes are again spread as described in section 4.1. Since the nodes are in relatively close range to each other and transmission power is at its maximum, all nodes can communicate with all other nodes, forming a so called fully connected network, were every node can reach every other node. This topology is depicted in figure 4.5.

In the following sections, experiments with the three different Kalman distribution algorithms as discussed in section 3.8 will be performed and evaluated.

4.5.1 Consensus

The first distributed Kalman filter algorithm that will be evaluated, is the consensus algorithm as discussed in section 3.8.1.

From the graph in 4.6(a) showing the estimation results made by node 3 of all the grid cells, it can be seen that consensus is able to make an estimate of the temperature in other grid cells, whereas the local Kalman filter from section 4.4 is not able to do this.

In figure 4.6(b) the metric *Correctness*is shown. It can be seen that the temperature estimate has an error with standard deviation of 2.1 degrees for all grid points in the network. Statistically, this means, that 68% of the nodes at this time have an estimate of the temperature within an error of 2.1 degrees. This equal error can be attributed to the consensus step, as it tries to spread all information to the entire network, independent of whether this spreading is actually improving the estimate.

However, it is also important for nodes to agree on the estimates between each other. From metric *Agreement*in figure 4.6(c), it shows that the nine nodes have a different value for the temperatures in the grid cells. This is especially visible in node 5, that has a deviation from the average estimate of 0.8 degrees. This high error can be attributed to the fact that node 5 is the node with the largest temperature increase. All neighbours of this node sent information with lower estimates, influencing the local estimation process.

The last three metrics that are defined in 4.3 are *Memory*, *Time*and *Storage*. These three metrics are shown in table 4.2.

The differences in agreement and loss in precision stems from a weakness in the consensus algorithm. Consensus is only able to get values to converge when the measured state is measured by all nodes. In this experiment, the nine nodes all measure a different state. To improve the performance of the consensus algorithm, the averaging algorithm is changed to use a weighted during the averaging step. The results of this are shown in the next section.

4.5.2 Weighted Consensus

In the previous consensus algorithm, every incoming message is treated as equal. In this section however a weighted consensus algorithm is used with a different approach. As described in 3.8.1, elements of the estimate vector x are given a weight that is based on the sender of the message. This weight expresses the general idea of how good the sender can estimate that element. A result of this is, that estimations that are most likely correct will be weighted more than others, giving results closer to the actual temperature.

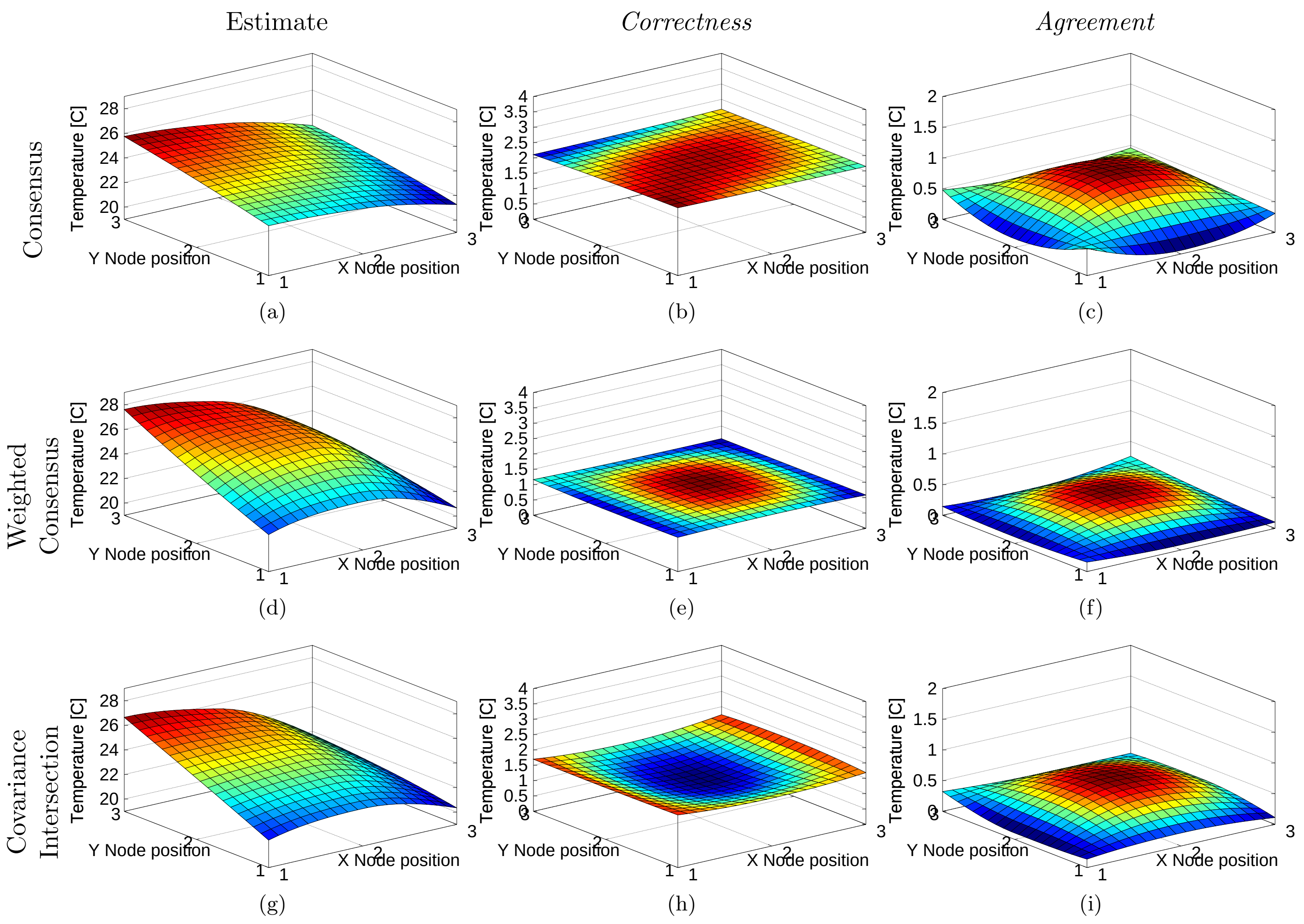


Figure 4.6: The estimate results, metric *Correctness*and metric *Agreement*for consensus, weighted consensus and covariance intersection for the fully connected network

In 4.6(d) the estimates of node 3 of all the grid cells at $t = 375$ are shown. The profile from this estimator is radically different than that from the non-weighted consensus from section 4.5.1. The nodes that do not experience any effects from the heat source are clearly visible in the plot.

The difference to the central Kalman filter, *Correctness*is given in 4.6(e). Where as in the non-weighted consensus the error of the estimate is 2.1 degrees, the weighted consensus is able to reduce this error to 1.2 degrees. This reduction is the result of the weighting in this algorithm. By applying the weights, the nodes can filter out members that do not help in the estimation. By apply this weighing, the error of the estimate has been reduced with 42%.

Metric *Agreement*in figure 4.6(f), shows that the differences between the estimates have gone down. The average error has been reduced to 0.17. The estimation results of node 5 also have improved, as its error has been reduced to 0.47.

The last metrics, metric *Memory*, *Time*and *Storage*are presented in table 4.2.

4.5.3 Covariance Intersection

The third algorithm that is evaluated for the fully connected network layout, is covariance intersection. As described in 3.8.1, in covariance intersection the covariance matrix is exchanged along with the estimated mean, potentially giving better results.

From figure 4.6(g), it can be seen that node 3 is able to estimate the temperature for other grid cells well. Just like the weighted consensus from the previous section, the nodes that are unaffected by the heat source can be clearly seen.

In figure 4.6(h), metric *Correctness*is shown for $t = 375$. The average error of all nodes is 1.6 degrees, a result that is between the error of the consensus and the weighted consensus algorithms. However, the metric *Agreement*in figure 4.6(i) shows that covariance intersection is able to outperform the naive consensus algorithm: it has an average error of 0.25. The weighted consensus however performs better on this metric. One interesting aspect of the covariance intersection is that node 5 is only in this experiment the node with the smallest error in *Correctness*. It is however not able to influence the other nodes enough, hence the peak on node 5 on the *Agreement*.

Metric *Time*, *Memory*and *Storage*are shown in table 4.2. Covariance intersection requires more calculation time and bytes transfers for its results than the consensus algorithm that only exchanges its estimates vector.

Metric		Consensus	Weighted Consensus	Covariance Intersection
	Minimum [^oC]	2.10	1.08	1.19
Correctness	Maximum [^oC]	2.20	1.25	1.81
	Mean [^oC]	2.16	1.12	1.62
	Minimum [^oC]	0.06	0.08	0.10
Agreement	Maximum [^oC]	0.85	0.47	0.61
	Mean [^oC]	0.39	0.17	0.25
Memory		18	18	112
Time		2.37	2.40	385.62
Storage	ROM [b]	916	1300	1726
	RAM [b]	124	498	1672

Table 4.2: Summary of all metrics of the fully connected network experiments. Note that the represented values are error values.

4.5.4 Summary

From the experiments with a fully connected network, it can be concluded that it is possible to get a reliable view of a grid in a wireless sensor network. All values from the previous sections, are repeated in table 4.2.

While the three different methods show different kind of results, all algorithms in this experiment are within 2.3 degrees from the actual temperature. Also, the three algorithms agree on their estimates with an error of 1 degree.

From table 4.2 however, it shows that the weighted consensus filter has the best performance on all metrics. Of course, as stated in section 3.8, the weighted consensus algorithm requires some assumptions about the network topology that are not always possible to be made. Covariance intersection fixes these problems, by being independent of the network layout. However, as can be seen in table 4.2, fusion requires more data transfer and the time taken for calculations is also 190% more than in consensus. In short, when assumptions can be made about the network topology, the weighted consensus is best suited for a fully connected network.

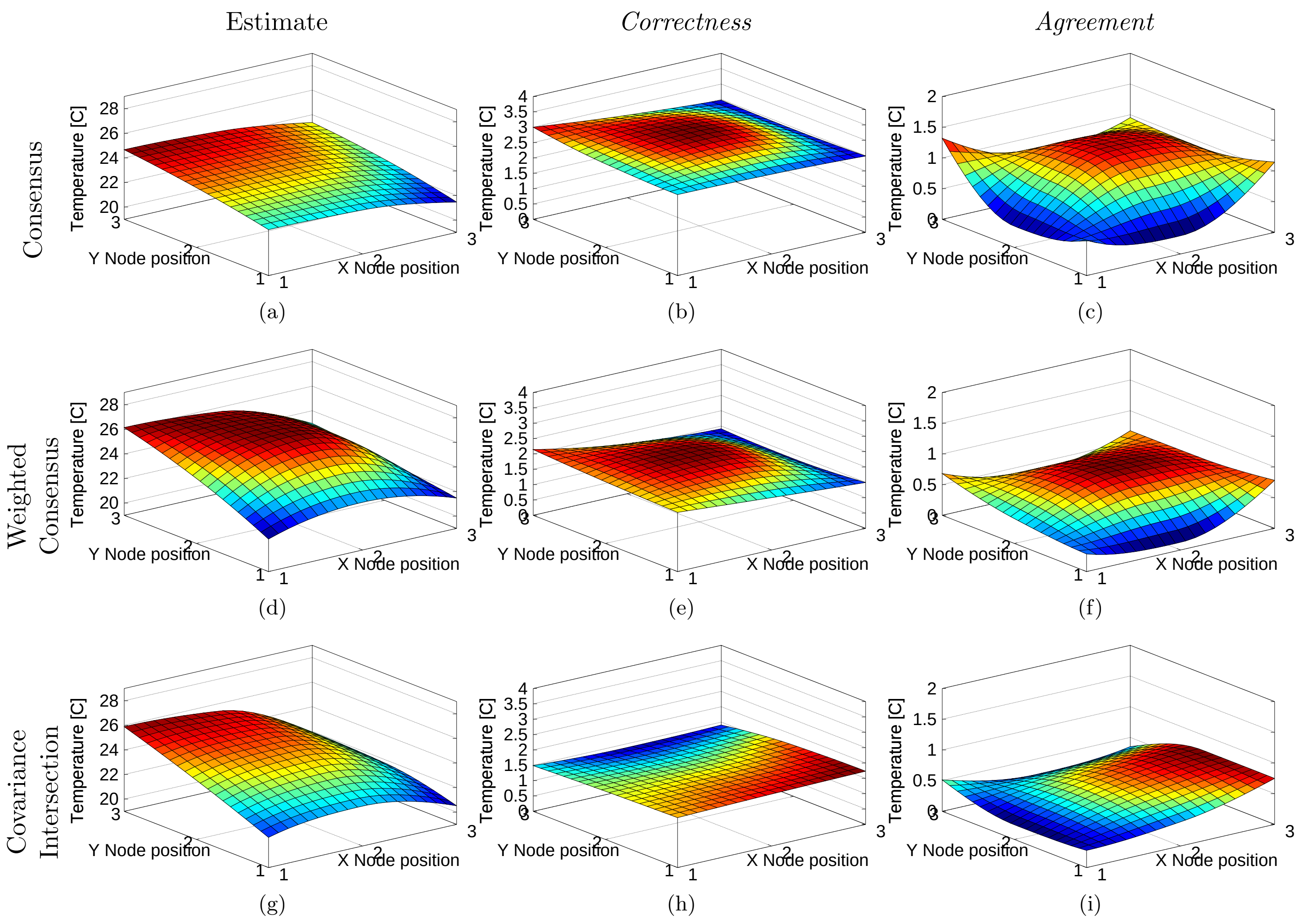


Figure 4.7: The estimate results, metric *Correctness*and metric *Agreement*for consensus, weighted consensus and covariance intersection for the Mesh Network

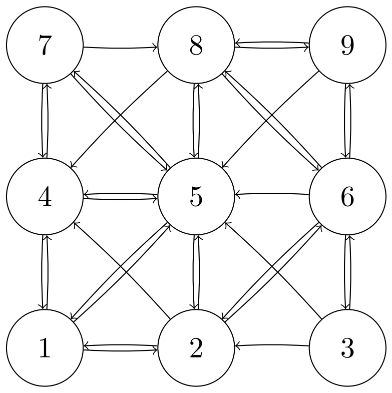


Figure 4.8: The mesh network as used in the experiments.

4.6 Mesh Network

The second network layout that is evaluated, is a mesh network. In a mesh network, not every node can communicate with every other node. In most cases, this is the network that one has in a real life setup, because of the distance between the nodes and communication failures. As such, nodes can only communicate with its direct neighbors, limiting the amount of information a node has of the network and the temperatures therein.

The network as used in these experiments, is shown in figure 4.8. The arrows in this figure represents the direction of the communication. When no arrow is present, no communication is possible. In this graph, the limited connectivity of the network is visible: only nodes that are direct neighbors can communicate. In order for the experiment to be repeatable among runs, this setup is reliably achieved by ignoring received messages that are not in the whitelist of a node as represented in figure 4.8.

In the following sections, the three distributed Kalman models as described in section 3.8 will be discussed.

4.6.1 Consensus

As in the previous section, the first experiment conducted is the consensus algorithm. In this consensus algorithm, all incoming messages are weighted equally.

From figure 4.7(a), it is already visible that node 3 is not able to estimate the temperatures as correctly as with consensus in the fully connected network layout. The plotted surface is flat, disregarding the high temperatures in node 5, 7 and 8. This shows even better in figure 4.7(b) representing *Correctness*. It shows that the standard deviation of the error has increased to 3 degrees, opposed to the 2.2 degrees in the fully connected topology, a decrease in performance of 25%. This rise can be attributed to the lack of information available to the nodes, resulting in estimates that do not show the rise in temperature, resulting in a bigger error.

As in the previous experiments, the offset to the actual temperature is not the only metric. The other metric, is the agreement of the nodes on the temperatures they have estimated: metric *Agreement*. In figure 4.7(c) the results for this metric are shown. Here it is clearly visible that without global communication, consensus is not able to actually reach consensus. The nine nodes have highly different values for the temperatures in the network, reaching an error of 1.3 degrees. Furthermore, this error is not equal through the network, which was the case for the fully connected consensus algorithm, i.e. different nodes have different estimation errors.

The last three remaining metrics are metrics *Time*, *Memory*and *Storage*, which have not changed with regards to the fully connected consensus algorithm as they are network layout independent. They are presented in table 4.3.

From this experiment, it becomes clear that the consensus algorithm is not able to perform well when not in a fully connected network. Both the offset to the actual temperature as the offset between the nodes have risen with 25%.

4.6.2 Weighted Consensus

The next algorithm that is evaluated in the mesh network layout, is the weighted consensus algorithm. In this consensus algorithm, the incoming estimate vectors are weighted depending on the sender of the vector, and the sender's possible knowledge of the temperatures that it has estimated.

In figure 4.7(d), one can see that the estimates of the nodes have improved over the estimates by the naive consensus algorithm. Nodes that are affected by the heat source are visible in the results, a direct improvement over the naive consensus algorithm.

The first metric as defined in 4.3, is metric *Correctness*and defines the offset of the estimates to the central Kalman filter. It is shown in figure 4.7(e). The error has risen with regards to the fully connected network layout, from 1.1 to 1.8 degrees. This is a direct result from the limited connectivity: because there is no direct communication, the information needs to flow through the network.

The next metric, *Agreement*, is the agreement of the temperature estimates between the different nodes in the network. This metric is shown in figure 4.7(f). As with the metric *Correctness*, the metric *Agreement*is also higher: 0.53 instead of the 0.17 from the fully connected network

The last three metrics metric *Time*, *Storage*and *Memory*, have not changed since these are network layout independent. They are presented in table 4.3.

4.6.3 Covariance Intersection

The third experiment in the mesh network layout is the covariance intersection algorithm. As described in 3.8, in covariance intersection the covariance matrix is send along the estimate vector, potentially increasing the correctness of the estimates with regard to the actual temperature.

In figure 4.7(g) it is visible that node 3 is able to estimate the temperatures relatively well, with clear differences between the nodes that are affected by the heat source and those who do not.

In figure 4.7(h) metric *Correctness*is depicted. Here, the strength of covariance intersection is visible. Covariance intersection is able to keep the mean error the same as in the fully connected network layout, 1.6 degrees. Furthermore, the metric *Agreement*in figure 4.7(i) also shows a more stable result than the other two methods with only a slight rise in error.

The last metrics, metric *Time*, *Memory*and *Storage*are independent of the network layout and have remained the same. They are presented in table 4.3.

4.6.4 Summary

With all three experiments performed in a mesh network, a comparison between the three can be made. Compared to the fully connected network however, it has become more difficult to get a reliable view of the temperature that is shared among all nodes. A table showing all results from the mesh network experiments is table 4.3.

As with the fully connected experiments, all three methods give different results. However, both the weighted consensus and the fusion algorithm are still able to give a temperature estimate with an error with a standard deviation of 2.3 degrees. The naive consensus algorithm however has seen an increase in its error with almost 30%. The error in the agreement among the nodes has also increased for most algorithms, remaining below 0.5 degrees for both weighted consensus and covariance intersection.

4.7 Fully Connected Network with reduced coverage

The third configuration that is tested with the three distributed Kalman algorithms, is a configuration with more grid cells than sensors. The temperature model in the Kalman filter is setup is such a way, that it is able to estimate temperatures in grid cells even though there is no actual sensor deployed in that grid cell. For this experiment, four sensor nodes are removed from the grid, which results in a grid with holes. The connectivity of the nodes are not limited, so in effect, the nodes are in a fully connected

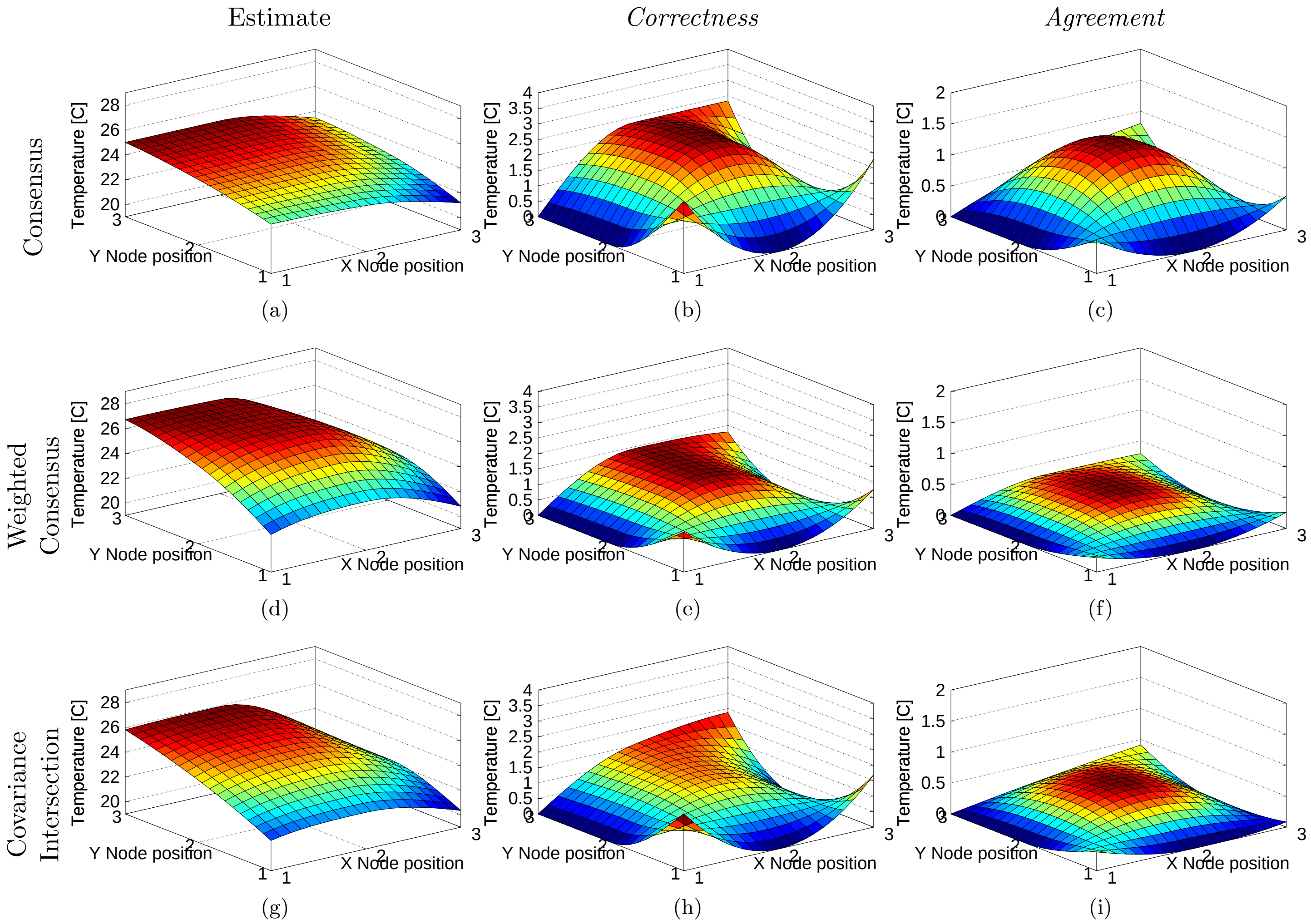


Figure 4.9: The estimate results, metric *Correctness*and metric *Agreement*for consensus, weighted consensus and covariance intersection for the Fully Connected Network with reduced coverage

Metric		Consensus	Weighted Consensus	Covariance Intersection
	Minimum [^{o}C]	2.43	1.35	1.37
Correctness	Maximum [^{o}C]	3.02	2.25	1.73
	Mean [^{o}C]	2.67	1.77	1.57
	Minimum [^{o}C]	0.29	0.12	0.24
Agreement	Maximum [^{o}C]	1.32	0.84	0.75
	Mean [^{o}C]	0.84	0.53	0.46
Memory		18	18	112
Time		2.37	2.40	385.62
Storage	ROM [b]	916	1300	1726
	RAM [b]	124	498	1672

Table 4.3: Summary of all metrics of the mesh network experiments. Note that the represented values are error values.

network just as in section 4.5. The grid layout with the missing sensors is depicted in figure 4.10.

In the following sections, the performance of the three different distributed Kalman Filter methods as described in section 3.8 will be assessed.

4.7.1 Consensus

The first experiment in this network layout, is the consensus algorithm. As described in section 3.8, in this algorithm all incoming messages are averaged, without any form of weighing.

From figure 4.9(a), it can be seen that the node is still able to estimate the temperature in all grid cells, although no sensor input from every cell in the network is available

In figure 4.9(b), the error relative to the central Kalman filter (metric *Correctness*) is shown. The error is in the same range as the error during the fully connected consensus algorithm. This is not completely surprising, as both network layouts are fully connected.

In figure 4.9(c) *Agreement*is shown. The error in the agreement however has risen and has almost doubled with relation to the fully connected network layout. In short, it can be stated that the naive consensus algorithm is not robust against the reduced coverage. The missing information from

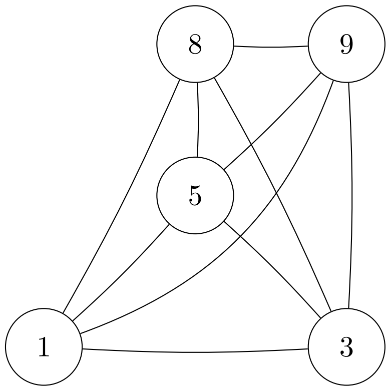


Figure 4.10: The network as used in the experiments.

these cells are not estimated well, decreasing performance on all metrics.

For metrics *Time*, *Storage*and *Memory*no changes have occurred, although the average number of incoming messages has decreased, resulting in less computational power usage. The metrics are presented in table 4.4.

4.7.2 Weighted Consensus

The second experiment conducted with a network with missing nodes, is the weighted consensus algorithm. In this algorithm, as described in section 3.8, the members of the incoming estimation vector are weighted based on the sender.

From figure 4.9(d), it is clear that the weighted consensus algorithm is able to estimate temperatures reasonably well, with a profile similar to the profiles in the previous experiments.

Metric *Correctness*is shown in figure 4.9(e) and shows an error slightly higher than in the fully connected case: 0.2 degrees more. This error can be attributed to the limited amount of knowledge that is available in the network about the temperatures in the grid cells without any sensors.

The metric *Agreement*is about the agreement of the nodes about their estimates. If all nodes have the same values for the temperatures, this metric will be lower. This metric in shown in 4.9(f). As with the naive consensus algorithm, the error of the agreement has increased slightly, with 0.2 degrees.

Metric *Time*, *Storage*and *Memory*are independent of network layout and are therefore not changed. They are presented in table 4.4.

4.7.3 Covariance Intersection

The last experiment that is performed with a network with empty grid cells, is with a Kalman Filter using covariance intersection. In covariance intersection, not only the estimate vector is transmitted, also the covariance matrix is shared among its neighbors.

From figure 4.9(g), it can be seen that the estimation results are quite good. From figure 4.9(h), it shows that the maximal error from the actual temperature is 1.8 degrees, the same error that has been reached with the fully connected network layout.

The agreement among nodes, *Agreement*is shown in figure 4.9(i). It can be concluded that there is only a small disagreement among the nodes. The highest peak is at 0.6 degrees. This is comparable with the results from the weighted consensus and almost 50% better than the naive consensus and the same as the error achieved with the fully connected network. It can therefore be stated that the covariance intersection algorithm has no problems with the empty grid cells.

As explained earlier, metric *Time*, *Storage*and *Memory*are independent on the algorithm and have not changed with regard to the previous fusion implementation. They are presented in table 4.4.

4.7.4 Summary

Concluding from the three experiments with this network configuration, a configuration with reduced coverage, a method selection can be performed. All results from the three experiments is shown in table 4.4.

From table 4.4, the differences between the naive consensus and the other two experiments can clearly be seen. The consensus algorithm is not able to cope with the fact that there is no information from all cells. The covariance intersection and weighted consensus however do perform well. Both the naive consensus and the weighted consensus have more difficulty now that there are grid cells without any sensor. Covariance intersection however, has relatively less trouble with this fact, although in absolute numbers covariance intersection and weighted consensus are comparable. From this, it can be concluded that depending on the assumptions that can be made, weighted consensus is best suited for this application. However, when the network positions are not known, covariance intersection shows that is very robust and able to estimate temperatures, even when the number of grid cells is larger than the number of nodes measuring temperature.

4.8 Method Selection

In the previous sections, the three algorithms as described in section 3.8 were tested for three different network layouts in both a actual experiment and a simulation. From these experiments a method selection can be made resulting in a method that is best suited for a given task.

Depending on the kind of network that will be setup, different methods can be used. From the previous sections, it follows that using the naive

Metric		Consensus	Weighted Consensus	Covariance Intersection
	Minimum [^{o}C]	2.27	1.26	1.37
Correctness	Maximum [^{o}C]	2.65	1.50	1.86
	Mean [^{o}C]	2.38	1.35	1.65
	Minimum [^{o}C]	0.51	0.23	0.08
Agreement	Maximum [^{o}C]	1.28	0.49	0.58
	Mean [^{o}C]	0.73	0.31	0.32
Memory		18	18	112
Time		2.37	2.40	385.62
Storage	ROM [b]	916	1300	1726
	RAM [b]	124	498	1672

Table 4.4: Summary of all metrics of the fully connected network with reduced coverage experiments. Note that the represented values are error values.

consensus is never the best option. In all metrics, it is unable to perform better than the local Kalman filter on the other two distributed Kalman Filters. The weighted consensus filter however, is a cheap alternative to the naive consensus: it performs better at all metrics than the naive implementation, while barely regressing the metrics *Time*, *Memory*and *Storage*concerned with computational and communication power.

The major choice an engineer there for has, is in the choice between covariance intersection and the weighted consensus algorithm. A major plus of the covariance intersection algorithm is that it just works. From the previous sections, it can be seen that during all three experiments, the results remain in the same order of magnitude, barely regressing when communication is difficult or the number of sensors is low. Covariance intersection however comes with a cost. As described in section 3.8, for covariance intersection to work additional internal state from the Kalman filter needs to be exchanged with its neighboring nodes, 112 bytes instead of the 18 used in the consensus based algorithms. Although computation is not as power consuming as communication, the rise in the computational time and thus power is large as well. The strength in covariance intersection however is, next to its robustness against network layout, the fact that it does not need to assume anything about its neighbors or the network. Where weighted consensus requires knowledge about all its possible neighbors, covariance intersection does not.

In a network where topology assumptions or indirect ways of getting the information about the topology is not a problem, weighted consensus can be considered. In all three experiments is performs well, usually as the best or just beaten by covariance intersection. This is caused by the implicit covariance intersection that is inherit in the weighted consensus. This implicit fusion however is static and cannot change along with its network, requiring reconfiguration any time the network changes. The implicit covariance intersection however is able to reduce the computational time and number of bytes per transmission to such an extent, that the increased battery life may be worth the reduction in flexibility.

Chapter 5

Conclusions and Future Work

Following the preceding chapters, it is possible to answer the research questions as stated in section 1.1. Furthermore, several suggestions for future work are stated.

5.1 Conclusions

The problem statement of this work is: What are the different methods of implementing Gossip Based Distributed Kalman Filtering and how does it perform.

This problem statement and the research questions can now be answered.

1. What form of Gossip is best suited for Gossip based distributed Kalman Filter?

   In section 2.1, relevant gossip algorithms are explained and assessed. From this, the quick convergence rate and the large network coverage of Broadcast gossip show that using gossiping information can be spread quickly and efficiently.

2. What are the different ways of distributing information in a distributed Kalman filter and how do they compare?

   Sections 2.2 and 3.8 discussed in depth the different ways of distributing the distributed Kalman filter and its implementation details. It followed that both a weight based consensus algorithm and covariance intersection is able to perform an accurate distributed Kalman filter, where covariance intersection is always able to perform well independent of knowledge about the network topology.

3. What limitations are encountered when implementing Gossip based Distributed Kalman Filter and how can one cope with those?

In chapter 3 the implementation details for this work are presented. Although the used network nodes, the TelosB, are low powered units they are able to perform the extensive matrix calculations. In the temperature case, these calculations using fixed point arithmetic have an error of only 0.1 degrees Celsius and should not pose for any problems.

5.2 Future Work

In this work, different distribution methods are accessed for a distributed Kalman filter using gossiping. However, there are ways in which the work can be improved. In this section, some suggestion are made that potentially can improve the results or make the network behave different.

1. As the main goal of this thesis was the comparison of the different distribution methods for a distributed Kalman filter, the results are not as optimal as they can be. As discussed in section 2.3, the temperature model that is used is a linear model. The results can therefor be improved by switching to a non-linear model or to a extended or unscented Kalman filter.

2. The only Gossip algorithm that is evaluated is the broadcast gossip algorithm. However, many other gossip algorithms exist in literature. It is interesting to also experiment with different gossip algorithms to how this influences the results from chapter 4.

Bibliography

[1] a.D. Amis, R. Prakash, and T.H.P. Vuong. Max-min d-cluster formation in wireless ad hoc networks. *Joint Conference of the*, 1:32–41, 2000.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, Mar. 2002.

[3] TC Aysal and ME Yildiz. Broadcast gossip algorithms. *Theory Workshop, 2008.*, pages 343–347, May 2008.

[4] Tuncer C. Aysal, Mehmet E. Yildiz, Anand D. Sarwate, and Anna Scaglione. Broadcast gossip algorithms: Design and analysis for consensus. *2008 47th IEEE Conference on Decision and Control*, pages 4843–4848, 2008.

[5] Stephen Boyd, A Ghosh, and B Prabhakar. Randomized gossip algorithms. *Information Theory, IEEE*, 2006.

[6] Stephen Boyd, A Ghosh, and B Prabhakar. Randomized gossip algorithms. *Information Theory, IEEE*, 2006.

[7] KK Chintalapudi, D Ganesan, and Alan Broad. A wireless sensor network for structural monitoring. *networked sensor*, 2004.

[8] ADG Dimakis and AD Sarwate. Geographic gossip: Efficient averaging for sensor networks. *Signal Processing, IEEE*, (Ipsn):1–15, 2008.

[9] Eiman Elnahrawy. Cleaning and querying noisy sensors. *international conference on Wireless sensor*, page 78, 2003.

[10] D Estrin, L Girod, and G Pottie. Instrumenting the world with wireless sensor networks. *Acoustics, Speech, and*, 2001.

[11] David Gay, P Levis, R Von Behren, and Matt Welsh. The nesC language: A holistic approach to networked embedded systems. *Acm Sigplan*, 2003.

[12] I Gupta and Denis Riordan. Cluster-head election using fuzzy logic for wireless sensor networks. *Communication Networks*, 2005.

[13] ZJ Haas and JY Halpern. Gossip-based ad hoc routing. *INFOCOM 2002. Twenty-First*, pages 1–10, 2002.

[14] Ali Jadbabaie, Jie Lin, A Stephen Morse, Ali Jadbabaie, Jie Lin, and A Stephen Morse. Coordination of Groups of Mobile Autonomous Agents Using Nearest Neighbor Rules Coordination of Groups of Mobile Autonomous Agents Using Nearest Neighbor Rules. *IEEE Transactions on Automatic Control*, 48(6):988–1001, 2003.

[15] S.J. Julier. A non-divergent estimation algorithm in the presence of unknown correlations. *American Control Conference, 1997.*, 4:2369–2373, 1997.

[16] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35, 1960.

[17] R Karp and C Schindelhauer. Randomized rumor spreading. *of Computer Science,*, 2000.

[18] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 482–491. IEEE Computer. Soc, 2003.

[19] B. Legat, K. Hofmann-Wellenhof, and M. Wieser. *Navigation, Principles of Positioning and Guidance*. Springer, 2003.

[20] MEMSIC. The TelosB platform.

[21] EM Mikhail. *Observations and least squares*. IEP, New York, New York, USA, 1976.

[22] Moteiv. TMote Sky, 2006.

[23] Wolfgang Niehsen. Information Fusion Based On Fast Covariance Intersection Filtering. *ISIF*, pages 901–904, 2002.

[24] R. Olfati-Saber. Consensus filters for sensor networks and distributed sensor fusion. *Decision and Control, 2005 and*, (1):6698–6703, 2005.

[25] R. Olfati-Saber. Distributed Kalman filter with embedded consensus filters. *Decision and Control, 2005 and 2005*, pages 8179–8184, 2005.

[26] R. Olfati-Saber. Distributed Kalman filtering for sensor networks. *Decision and Control, 2007 46th IEEE*, pages 5492–5498, 2007.

[27] G.J. Pottie. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[28] Andrei Pruteanu, Venkat Iyer, and Stefan Dulman. NetSize: Gossip-based Algorithms for Network Size Estimation, 2011.

[29] Lawrence G Roberts and Barry D Wessler. Computer network development to achieve resource sharing. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, AFIPS '70 (Spring), pages 543–549, New York, NY, USA, 1970. ACM.

[30] Luca Schenato. A distributed consensus protocol for clock synchronization in wireless sensor network. *Decision and Control, 2007 46th IEEE*, pages 2289–2294, 2007.

[31] Sensirion. Humidity and Temperature Sensor SHT1x, 2011.

[32] Devavrat Shah. Gossip Algorithms. *Foundations and Trends in Networking*, 3(1):1–125, 2007.

[33] X Sheng and YH Hu. Distributed particle filter with GMM approximation for multiple targets localization and tracking in wireless sensor network. *on Information processing in sensor*, pages 181–188, 2005.

[34] Joris Sijs. *State-estimation in networked systems*. PhD thesis, TU Eindhoven, 2012.

[35] P Singh. Wireless Sensor Networks for Habitat Monitoring. *cs.umd.edu*, page 88, 2002.

[36] Stanford. TinyOS.

[37] B Warneke, M Last, and B Liebowitz. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 2001.

[38] Greg Welch. An introduction to the Kalman filter. *University of North Carolina at Chapel*, pages 1–16, 1995.

[39] G Werner-Allen, Konrad Lorincz, and M Ruiz. Deploying a wireless sensor network on an active volcano. *Internet Computing*, (April):18–25, 2006.