spiNNaker - Sensor Fusion

eingereichter
Projekt Report
von

Simon Trendel
geb. am 16.02.1990
wohnhaft in: München
Geibelstr 10
81679 München
Tel.: 0176/62983730

Benedict Simlinger
geb. am 11.10.1988
wohnhaft in:
Geibelstr 10
81679 München

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Univ.-Prof. Dr.-Ing./Univ. Tokio Martin Buss

Betreuer: Cristian Axenie
Beginn: March 2015
Abgabe: June 2015

In your final hardback copy, replace this page with the signed exercise sheet.

Abstract

Essential part of mobile robotics application is gaining correct information about the robot's own position and heading in its environment. The noisy and partially observable nature of the environment calls for sensor fusion, which tries to extract reliable information from related multi-modal sensory queues. At the same time, mobile robot applications impose strict constraints on the real-time processing hardware. This work presents two different sensor fusion strategies implemented for embedded hardware. The first is based on Kalman filters and the second strategy is a bio-inspired system designed at TU Munich.
With parameter tuning, knowledge about the physical system and disregard of fault tolerance the Kalman based implementations outperforms the suggested network architecture in computation time and quality of results. If any of the aforementioned conditions can not be met the suggested network architecture is a viable solution embedded sensor fusion.

Contents

Glossary

associated base map value ($v' = v'_i$) denotes the value of the base map m'_i associated to offset map m''_i and preprocessing map m'''_i.

associated offset map value ($v'' = v''_i$) denotes the value of the offset map m''_i associated to base map m'_i.

associated preprocessing map value ($v''' = v'''_i$) denotes the value of the preprocessing map m'''_i associated to base map m'_i.

base map (m') a map that is elicied by sensory input and other maps.

base map set (s') the map set of base maps.

initial trust factor ($\eta* = \eta*_{i,j} \forall i, j$) denotes how much trust map m has into the information reived from map n without any prior information.

local error (e) the discrepancy between two map values v_m and v_n denoted as $e_{m,n}$.

map (m)a unit x within the architecture that holds a map value v_x which represents its current belief about the environment, a vector of local errors and a vector of trust factors which hold the respective values.

map set (s) the set of all maps in the architecture.

map value (v)the current belief of a map about the environment. v_x denotes the map value of map x.

offset map (m'') this is an optional map type of the architecture. It holds the offset between two base maps.

preprocessing map (m''')this is an optional map type of the architecture. It helps to preprocess raw sensory input..

readout map (M) this map is connected to all base maps and fuses their communicated map values.

readout map value (V)the current belief of a map about the environment.

relative error (E) the local errors between two maps m and n relative to the average local error of all other maps.

sensor input (f) input from sensors. f_i denotes the sensor input associated to map i.

SpiNNaker a hardware platform developed at the University of Manchester that provides a big number of low-power cores for parallel computing..

trust factor (η) $\eta_{m,n}$ denotes how much trust map m has into the information reived from map n.

tubotron a program written by the University of Manchester for receiving UDP messages from SpiNNaker boards via Ethernet.

ybug a perl script written by the University of Manchester for interaction with the SpiNNaker baord.

Chapter 1

Problem Statement

Essential part of mobile robotics application is gaining correct information about the robot's own position and heading in its environment. This information required to be reliable since future planning and successful execution of tasks depend on it. The noisy and partially observable nature of the environment calls for sensor fusion, which tries to extract reliable information from related multi-modal sensory queues. At the same time, mobile robot applications impose strict constraints on the real-time processing hardware. Power consumption, weight and physical dimensions of the hardware must be adhere to the requirements of the mobile system.
This work presents two different sensor fusion strategies. The first is based on Kalman filters and the second strategy is a bio-inspired system designed at TU Munich. Both strategies must be applicable for real-time sensor fusion on embedded hardware.

Chapter 2

SpiNNaker Architecture

SpiNNaker is a hardware architecture designed at the university of Manchaster supported by the human brain project. The hardware enables massive parallel computing and is targeted at application in neuroscience, robotics and computer science. In this project, SpiNNaker's architecture reflects the distributed, asynchronous and bio-inspired design of the proposed architecture and was therefore chosen as test platform.

2.1 Harware properties

This work was conducted on the SpiNN-3 model of the SpiNNaker family which may also be referred to as machine 102. It features four chips with 18 [1] ARM968 [2] cores per chip as displayed in Figure 2.1. The ARM cores clock at 200 Mhz, lack a floating point unit but feature FIQs [3] in addition to IRQs. The logical schemata of a chip and a core are displayed in Figure 2.2. The board measures nine by eight centimeters, requires a 5V 1A power supply and provides a 100 Mbps Ethernet connection for communication and control.

2.2 Workflow

The team behind SpiNNaker provides a lot of documentation, background information and resources for their hardware which might be overwhelming at first. This sections contains some practical information on the general work flow with SpiNNaker.

[1] Only 16 cores per chip can be used for application because two cores per chip are reserved for administrative tasks and redundancy.

[2] for detailed information visit ARMS's info site.

[3] fast interrupt requests which allow more fine grained control of interrupts in real time application

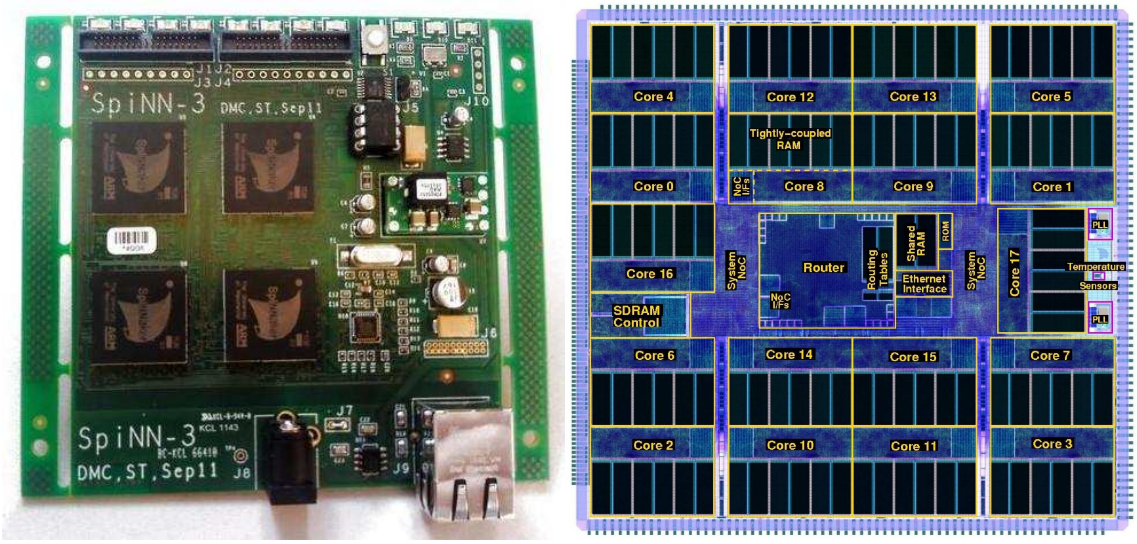


Figure 2.1: Left the SpiNN-3 board, with 4 chips is shown. On the right the chip die is displayed, exhibiting the arrangement of cores and auxiliary processors..

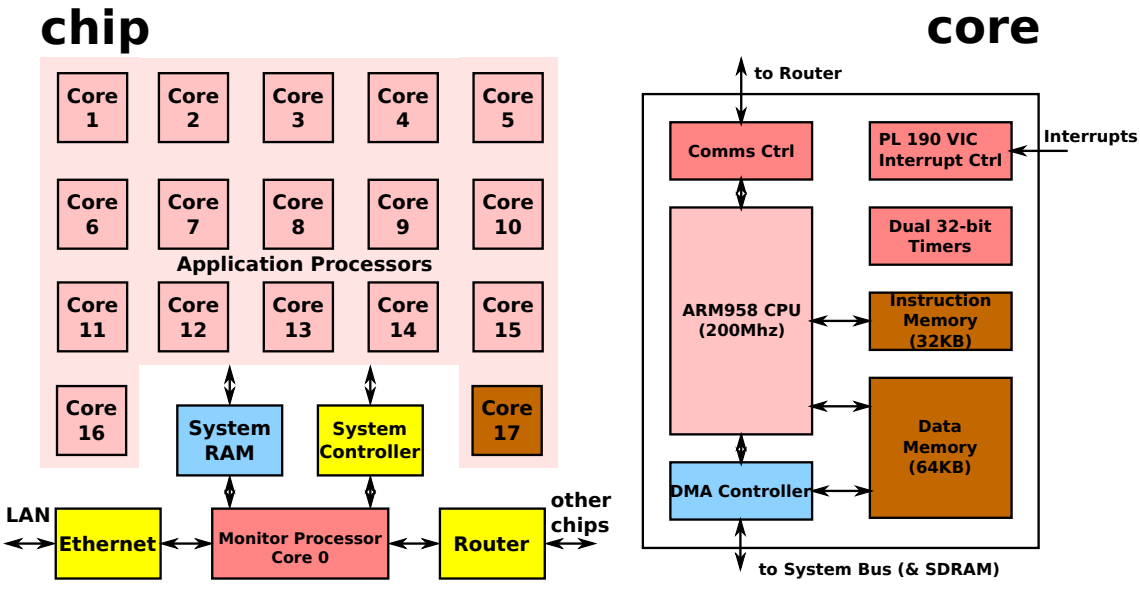


Figure 2.2: left the spinn-3 board, with 4 chips. On the chip die is displayed, exhibiting the arrangement of cores and router.

2.2.1 Workflow data

SpiNNaker does not support floating point calculations and requires fixed point formatted data. Although several fixed point formats are supported, *signed accum* is the only comprehensively tested format and therefore recommended. It is a 32 bit format, with one signed bit, 16 bit for integer and 15 bit for digits after the radix point. The steps to push and pull data to the SpiNNaker board are

1. use MATLAB to save floating point data in binary floating point file
2. use a C++ program to convert the binary floating point data to binary fixed point data
3. use ybug to push the data to the SpiNNaker board
4. use ybug to pull the data from the SpiNNaker board after simulation
5. use a C++ program to convert the binary fixed point data file to a binary floating point data file
6. use MATLAB to read in the binary floating point file

The necessary MATLAB scripts and C++ programs are part of this project and are provided in the code repository.

2.2.2 Workflow compiling

Since the target platform is ARM the source code must be cross compiled on any PC platform. The SpiNNaker project provides a cross compiler binary, example code and make files. The result are *.aplx* files which are loaded to SpiNNaker via ybug.

2.2.3 Workfow Spinnaker control

Two programs are provided for interaction with SpiNNaker boards. ybug for issuing commands to the SpiNNaker boards and tubotron for receiving messages from the board during runtime. Note, that the messages are sent as UDP packets so that lost packets will not be resent.

Chapter 3

Sensor Fusion Network on Spinnaker

The implemented network is based on [AC14] and follows a bio-inspired approach for sensor fusion. In the architecture, maps represent different sensor modalities which try to find a common belief by exchanging exclusively local information. Based on this asynchronous, continuous data exchange the maps update their local belief in a gradient descent fashion until they converge to a relaxed state within a given time limit.

3.1 Architecture

the network may consists of up to four different types of maps which perform distinct tasks within the architecture:

Base maps are the core of the architecture. They are connected to all other base maps, their associated offset map, their associated preprocessing map and the readout maps. Each base map corresponds to one sensor and implements the systems inner representation of the associated sensor value.

Offset maps are optional maps. They represent the offset between their associated base map and all other distinct base maps. Therefore they require connection to all base maps. Note that offset maps may be interpreted as a redundancy mechanism for a network that consists of base maps and a readout map.

Preprocessing maps are optional maps. They integrated differential sensor input (or differentiate integral sensor input) and thereby help to stabilize the map value of their associated base map. They are only connected to their respective base map.

Readout maps are obligatory part of the network and connect to all base maps. Their task is to fuse the output of the base maps.

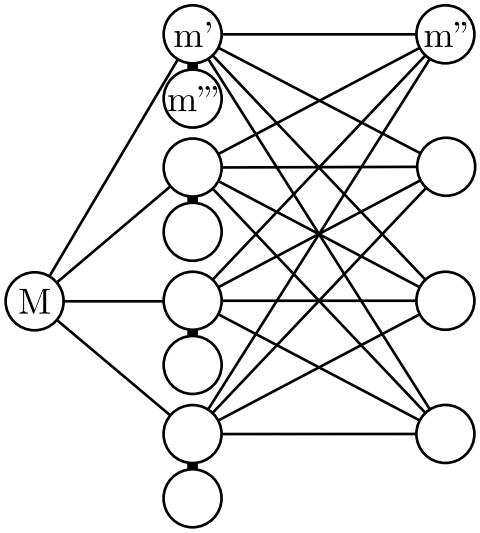


Figure 3.1: Depiction of the sensor fusion network as proposed by [AC14]. The readout map and base maps are are compulsory, preprocessing maps and offset maps are optional. In this case, preprocessing maps are active, therefore sensor input elicits the architecture through the preprocessing maps.

All maps exchanged information with a map set specific to their type. Additionally base maps or preprocessing maps may be exclusively elicited by sensor input depending on the architecture's setup. Each map weights incoming information with a trust factor associated to the source of information. The trust factor depends on the discrepancy between a remote source's communicated belief and a map's own belief about the system's state in its environment.

Map performs three steps when new information is received. First, calculate its mismatch to the source map of the information. Second, update its own map value with the new information. Finally update its trust factor in the source map.

3.1.1 Error Updates

The update of a map's own map value as defined in Equation 3.2 requires knowledge of the local error $e_{i,j}$ which is presented in this section and a trust factor $\eta_{i,j}$ which is discussed in subsection 3.1.3.

Depending on the type of maps involved in the update process of the local error, different functions must be used. The exhaustive collection of update functions is presented in Table 3.1. Each function may be subject to constraints as show in Table 3.2.

to i \ from j $e_{i,j} =$	m'	m''	m'''	M	f
m'	$v_i - v_j$	$v_i - v_j + {v''}_i$	$v_i - {v'''}_j$	$\sum_{k \in s'} v_k - \lvert s' \rvert \cdot V$	$v_i - f_i$
m''	${v'}_i - v_j + v_i$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
m'''	$v_i - {v'}_j$	$\emptyset$	$\emptyset$	$\emptyset$	$v_i - f_i$
M	$V - \frac{1}{\lvert s' \rvert} \sum_{k \in s'} v_k$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.1: Error are updated according to this tables. Columns represent the source map type and rows represent the destination map type. Pay special attention to map types and the use of values form associated map types. Note that the update from sensor can be done *either* in the preprocessing map *or* in the base map but not both. The choice depends on whether the architecture has to preprocess sensor input on its own or not.

to i \ from j	m'	m''	m'''	M	f
m'	$i \neq j$	$i \neq j$	$i = j$	$\emptyset$	$i = j$
m''	$i \neq j$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
m'''	$i = j$	$\emptyset$	$\emptyset$	$\emptyset$	$i = j$
M	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
f	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.2: Due to the networks structure each map type has a specific set of maps it is connected to. Within these sets certain constraints must be honoured. Base maps may update from any other base map except from itself as indicated by $i \neq j$. Further on preprocessing maps may only update from their associated base map value which is represented by $i = j$. All other constraints are to be interpreted accordingly.

3.1.2 Value Updates

Maps update their map value whenever new information is received. That happens when information from other maps is received or, in the case of either base maps or preprocessing maps, new sensor input stimulates the map.
When a map updates its map value, it tries to minimize the mismatch e between its own belief and the new information. Due to the nature of local error e as presented in subsection 3.1.1 the map value will be updated in a gradient descent fashion

$$\Delta v_i = -\eta_{i,j} \frac{\partial e_{i,j}}{\partial v_i} \tag{3.1}$$

$$v_i[t+1] = v_i[t] + \Delta v_i \tag{3.2}$$

weighted by a trust factor η, which is treated in subsection 3.1.3.
Note that Equation 3.2 also holds for map value updates from sensor input. In that case, $\eta_{i,j}$ is interpreted as the trust of a map in its own sensor input. This allows maps to dismiss information from its associated sensor if it is inconsistent with the information of other sources.
In order the propagate new insights about the environment through the network, maps inform their peers about their new belief. The implementation details of this procedure are discussed in section 3.2.

3.1.3 Trust update

The trust factor $\eta_{i,j}$ of a map m_i towards a source map m_j a depends on the local error $e_{i,j}$ and its relation to all other distinct and eligible local errors. This concept can be formalized in a general manner as

$$E_{i,j} = \frac{1}{|s|} \sum_{\forall k \in s} \text{abs}\left(\frac{e_{i,k}}{e_{i,j}}\right) \tag{3.3}$$

$$\Delta \eta_{i,j} = \eta * \cdot E_{i,j} \tag{3.4}$$

The term $\frac{e_{i,k}}{e_{i,j}}$ in Equation 3.3 should be interpreted as relative or normalized local error $e_{i,k}$ with respect to $e_{i,j}$. Be aware, that the map set in Equation 3.3 must be constrained according to Table 3.3 and that Equation 3.4 uses the initial trust factor which allows $\eta_{i,j}$ to relax temporarily in order to meet the architecture's design but prevents any kind of sustainable learning or adaptation.

3.2 Implementation

There are several implementation of the proposed architecture. The first implementation is written in MATLAB by the architecture's designer. It served as reference

to i	from j $s =$	m'	m''	m'''	M	f
m'		$s' \setminus \{i, j\}$	$s' \setminus \{i, j\}$	$s' \setminus \{i\}$	$s' \setminus \{i, j\}$	$s' \setminus \{i\}$
m''		$s' \setminus \{i, j\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
m'''		$i = j$	$\emptyset$	$\emptyset$	$\emptyset$	$e_{i,j}$
M		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
f		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 3.3: The contraints on the map sets used for calculating local error. For example when updating the local error of a base map from an offset map, the local error of the base map towards itself and its associated offset map value must be ignore (since they are undefined by definition).

implementation	files	comment	code	languages
reference MATLAB	4	650	1796	MATLAB
C port	39	1587	6056	MATLAB, C
SpiNNaker	10	390	1301	MATLAB, C, scripts

Table 3.4: Lines of code. Counted with cloc http://cloc.sourceforge.net/

code and catered for early proof of concept and visualization purposes. Second, a C implementation was created, which worked as drop in for the MATLAB implementation through MATLAB's MEX interface. It was written by two students under the Axenie's supervision. As a secondary target, the C implementation was tested on a 64-core server with the OpenMP API for parallelization tests. Lastly, the architecture was implemented in C for SpiNNaker boards within this work because SpiNNaker boards provide a more suitable hardware environment for the architecture that PCs. Table 3.4 provides a rough impression on the implementation complexity.

This project's implementations uses one core per map. That way, the advantages of SpiNNaker are deployed and load is evenly distributed to the hardware resources. At the same time, one chip with 16 available cores is sufficient for heading estimation or one dimension of position estimation. In total 33 cores are required, which consist of 4*3 cores for heading estimations, 3*3 core for x position estimation, 3*3 cores for y position estimation and three cores for reading out the estimation values.

The implementation works on preprocessed data, therefore the sensor input is stored locally (DTCM) at the base map cores and presented to the map at a pre-set interval. Note that this aims to mimic sensory input from a real sensor.

Sensor values were stored and processed in signed accum format because SpiNNaker boards do not support floating point format. Heading estimation takes values between 0 degree and 360 degree and therefore does not require any precautions against overflow. Position data was provided in a suitable value range.

The implementation follows an event-driven programming paradigm. With this

method functions, so called *callbacks*, are assigned to specific event classes which implement a programs *reaction* to an event. This allows to write well structured code and an implementation that closely follows the architectures design. On the down side, interrupt management and priority management require additional care. The means for assigning call backs as well as memory management were provided by libraries from the SpiNNaker team.

Information exchange by shared memory

In this implementation, the maps store their values in a share memory that is accessible to all maps. All maps read the information from the shared memory at the same time, process it, write back their results and wait for the next signal to repeat aforementioned steps. That way the implementation takes advantage of SpiNNaker's parallel hardware while being easy to debug and to control.
At the same time, this implementation violates the local nature of the map value since the map value are openly accessible to all other maps. Further on the architecture works synchronously which does not adhere to the bio-plausible motivation of the architecture.

Information exchange by message passing

This implementation is based on the implementation described in section 3.2 but takes full advantage of SpiNNaker design. map values are not shared via memory but are stored locally on each core. Information is share via asynchronous message passing, which is provided by SpiNNaker's API and libraries. Note, that SpiNNaker was designed with spiking neural networks in mind. In those networks timing of signals is critical. Therefore SpiNNaker features an additional routing core which can process big volumes of messages sufficiently fast. The only limitation here is that messages may carry one or no value. As a consequence two-dimensional position information must spilt into one-dimensional input before processing.
Note that attention must be paid to the relation between message sending and callbacks on received messages. The architecture is driven my clocked elicitation of base maps. base maps will updated their local values whenever they receive new sensory input. In this implementation maps send their updated value to all other connected maps after every update of their local value. At the same times all maps will updated their local map value when new information is received. In a network of maps this will lead to message avalanches unless precautions are made. In this implementation this problem is solved by sending an updated message every $n - th$ update, where n is the number of connected neighbours of a map.

Chapter 4

Sensor Fusion

The robots of our generation are complex dynamical systems. Their control depends on accurate and reliable self-motion and environment estimations. These estimations usually depend on measurements from sensors. When multiple sensors are combined we find ourselves in the realm of sensor fusion. The task of sensor fusion, generally refers to combining multiple sensor signals into a coherent and possibly more accurate estimate, than any individually used sensor signal could provide. One of the most important algorithms for sensor fusion is the Kalman Filter. Even though the theoretical background is rather involved, due to its straight-forward application, the Kalman Filter has quickly become one of the most widely applied state estimation algorithms today. But the Kalman Filter has its prerequisites and special care must be taken in scenarios where these are not fulfilled.
In this chapter the theory behind Kalman filtering is discussed and some of its limitations are explored. A very useful extension of the Kalman Filter, called Covariance Intersection, is then presented, which alleviates some of the limitations in practical applications. The implementation of these algorithms on the neuromorphic platform spiNNaker completes this chapter.

4.1 Kalman Filter

For the Kalman-filtering theory to be applicable, our real world must be cast into a set of differential equations, the state space form [Zar09]:

$$\dot{x} = Fx + Gu + w \tag{4.1}$$

Where x is a vector containing the states of our system, F is the system dynamics matrix, u is the control vector, G is the control matrix and w is a white-noise process. The process-noise matrix is:

$$Q = E[ww^T] \tag{4.2}$$

The measurements in our system must be linearly related to the states in order for the Kalman filtering theory to be applicable:

$$z = Hx + v \tag{4.3}$$

Where H is the measurement matrix and v is white measurement noise. The measurement noise matrix is:

$$R = E[vv^T] \tag{4.4}$$

Equation 4.1-Equation 4.4 describe the continuous world. For a discretization one must find the fundamental matrix:

$$\Phi(t) = \mathscr{L}^{-1}[(sI - F)^{-1}] \tag{4.5}$$

where I is the identity matrix and $\mathscr{L}^{-1}$ the inverse Laplace transform. Another way to find the fundamental matrix is via a Taylor series expansion:

$$\Phi(t) = \exp(Ft) = I + Ft + \frac{(Ft)^2}{2!} + ... + \frac{(Ft)^n}{n!} \tag{4.6}$$

If we sample our system every T_s seconds the fundamental matrix simply has to be sampled at these timesteps:

$$\Phi_k = \Phi(T_s) \tag{4.7}$$

Equation 4.3 and Equation 4.4 can now we stated discretely as:

$$z_k = Hx_k + v_k \tag{4.8}$$

$$R_k = E[v_k v_k^T] \tag{4.9}$$

Under the assumption that the control u_{k-1} is constant between sampling intervals, the discrete control matrix G_k is given by:

$$G_k = \int_0^{T_s} \Phi(\tau) G \Phi(\tau)^T d\tau \tag{4.10}$$

The discrete process noise matrix Q_k is given by:

$$Q_k = \int_0^{T_s} \Phi(\tau) Q \Phi(\tau)^T d\tau \tag{4.11}$$

Discrete Kalman filtering equation

$$\hat{x}_k = \Phi_k \hat{x}_{k-1} + G_k u_{k-1} + K_k(z_k - H\Phi_k \hat{x}_{k-1} - HG_k u_{k-1}) \tag{4.12}$$

The following matrix Riccati equations are used to recursively calculate the Kalman gains K_k:

$$M_k = \Phi_k P_{k-1} \Phi_k^T + Q_k \tag{4.13}$$

$$K_k = M_k H^T (H M_k H^T + R_k)^{-1} \tag{4.14}$$

$$P_k = (I - K_k H) M_k \tag{4.15}$$

P_k in Equation 4.15 is a covariance matrix that represents errors in the state estimate after an update, whereas M_k in Equation 4.13 is a covariance matrix that represents errors in the state estimate before an update.

4.2 Sensor Signal Filtering

Suppose we are receiving sensory data from a system whose properties are not known to us. Everything we know about the system comes from the measurements of the state of the system, taken by a sensor. Assuming this signal to be constant

$$x = a_0$$

would mean that its derivative is zero

$$\dot{x} = 0$$

so the state-space representation of our model would be

$$\dot{x} = Fx$$

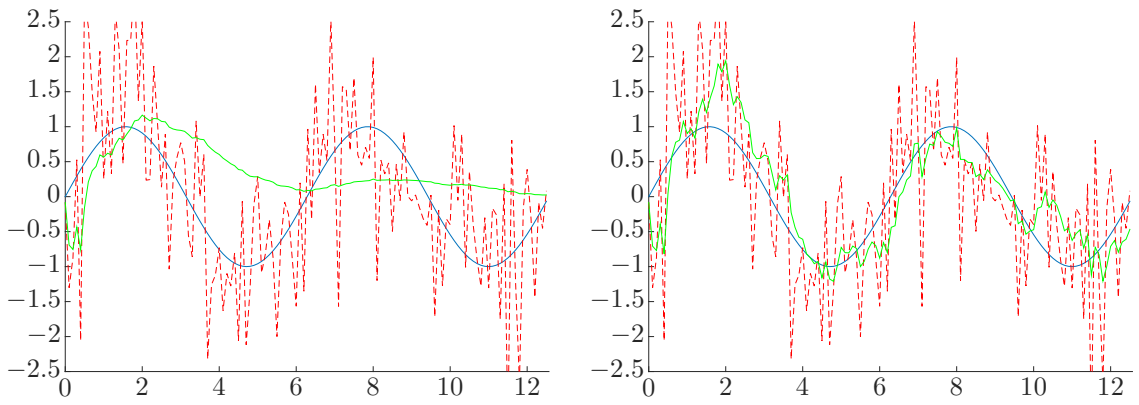
with $F = 0$.
In this case our fundamental matrix in Equation 4.6 is the identity matrix I and the Kalman filtering Equation 4.12 simplifies to:

$$\hat{x}_k = \hat{x}_{k-1} + K_k(z_k - \hat{x}_{k-1}) \tag{4.16}$$

This form of the Kalman filter is known as the zeroth-order Kalman filter. The assumption of the system to be static, i.e. the state of the system to be constant, may seem very implausible to you in case your robot or whatever is moving around. But as we will see soon, this modeling error can be compensated by telling the filter that we are really not sure about our model. This can be done by increasing the process noise Q_k in our system.
Figure 4.1a shows a sinus signal (blue) together with measurements (red dashed, zero mean gaussian noise, with unity standard deviation). With the above filter characteristics and assuming no process noise, the Kalman filter (green) is obviously unable to track the signal. While using process noise $Q_k = 0.05$ enables the filter to track the signal and improve, i.e. filter the measurements, as can be seen in Figure 4.1b. Increasing the process noise also increases the noise level that is propagated by the filter. For the particular case of a noisy sinus signal a much more accurate Kalman filter could be designed, exploiting information about the system, i.e. sinusoidal signal. As described above, such information about our system will not be available to us.

4.3 Sensor Fusion

Suppose now our system does not only have one sensor but two, both measuring the same physical quantity. Because the technical characteristics of any two sensors will in reality never be equal, the accuracy of the first sensor will differ



(a) without process noise (b) with process noise

Figure 4.1: Sinus signal Kalman filtered

from the other. This can easily be expressed through their respective means (a, b) and covariances (A, B). If the two measurements are statistically independent, the signals can be fused through a convex combination, i.e. maximum likelihood estimation(MLE)[CM01]:

$$C = (A^{-1} + B^{-1})^{-1} \tag{4.17}$$

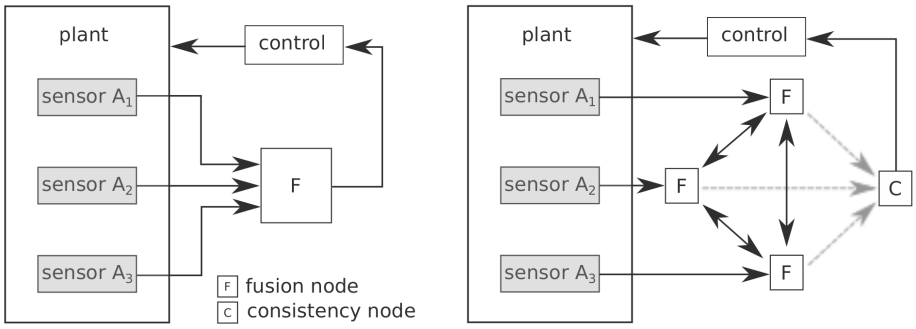
$$c = C(A^{-1}a + B^{-1}b) \tag{4.18}$$

Equation 4.17 represents the covariance of the two fused estimates, while equation 4.18 represents the fused mean. This result is only optimal, if the cross-covariance between the two sensor signals is zero [CM01].
If the cross-covariance is *exactly* known, the signals can be fused with the BLUE algorithm [CM01], which in case of gaussian signals is the maximum a posteriori(MAP). Because we will assume that the cross-covariance amongst the signals of our system are not known to us, another technique called Covariance Intersection will be presented in the next section. The assumption of unknown cross-covariance is mainly motivated by the fact that the correlation of multiple sensors induced by changes in humidity, temperature and especially by vibrations of the robot is usually very hard to acquire. Another motivation is the need for a consistent fusion algorithm, that can be used in a distributed fusion architecture.

4.4 Distributed Fusion Network

In increasingly complex systems it becomes favorable to design the system in a modular fashion. Some of the benefits of a modular design include [LCK+97]:

1. decomposition of a larger problem into smaller and easier to manage subproblems.



(a) centralized (b) decentralized

Figure 4.2: Data Fusion architectures

2. increased debugging abilities due to the possibility of testing modules in isolation.

3. the ability to replace modules by different, though in the global structure functionally equivalent modules.

4. the construction of fault-tolerant systems, where the failure of one module does not cause total failure of the system.

In the context of data/sensor fusion, there exist basically two approaches: centralized and decentralized. In a centralized data fusion architecture (illustrated in Figure 4.2a) a central fusion node is receiving input from all sensors. These signals are fused and propagated to some consumer, e.g. the control of a plant. One obvious drawback of such a design is the central role of the sensor fusion node. In case of its failure, the system would become uncontrollable. Another drawback is the necessity for synchronous measurements, i.e. the alignment of the sensory data. This can be quite obstructive when different types of sensors also have different acquisition rates, or the preprocessing of one sensor signal is more time-consuming than for the others. In such cases the fusion rate has to be throttled to the lowest sensor rate.
The drawbacks of a centralized architecture are alleviated in a decentralized architecture, which uses a modular design. As depicted in Figure 4.2b, there is no central fusion node, but rather several nodes. Each of these fusion nodes receives local sensory data only available to this particular node, but also receives the fusion results from all the other fusion nodes. The consistency node then chooses one of the estimates from the fusion nodes to propagate to some consumer, e.g. again the control of the plant. This architecture is obviously a lot more fault-tolerant than the centralized architecture, since even in case of failure of two of the three fusion nodes, one would still be available. This architecture also supports asynchronous, unaligned sensor data fusion. Each sensor can contribute at its own pace and the respective fusion node fuses these new measurements into the somewhat global estimate of the fusion network.

However one aspect has to be handled in such networks, namely feedback effects. Because each fusion node receives input from every other node, these estimates can be corrupted to an arbitrary extend by the nodes own results. This leads to an overall underestimation of the true error, which tempts the network to become overconfident, degrading the overall quality of the sensor fusion. For the Kalman filter theory to be applicable, the cross-covariance has to be known *exactly*[LCK+97]. In theory it would be possible to tag the estimates from each fusion node in some way, in order to exclude any self-correlation. The increasing communication overhead and degeneration of the modular design unfortunately limits such a realization to a small number of sensors. Another problem is the fact that the sensor signals are usually already to some extent correlated on measurement (e.g. through vibrations of the robot). These problems led to the invention of a new technique, called *Covariance Intersection*(CI) [IFA97].
The joint covariance of two sensor signals, represented by their respective mean and covariance (a, A) and (b, B), is given by:

$$C = \begin{bmatrix} A & X \\ X^T & B \end{bmatrix} \tag{4.19}$$

The requirement of precise knowledge of the cross-covariance X can be avoided by finding a covariance matrix M, whose diagonal blocks $M_A > A$ and $M_B > B$, such that

$$M \geq \begin{bmatrix} A & X \\ X^T & B \end{bmatrix} \tag{4.20}$$

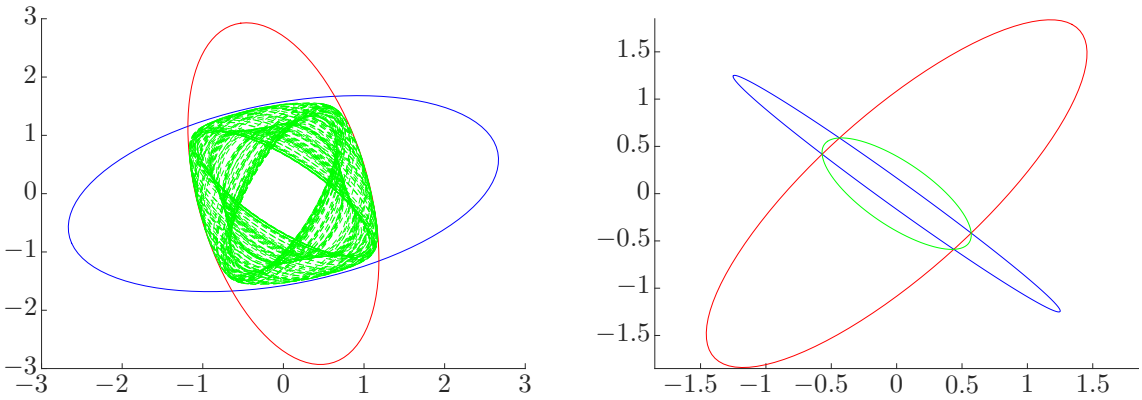
for any possible instantiation of the unknown cross-covariance X[IFA97]. This is illustrated in Figure 4.3a on the basis of again two sensor signal covariances (blue and red, 0.5 confidence ellipsoids of gaussian distributions) and multiple results of MLEs, with different cross-covariances. As can be seen, the MLE always lies in the intersection of the two covariance ellipsoids, hence the name. The CI algorithm finds the matrix M through the following equations [IFA97]:

$$M^{-1} = \omega A^{-1} + (1 - \omega)B^{-1} \tag{4.21}$$

$$m = M(\omega A^{-1}a + (1 - \omega)B^{-1}b) \tag{4.22}$$

Comparing the above equations with Equation 4.17 and Equation 4.18 reveals how CI is functioning as a weighted maximum likelihood estimator. The parameter ω in Equation 4.22 and Equation 4.21 serves as a weighting between the two measurements and can be found with respect to some performance criterion on M, e.g. the minimization of the trace or determinant of M. Figure 4.3b illustrates the function of CI(adapted from [STL09], the ellipsoids correspond to 0.5 confidence level of a gaussian normal distribution). The covariances of the two sensors are:

$$A = \begin{bmatrix} 0.8 & -0.7 \\ -0.7 & 0.8 \end{bmatrix}^2, B = \begin{bmatrix} 0.3 & 1.2 \\ 1.2 & 1 \end{bmatrix}^2$$



(a) MLE always lies in intersection (b) Covariance Intersection

Figure 4.3: Geometrical Motivation of CI

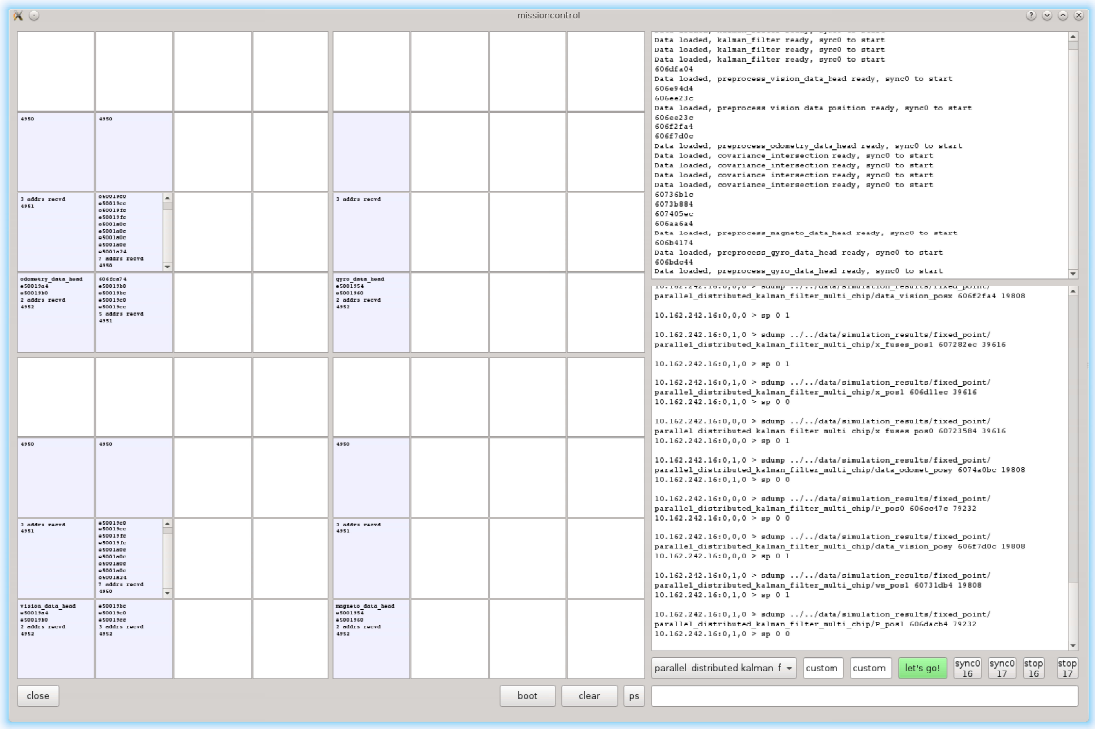
Sensor A (blue) is highly accurate along the measurement direction but has a wide detection aperture, while Sensor B (red) has poor accuracy. The fused covariance (green) is calculated from Equation 4.21, with a minimal trace of M as a performance measure (resulting $\omega = 0.1557$). As can be seen in Figure 4.3b, the CI algorithm locks onto the more accurate Sensor A, if the two signals are highly uncorrelated. This behavior is very useful in the decentralized fusion network, where the estimates from other nodes should be fused weighted by their accuracy. Comparing Figure 4.3b to the MLEs next to it reveals how CIs covariance encloses any MLE estimate and therefor serves as a conservative and robust fusion technique. This covariance can then be consistently in the Kalman filter equations of the decentralized sensor fusion network.

4.5 Implementation On SpiNNaker

SpiNNaker is an experimental platform, orginally designed for low-power consuming simulations of spiking neural networks. The platform used for the implementation is the spiNN-3. The following description of the spiNN-3 hardware and software interfaces serves only as a brief overview and the reader is referred to the projects website for a detailed description.
The spiNN-3 consists of four spiNNaker chips, where each chip consists of:

- eighteen ARM968 cores, clocked at 200 MHz
- SDRAM, 128 Mb
- router, for asynchronous communication among the cores and chips

The software provided by the spiNNaker project is:



(a) Qt GUI for spiNNaker

- spin1_api and sark (C code, implementing basic functionality for the chips, e.g. memory allocation, message passing etc.)
- ybug, perl command-line interface for communication with the platform via ethernet

The cross-compiler arm-none-eabi-gcc (Sourcery CodeBench Lite 2013.05-23, 4.7.3), is used for compilation of the C code.
The spiNNaker package comes with a GUI called tubotron, which is a simple UDP-package listener printing out messages, coming from each core on the platform. In the course of this project a new more powerful GUI is programmed in C++ together with the Qt-framework. This GUI implements the same functionality as tubotron, but has some additional features, mainly focussed on increased debugging functionality and workflow optimization. The GUI is depicted in Figure 4.4a. Besides a clearer layout and an included ybug commandline, the main advantage of this GUI is a switch in the UDP-listener that triggers automatic memory retrieval from spiNNaker. The addresses for this can be send from spiNNaker executing code. This is very convenient, when many simulations are run.

Chapter 5

Results

5.1 Simulation Results

5.1.1 Kalman filter

The zeroth-order Kalman filter as described in section 4.2 is implemented as a single core implementation running on one chip. It fuses preprocessed sensory signals from four different sensor types (gyroscope, magnetometer, odometry, vision). Figure 5.1 shows a comparison between the results from spiNNaker (red dashed) and the sensor signals (blue). The mean squared errors (MSE) in Table 5.1 show how closely the Kalman filter is able to follow the sensor signals. The parameters of the Kalman filter are $P_0 = 1000$, process noise $Q_k = 0.5$ and measurement noise $R_k = 1$. The high initial covariance P_0 expresses our uncertainty about the initial state ($x_0 = 0$). The execution time (averaged over 10 runs) T_{10} of this implementation amounts to $T_{10} = 56ms$, processing 4952 samples for each of the four sensor signals. The execution time includes loading the sensory data from SDRAM (where the preprocessed data is uploaded on program start) into DTCM and the result back to SDRAM. The execution time for each sensor signal filtering is $T_{10} = 19ms$. Therefor the next step is to distribute the four individual Kalman filters onto four cores. As expected, when distributing the kalman filtering onto four independent cores the execution time reduces to $T_{10} = 18,8ms$. This can be expected to be the fastest implementation of the zeroth-order Kalman filter, because the individual fusion signals are single values which have to be fused sequentially. A further parallelization is not possible due to the already atomic calculations.

	gyroscope	magneto	odometry	vision
MSE	0.035	0.068	0.035	0.146

Table 5.1: Mean squared error comparison spiNNaker and signals

So far the sensor signals used in the Kalman filter are already preprocessed. Since even the small spiNN-3 platform still has a lot more computing power to exploit,

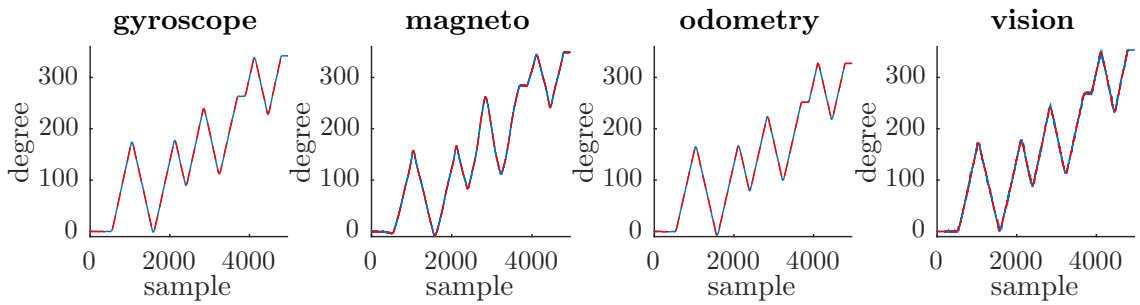


Figure 5.1: Comparison between Kalman filter and sensor signals

the next step is to implement the preprocessing of raw sensory data. Except for the odometry data the preprocessors can run independently on separate cores. For the transformation of body-fixed odometry data to global position estimates, a heading estimate is necessary from one of the heading preprocessors.

5.1.2 Distributed Fusion Network

The Covariance Intersection algorithm finds the weight ω in Equation 4.21 by minimizing the trace of M. The minimizer chosen for the implementation on spiNNaker is Brents method (adapted from [Pre07]), which can find the roots of a polynomial without usage of derivatives. The necessary arithmetics, such as matrix multiplication/inversion, are implemented.
The overall infrastructure of the implementation is illustrated in Figure 5.7. As described in section 4.4, each Kalman filter unit receives a state estimate from its preprocessor and fuses it with the current believe from the Covariance Intersection unit. This fused result is then sent to one of the other CI units, which fuses this estimate with the CI equations. Each functional unit depicted in Figure 5.7 corresponds to one core on the spiNNaker platform, where the distribution amongst the four chips is as marked. The communication among the cores is implemented via the System-BUS where possible. The inter-chip communication on spiNNaker is possible soley via multicast packages (MCPL). Figure 5.2 shows the heading simulation results for the different units (PP-h blue, KF-h red dashed, CI-H green). It can be seen comparing the subplots in Figure 5.2 how all CI nodes tend to level onto a global heading estimate. Figure 5.3 shows the calculated weights in the CI units. The odometry CI unit is obviously averaging the signal, while the other units alternate between preferring their own signal ($\omega = 1$) and the estimates received from the other nodes ($\omega = 0$).
The odometry heading estimation is in fact accumulating errors due to slippage of the robot wheels. Additionally both odometry and gyroscope preprocessors are integrating their sensor signals which introduces additional round of errors. This can be seen in Figure 5.2 through the heading difference between the odometry/gyroscope preprocessors and the other nodes.
This error accumulation becomes quite severe in the global position estimation.

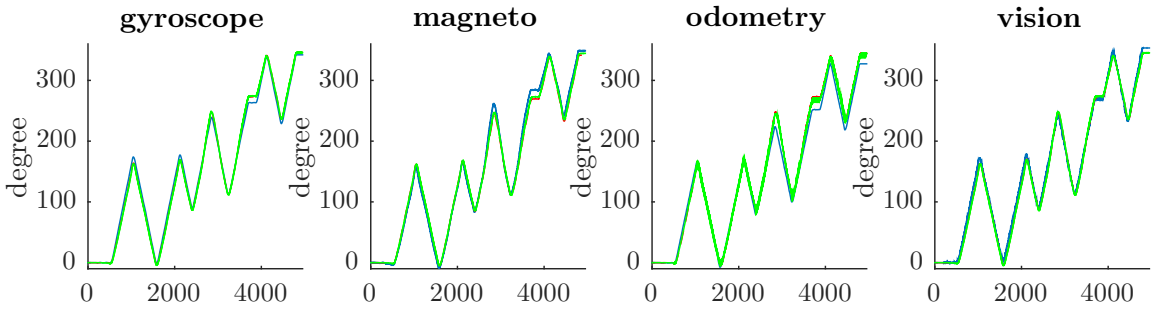


Figure 5.2: Covariance intersection heading result together with Kalman filter and sensor signals

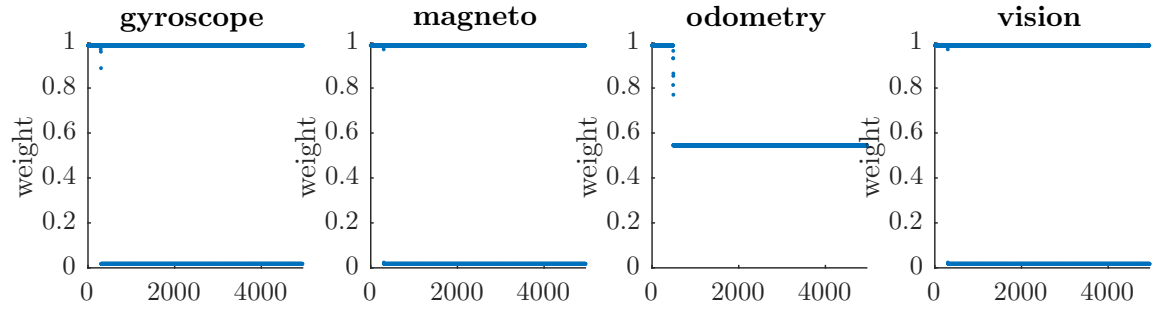


Figure 5.3: Covariance intersection weights heading

Figure 5.4 shows the position estimation results of the vision and odometry node. The vision nodes position estimate can be expected to be very accurate, because it is calculated from a camera mounted above the moving robot. The odometry heading data already suffers from the aforementioned error accumulation. In addition the round-of errors introduced by the body-fixed to global position transformation cause further inaccuracy. In total the odometry position estimation can be expected to be quite inaccurate. This can be expressed by an increased measurement noise R_k, which causes the fusion network to reject the odometry estimates continuously (cf. Figure 5.5). The global position estimation is clearly dominated by the vision estimate, which can be seen in Figure 5.4.

In terms of performance, the sensor fusion is a lot slower than pure Kalman filtering without preprocessing. This is mainly caused by the Covariance Intersection nodes which need to minimize the trace of M. The execution time for heading is $T_{10} \approx 6s$ while the execution time for position is even longer $T_{10} \approx 16s$. For the 2D position, the execution time scales significantly with the maximum allowed number of iterations in Brents method. Simulation results from 0 to 400 allowed iterations are plotted in Figure 5.6. While Brents method levels to a constant execution time for the 1D heading data, for the 2D position data it grows linearly. Tests show that this probably caused by numerical problems in the inversion routines.

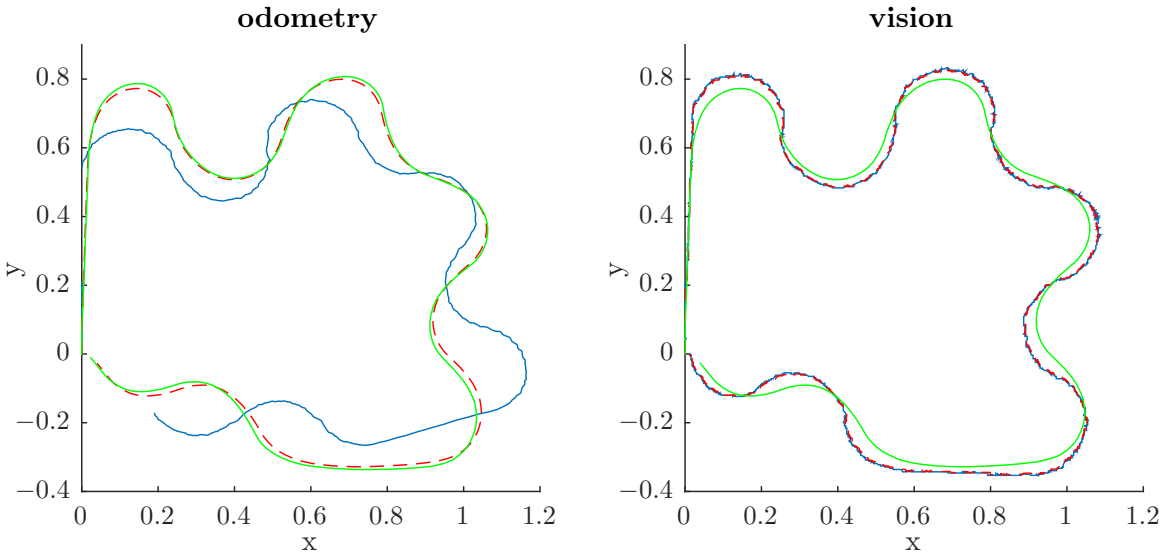


Figure 5.4: Covariance intersection position result(green) together with Kalman filter(red dashed) and sensor signals(blue)

5.2 Sensor Fusion Network Results

The implementation was tested with experimentally gained sensory data. The sensor data consists of 4952 samples acquired over the timespan of 183 seconds at 25 Hz. The same data was used for testing the competing implementation as well as the reference implementation. All measurements are made with the message-passing based implementation described in section 3.2. The architecture requires 491 ms to process the heading information and 392 ms to process position information. Since both tasks are processed at the same time the implementation requires 491 ms to work the provided data set. The difference in runtime for heading- and position estimation is explained in section 6.2. The result of heading estimation is displayed in Figure 5.9. The combined result of x-position and y-position estimation is shown in Figure 5.8.

5.3 MATLAB implementation and C port

For completeness the runtime measurements of the reference MATLAB implementation and the C port are mentioned here. The reference implementation requires 930 seconds to process the test data set [1]. The drop in C implementation reduced the netto runtime to 31 seconds. Tests of the C implementation on a 64-core server with OpenMP resulted in inferior results which is explained by the additional overhead of the multiprocessor structure.

[1]The report of the C implementation stated 5030 seconds for the reference implementation but this includes all overhead.

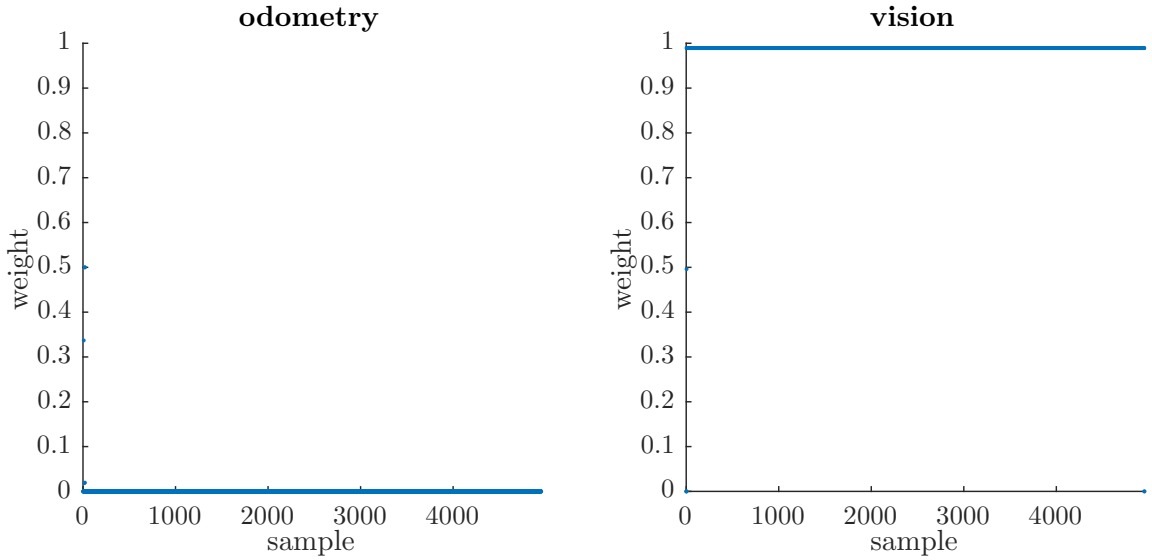


Figure 5.5: Covariance intersection weights position

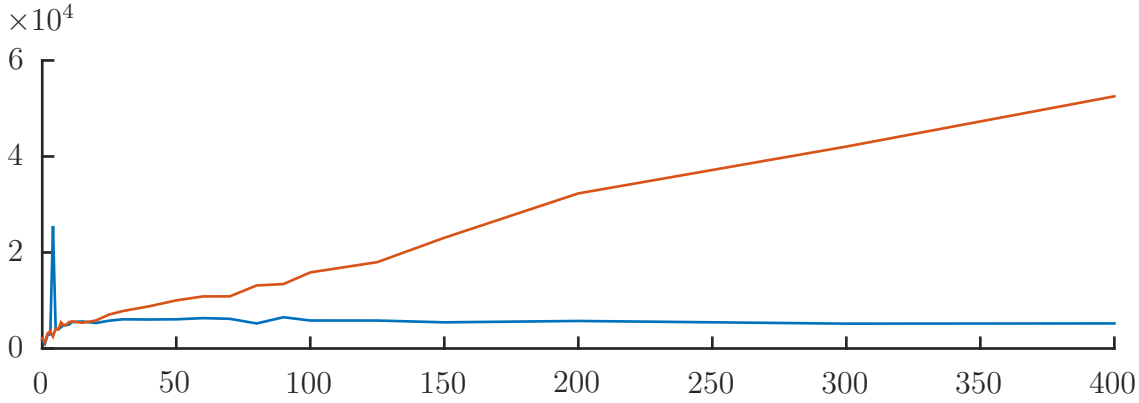


Figure 5.6: Execution time with respect to maximum allowed iterations in Brents method (blue: heading, red: position)

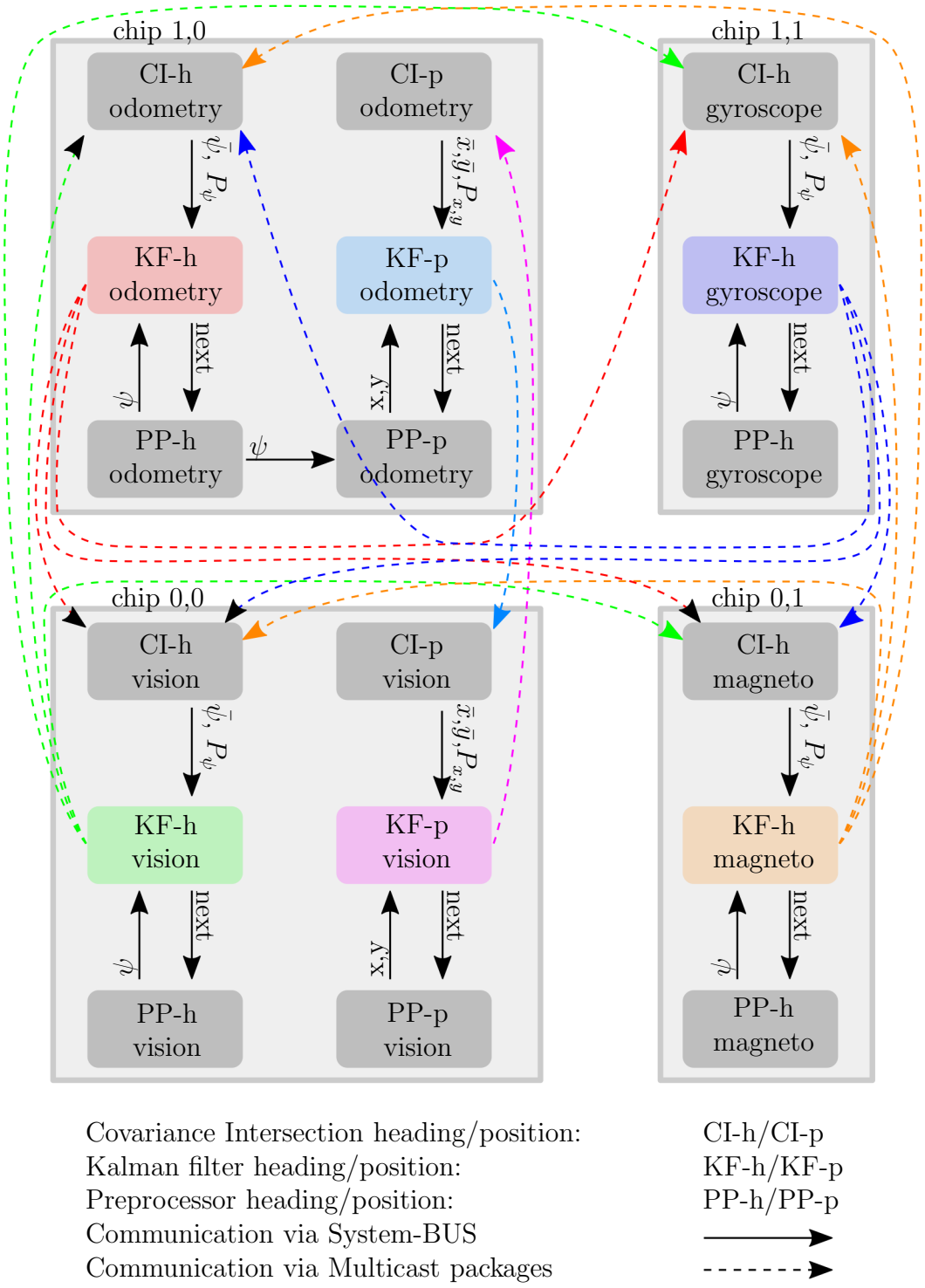


Figure 5.7: Infrastructure of distributed fusion network on spiNNaker

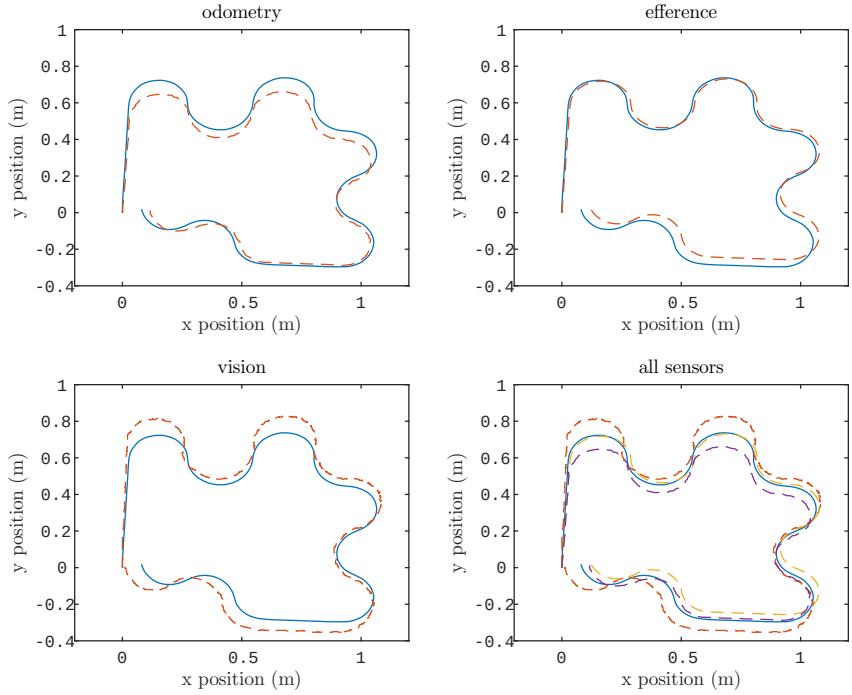


Figure 5.8: Position estimation of the network (blue) and respective sensor inputs (dashed).

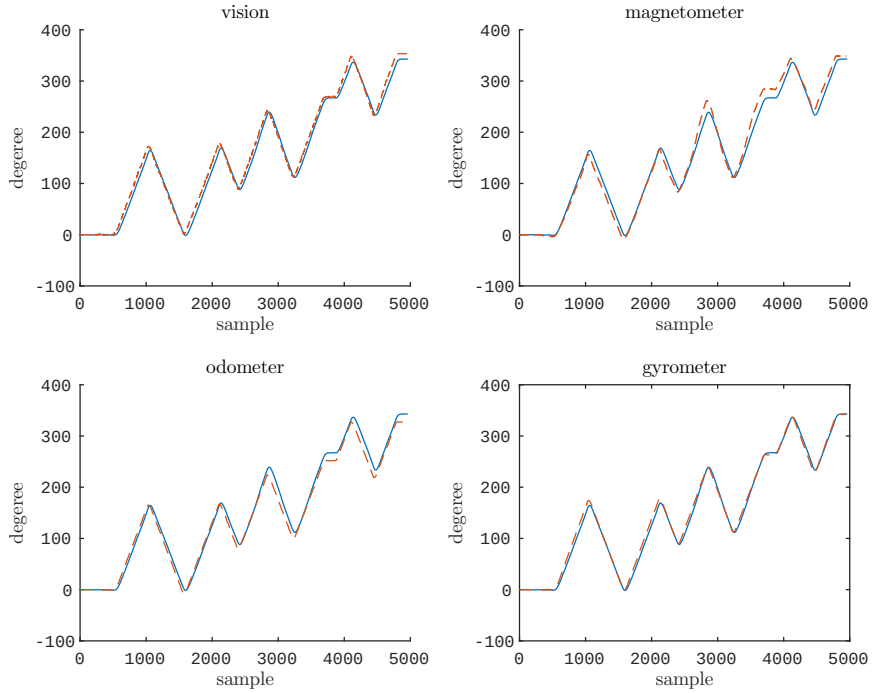


Figure 5.9: Heading estimation of the network (blue) and respective sensor inputs (red dashed).

Chapter 6

Conclusion

6.1 Distributed Fusion Network

This report documents the design and implementation of a distributed sensor fusion network which runs on the experimental platform spiNNaker. It uses a zeroth-order Kalman filter in connection with Covariance Intersection to fuse the heading and global position estimates from four different sensors. The raw sensor preprocessing is also implemented on the platform. The implementation is modular, enabling the usage of more sensors. The implementation exploits given hardware features, such as the asynchronous multicast communication among the cores and chips. This work shows that the spiNNaker platform can be used for general purpose calculations and that even complex algorithms can be implemented successfully.
The heading sensor fusion takes around 6s for 5000 samples taken at 25 Hz, while the sensor fusion of position data takes around 15s. The whole maneuver takes the robot approx. 198s, such that real-time sensor fusion is possible.
The network could easily be made fault-tolerant. In case of a defective camera the network can still use odometry data for a, at least course, position estimation. For this to be possible, there has to be some mechanism that detects inconsistency in sensory data. This can be done via Covariance Union [Uhl03], which could be done in future work.

6.2 Conclusions on the Sensor Fusion Network

The networks implementation on the SpiNNaker hardware uses only half of the cores available which in turn allows for doubling the number of sensor inputs without any noticeable penalty in hardware requirements or runtime. The runtime depends on the slowest map type. In this implementation base maps are the slowes map map type, because they have the highest number of inter-map connections which results in an increased number of value update calculations.
The parallel hardware which mirrors the suggested networks architecture in combi-

	MATLAB	C	SpiNN. network	Kalman	CovIn
CPU	3 GHz	3 GHz	0.2 GHz	0.2 GHz	0.2 Ghz
cores used	1	1	30	4	18
CPU total Wattage [a]	125 W	125 W	5 W	5 W	5 W
memory	4GB	4GB	128 MB	128 MB	128 MB
runtime	930 s	31 s	49 s	19 ms	15 s
files/lines of code	4/1796	39/6056	10/1301	10/884	11/1070
expandable w/o loss [b]	No	No	Yes	Yes	Yes

[a] for MATLAB and C the wattage is based on AMD Phenom II X4 945

[b] this only holds as long as < 64 cores are used

Table 6.1: The runtime is based on the datas 4952 samples, where each sample is represented to the network 100 times for relaxation.

nation with event-based programming form a viable solution for real-time application. Additionally the low power consumption and form factor make it very suitable for mobile application.

6.3 Comparison of implementations

The network is slower that the Kalman based solution but beats it in fault-tolerance and robustness since no knowledge about the systems physical properties is required. It is faster than the MATLAB implementation and in the same order of runtime of the C implementation with a fraction of power consumption.
A comprehensive comparison can be see in Table 6.1.

List of Figures

Bibliography

[AC14] Cristian Axenie and Jorg Conradt. Cortically inspired sensor fusion network for mobile robot egomotion estimation. *Robotics and Autonomous Systems*, (0):–, 2014. URL: http://www.sciencedirect.com/science/article/pii/S0921889014003133, doi:http://dx.doi.org/10.1016/j.robot.2014.11.019.

[CM01] Chee-Yee Chong and Shozo Mori. Convex combination and covariance intersection algorithms in distributed fusion. In *Proc. of the 4th International Conference on Information Fusion*, 2001.

[DL13] Christian Denk, Francisco Llobet-Blandino, Francesco Galluppi, Luis A. Plana, Steve Furber, and Jorg Conradt. Real-Time Interface Board for Closed-Loop Robotic Tasks on the SpiNNaker Neural Computing System, International Conf. on Artificial Neural Networks (ICANN), p. 467-74, Sofia, Bulgaria, 2013.

[IFA97] IFAC and American automatic control council. *A non-divergent estimation algorithm in the presence of unknown correlations*, IEEE catalog, 15, Albuquerque, NM US, 1997. IEEE computer society press. URL: http://opac.inria.fr/record=b1113303.

[LCK+97] M. E. Liggins, C. Y. Chong, I. Kadar, M. G. Alford, V. Vannicola, and S. Thomopoulos. Distributed fusion architectures and algorithms for target tracking. *Proceedings of the IEEE*, 85(1):95–107, 1997.

[Pre07] William Press. *Numerical recipes : the art of scientific computing*. Cam-bridge University Press, Cambridge, UK New York, 2007.

[STL09] J. Sequeira, A. Tsourdos, and S. Lazarus. Robust covariance estimation in sensor data fusion. *Safety, Security & Rescue Robotics (SSRR), 2009 IEEE International Workshop on date, 3-6 Nov. 2009*, 2009.

[Uhl03] J. Uhlmann. Covariance consistency methods for fault-tolerant distributed data fusion. *Information Fusion*, 4(3):201–215, September 2003. doi:10.1016/s1566-2535(03)00036-8.

[WA14] Weikersdorfer D., Conradt J. (2012), Event-based Particle Filtering for Robot Self-Localization, Proceedings of the IEEE International Conference on Robotics and Biomimetics (IEEE-ROBIO), pages: 866-870, Guangzhou, China.

[WC12] Weikersdorfer D., Conradt J. (2012), Event-based Particle Filtering for Robot Self-Localization, Proceedings of the IEEE International Conference on Robotics and Biomimetics (IEEE-ROBIO), pages: 866-870, Guangzhou, China.

[WH13] Weikersdorfer D., Hoffmann R., and Conradt J. (2013) Simultaneous Localization and Mapping for event-based Vision Systems, International Conference on Computer Vision Systems (ICVS), p. 133-142, St. Petersburg, Russia.

[Zar09] Paul Zarchan. *Fundamentals of Kalman filtering a practical approach*. American Institute of Aeronautics and Astronautics, Reston, Va, 2009.

License