"The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking"

Kalbusch, Sébastien ; Verpoten, Vincent

ABSTRACT

Classical sensor fusion approaches require to work directly with the hardware and involve a lot of low-level programming that is not suited for reliable and user-friendly sensor fusion for Internet of Things (IoT) applications. In this master thesis, we have developed Hera, a Kalman filter-based sensor fusion framework for Erlang which offers a high-level approach for dynamic, soft real-time, and fault-tolerant sensor fusion directly at the edge of an IoT network. We use the GRiSP-Base board, a low-cost platform specially designed for Erlang and to ease development by avoiding soldering or dropping down to C. We emphasise on the importance of performing all the computations directly at the sensor-equipped devices themselves, completely removing the cloud necessity. With Hera, the implementation effort is significantly reduced which makes it an excellent candidate for IoT prototyping and education in the field of sensor fusion. We explain the basis of inertial navigation and tracking, and show that with the use of Erlang, GRiSP, and Hera, it is possible to perform simple sensor fusion for position and orientation tracking at a high-level of abstraction. From a fault-tolerance analysis based on fault-injection, we show that Hera gives the strong guarantee to do sensor fusion as long as one GRiSP board is alive. In a first phase, we experiment with Hera to show how we can build a sensor fusion model for a position tracking application and illustrate the benefits of sensor fusion as we add more sensors or increase the model complexity. In a second phase, we attain the limit o...

CITE THIS VERSION

UCLouvain

epl

École polytechnique de Louvain

The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking

Sensor fusion directly at the edge with Erlang and GRiSP

Authors: **Sébastien KALBUSCH, Vincent VERPOTEN**
Supervisor: **Peter VAN ROY**
Readers: **Ramin SADRE, Peer STRITZINGER**
Academic year 2020–2021
Master [120] in Computer Science and Engineering
Master [120] in Computer Science

Abstract

Classical sensor fusion approaches require to work directly with the hardware and involve a lot of low-level programming that is not suited for reliable and user-friendly sensor fusion for Internet of Things (IoT) applications.

In this master thesis, we have developed Hera, a Kalman filter-based sensor fusion framework for Erlang which offers a high-level approach for dynamic, soft real-time, and fault-tolerant sensor fusion directly at the edge of an IoT network. We use the GRiSP-Base board, a low-cost platform specially designed for Erlang and to ease development by avoiding soldering or dropping down to C. We emphasise on the importance of performing all the computations directly at the sensor-equipped devices themselves, completely removing the cloud necessity. With Hera, the implementation effort is significantly reduced which makes it an excellent candidate for IoT prototyping and education in the field of sensor fusion.

We explain the basis of inertial navigation and tracking, and show that with the use of Erlang, GRiSP, and Hera, it is possible to perform simple sensor fusion for position and orientation tracking at a high-level of abstraction. From a fault-tolerance analysis based on fault-injection, we show that Hera gives the strong guarantee to do sensor fusion as long as one GRiSP board is alive.

In a first phase, we experiment with Hera to show how we can build a sensor fusion model for a position tracking application and illustrate the benefits of sensor fusion as we add more sensors or increase the model complexity.

In a second phase, we attain the limit of the current system by tackling a more challenging 6 degrees of freedom (DOF) inertial measurement unit (IMU). We explain the theory and show surprisingly good results for the attitude and heading reference system (AHRS).

Finally, we give information about planned performance improvements, to address certain limitations, as well as possibilities for future work.

Acknowledgements

If we were able to complete this work it is partially because of the help we received from various people. We would like to express our gratitude to them.

First of all, we thank Peer Stritzinger, Mirjam Friesen and Adam Lindberg for the help they provided with the GRiSP platform. We have had a hard time installing the software to get started and we are thankful that they discovered an incompatibility with recent versions. We also want to thank Peer Stritzinger in particular, because a vast portion of our work is based on Kalman filters and it was his idea in the first place. He also helped us identify a small bug in the Pmod NAV driver.

Next, we thank Guillaume Neirinckx and Julien Bastin not only for their help to get started with our master thesis, but also and mainly for the good work they did. Indeed, the user manual and explanations in their master thesis were really useful. We could not have done this work without them since we have used Hera, an application originally built by them, as a foundation for our work.

Then, we would like to thank Vanessa Maons who gave us access to a private room for us to conduct our experiments as well as providing us a locker so that we could safely store our equipment. And, of course, we thank her for making herself highly available when we had to fetch our equipment.

I, Sébastien Kalbusch, would also like to thank my father for his help with the sonar supports.

Last but not least, we greatly thank our professor and supervisor, Peter Van Roy. We met him on a weekly basis and he was always very supportive. He also allowed us to purchase some equipment like battery packs or a toy train. The latter was his idea and we are very glad for it because the train allowed us to better analyse and understand our system with a stable experimental setup. We also express our gratitude to Peter Van Roy for the article about Hera that we wrote together. His participation has been a delight and we were not expecting such a high quality and constructive contribution. Finally, we are thankful towards our supervisor for having suggested the subject of sensor fusion and inertial navigation with GRiSP and Erlang. These topics and technologies have interested us deeply and made us extremely happy about our master thesis as a whole.

Contents

List of Figures

Chapter 1

Introduction

1.1 Sensor fusion and edge computing

Sensor fusion is the ability to combine information from multiple sensors to give a single coherent view of a real-world situation. Sensor fusion is a crucial ability for Internet of Things (IoT) applications. Yet, it is not trivial to provide it since it requires to do significant computation with sensor data in real time, preferably directly at the edge (at the sensor devices themselves). The IoT infrastructure consists of the growing set of small devices at the logical[1] edge of the Internet, farthest away from the cloud. IoT is a fast-growing part of the Internet infrastructure with a rapidly increasing computational power and functionality directly at the edge. In fact, IoT is growing significantly faster than the cloud at the current time [23] (13% per year versus 5% per year) so it is important for IoT devices to take over some of the computation previously done in the cloud. Moreover, delegating computation to the cloud sometimes comes with significant disadvantages. The cloud infrastructure is massive and expensive [2]. Computing at the extreme edge not only has the potential to reduce costs, but also to largely simplify the infrastructure. Indeed, going to cloud still requires a fair amount of work and increases the overall complexity of IoT projects. Additionally, certain domains, like tracking, require real-time computation in which the slightest delays decrease the accuracy. The edge-cloud connection has an unpredictable reliability and might even do down surprisingly often. Furthermore, the latency of the edge-cloud connection is not negligible (usually tens of [ms]). For such applications, the physical distance between the cloud infrastructure (data centers) and the sensors matters a lot. Therefore, it is essential to bring the computation to the extreme edge instead of simply using the edge devices for data gathering.

[1] In terms of structure and not physical distance.

[2] Renting a server is not cheap.

In this master thesis, we present an approach for fault-tolerant sensor fusion in Erlang with GRiSP to advance the state of the art in low-cost sensor fusion directly at the edge, and to give access to an efficient sensor fusion platform to all interested parties. The hardware consists of a network of GRiSP boards [3], each of which can host Pmod modules [5] as sensors or actuators. Our software consists of an extensible open-source application intended to enable low-cost and fault-tolerant sensor fusion prototyping in IoT applications for education or product development. It was developed in the context of a long-term research in IoT applications at UCLouvain in collaboration with Stritzinger GmbH[4]. This research started in the European Horizon 2020 LightKone project [15] and continued with multiple master thesis including [21] which was used as a starting point for this work.

1.2 Inertial navigation and tracking

Inertial navigation is a self-contained navigation technique that consists of tracking the position and orientation of an object relative to a known starting point with inertial sensors (accelerometers and gyroscopes). A system that only provides the orientation is refereed to as an attitude and heading reference system (AHRS).

Expensive and accurate inertial sensors are used in fields like aeronautics, but microelectromechanical systems (MEMS) sensors are becoming more and more present in our daily environment. These sensors are not nearly as accurate and suffer terribly from drift. Modern inertial navigation systems (INS) constantly correct the position and orientation with absolute information from other sources like the global positioning system (GPS) in a process called sensor fusion. There exist multiple sensor fusion techniques [2]. In this master thesis, we have explored the feasibility of inertial navigation and tracking with a Kalman filter-based [30] sensor fusion engine running directly at the edge of an IoT network.

1.3 Results and contributions

We present a successful approach for fault-tolerant sensor fusion in IoT applications. We show the feasibility and analyse the quality of simple position and orientation tracking directly at the edge. We also show limitations on the current hardware and software, and give perspectives for future work.

The contributions are:

1. Hera, an open-source framework for fault-tolerant sensor fusion running on a

[3] `https://www.grisp.org`

[4] `https://www.stritzinger.com`

network of GRiSP boards and based on Kalman filters. Our complete system features:

- A soft real-time sensor fusion engine based on Kalman filters accepting asynchronous measurements from sensors with examples for position and orientation estimation using four sensor types, namely accelerometer, gyroscope, magnetometer, and sonar.
- A dynamic, distributed, and fault-tolerant architecture with the guarantee to continuously do sensor fusion as long as one GRiSP board in the network is running. If a sensor or a board fails then the software continues to work, with degraded accuracy.
- A modular visualisation tool called LiveView built with GNU Octave [7].

2. Evaluation of the fault tolerance of the framework by performing fault injection and observing how the framework responds. It will run correctly as long as one GRiSP board is operational.

3. Evaluation of the abilities and limitations of our approach with an experimental model that shows how the sensor fusion quality improves as we add more sensors.

4. Presentation and validation of an AHRS able to update its physical model at a rate of 3.75 [Hz], running completely on a GRiSP board.

5. Correction of a small bug in the Pmod NAV driver (see appendix E).

1.4 Related work

1.4.1 Low-cost sensor fusion in IoT

We compare our system to other low-cost systems based on platforms like Raspberry Pi (R-Pi) or Arduino. The purpose of Hera is to provide a "high-level" approach for sensor fusion projects without cumbersome hardware wiring or driver development and without having a complex cloud infrastructure.

In [3], a R-Pi is used as a gateway for a blood pressure monitoring application and the data is sent to the cloud [5] instead of processing the data directly at the edge as we do. In [28], two Arduino and sensors are used to monitor a refrigerator. The two Arduino are interconnected to exchange sensor information

[5] In that research, the domain is restricted to localhost.

via I2C communications. One of them uses a WIFI interface to send data to Dropbox, used as a cloud storage.

This type of system is not resilient to failure and also reveals that Arduino requires to work close to the hardware which tends to increase the overall complexity of the system. In our case, the use of GRiSP with the Pmod interfaces and Erlang greatly facilitate these steps.

A miniature resilient and low-cost data repository on R-Pi is demonstrated in [27]. The data is harvested by sensors connected to the R-Pi and stored permanently on the SD card. This paper shows that building a resilient edge data repository is difficult. With Hera, we store the data in RAM and our system requires that at least one GRiSP board remains alive.

1.4.2 Sensor networks

A sensor network is a group of sensors sending their data to a central location for storage or computation. We compare this field with our system, where sensor entities interact directly with each other.

In [19], a multi-sensor data fusion structure is used to help in early heart disease prediction by fusing data coming from different wearable sensors with machine learning methods. The data is stored in a cloud server for backup, but all the machine learning computations are done in a fog computing environment. The fog computing environment is a decentralized computing infrastructure located between the data source and the cloud. The structure of the project is divided into three distinct parts: the sensors, the fog computing environment, and the cloud, each part having its own role. In our structure, all of these roles (sensors, computation, and storage) are fulfilled by the sensors equipped devices located at the edge of the network. The GRiSP boards are more powerful than simple sensors and exchange their information directly with each other. This reduces the complexity of the whole system.

The theoretical study [14] proposes a method for single and multi target tracking in large-scale sensor networks based on maximum entropy fuzzy clustering. The paper only shows simulations, but could be a good starting point for the problem of multi-target tracking which we did not address. Their approach is split in two parts: data association and tracking at the sensor-level and a sensor selection based on fuzzy membership at a global-level. The advantage of this approach is to share the computational load between small edge devices and a more powerful centralized computer. This kind of architecture offers the possibility to use more sophisticated algorithms in large-scale environments.

1.4.3 Inertial navigation

Inertial navigation is a topic that brought a lot of research and solutions in the scientific world. We learned the principles in [32] and tried to apply these techniques at the extreme edge of an IoT network. In this technical report, we also learnt about aided INS with Kalman filter. We experimented this method for linear motion tracking with an accelerometer and a sonar.

[18] is another example of aided inertial navigation that presents an extended Kalman filter-based framework for tracking satellites using Doppler measurements drawn from their signals to help the INS of a vehicle. In [13], we discovered that the Extended Kalman filter (EKF) is able to model non-linear systems and so, we also performed more advanced sensor fusion with it.

There are other Kalman filter versions like the Cubature Kalman filter (CKF) [1] or the unscented Kalman filter [29] for highly-non linear problems.

1.4.4 Attitude and Heading Reference System

An attitude and heading reference system (AHRS) allows to track the orientation of an object in 3 dimensions with a magnetometer, an accelerometer, and/or a gyroscope. In an IoT environment, this task is not easy because the data comes from MEMS sensors with limited accuracy and the hardware is often running at a low frequency. [20] emphases the difficulty to match an estimated orientation with the real one via an IoT structure, based on the Shimmer3 IMU[6], and concludes that a precise estimation of a rigid body angle may not be achieved using wearable motion trackers in daily activities or clinical applications. We, on the other hand, have shown that a simple AHRS can already give substantial results.

[25] presents a solution to increase the accuracy of a MEMS AHRS by dynamically adapting the measurement noise covariance according to the measured acceleration. In our case, we have kept a constant noise measurement covariance, but it would be trivial to implement it with Hera.

In [11], the authors present a quaternion-based Kalman filter for which the input has been previously computed by a two-step geometrically-intuitive correction (TGIC). The data of an accelerometer and a magnetometer goes through the TGIC before joining a gyroscope in the Kalman filter at a frequency of 1 [kHz]. These two layers reduce the magnetic disturbances as well as the effect of linear acceleration. Our AHRS is inspired from it, but we have not used the TGIC because it requires a too high update frequency.

[33] also uses the Shimmer3 IMU, a relatively[7] low-cost device made for body worn applications with 6 DOF. The authors propose a new orientation estimation

[6] `http://www.shimmersensing.com//products/shimmer3-development-kit`

[7] More than twice as expensive as a GRiSP board.

method to fight magnetic distortions. This is a good example because they used a particle filter and an adaptive cost function instead of the more popular Kalman filter.

Chapter 2

Resources and background

2.1 The GRiSP environment

2.1.1 The GRiSP-Base board

The GRiSP-Base board (Fig.2.1) is an embedded device produced by *Peer Stritzinger GMBH* with multiple advantages, from a software and a hardware point of view.

The GRiSP platform consists of a single board with built-in WiFi, USB connectivity and six sockets for Digilent Pmod sensor and actuator modules. This Pmod-compatibility is a great benefit for IoT device like GRiSP, expanding the range of interesting projects thanks to a large selection of sensors and actuators. Its network properties allow to connect multiple boards via WiFi or Adhoc network to easily exchange information.

The board contains a full Erlang/OTP (see section 2.2) platform running on a RTEMS real-time layer, a hard real-time embedded "operating system". This property makes the use of Erlang straightforward, and avoids us to drop down to C unlike on a lot of other IoT devices. Furthermore, GRiSP provides Erlang drivers for the most common Pmod modules, which facilitates their integration inside project code. Overall, GRiSP is a fantastic platform for "out of the box" IoT developments.

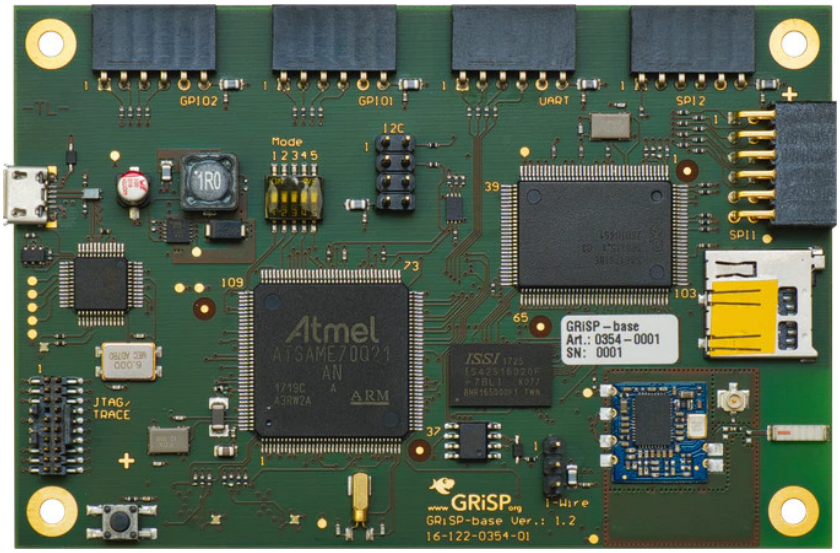

Figure 2.1: The GRiSP-Base board

The following specifications come directly from the GRiSP website[1].

The **CPU** of the GRiSP is an Atmel SAM V71, with a ARM Cortex M7 as core processor and runs at a clock frequency of 300 [MHz].

The **Internal Memory** is composed of 2048 [KB] of Flash memory and 384 [KB] of SRAM.

The **External Memory** is made up of 64 [MB] of SDRAM and the given space of the standard MicroSD card plugged in the MicroSD socket.

The **Network** uses a WiFi antenna to provide communication via the 802.11b/g/n protocol.

The **I/O Interface** presents one Dallas 1-Wire via 3-pin connector, one Digilent Pmod compatible I2C interface, two Digilent Pmod Type 1 interfaces (GPIO), one Digilent Pmod Type 2 interface (SPI), one Digilent Pmod Type 2A interface (expanded SPI with interrupts) and one Digilent Pmod Type 4 interface (UART). There is a serial port for console (Erlang Shell) and a JTAG debugger via Micro USB. The power supply is also provided via this Micro USB.

[1] `https://www.grisp.org/specs`

2.1.2 Digilent Pmod sensors

Digilent Pmod stands for Periphical Modules and provides sensitive signal via pin connectors to microcontroller boards like GRiSP. For this project, we use two of them : the Pmod MAXSONAR [2] and the Pmod NAV[3].

Pmod MAXSONAR This Pmod is an ultrasonic range finder allowing to detect objects and people at a distance from 15 [cm] to 6.5 [m] with an accuracy of 2.5 [cm] (Fig.2.2).

Figure 2.2: Pmod MAXSONAR

Pmod NAV This Pmod provides a 3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer, and a digital barometer. All this information in one Pmod makes this module very interesting and complete for tracking applications (Fig.2.3).

[2] `https://reference.digilentinc.com/reference/pmod/pmodmaxsonar/start`
[3] `https://reference.digilentinc.com/reference/pmod/pmodnav/start`

Figure 2.3: Pmod NAV

2.2 Erlang/OTP

One of the features that makes the GRiSP platform desirable for IoT applications is its support for Erlang/OTP.

Erlang is a programming language designed for building massively scalable and highly available systems. It is a dynamically typed functional language with concurrent processes that communicates with message passing. The Open Telecom Platform (OTP) is a set of Erlang libraries and design principles for building Erlang applications. It provides, among other things, support for fault-tolerant systems and distributed computing.

It is largely because of the abilities provided by GRiSP and Erlang/OTP that this work was possible.

2.3 Kalman filter

The Kalman filter is a well known data fusion method for state estimation with multiple benefits. It is simple to implement, requires low computational power, can handle noisy data and can be used for a variety of problems. We implemented two general variations of Kalman filters and in this section, we briefly explain how they work. The generic discrete Kalman filters we propose are slightly simpler than those in [13] because we do not use any control input.

The linear Kalman filter estimates the normally distributed random state of a system that can be modeled by a set of linear equations. It is composed of two independent phases called prediction and update.

The prediction phase allows to estimate the state of the system $\hat{x}_{k+1}^-$ from the previous known estimated $\hat{x}_k$ and its physical model F_k (2.1). In addition, it will produce the *a priori* state estimate covariance matrix P_{k+1}^- from the so-called process covariance Q_k and *a posteriori* state estimate covariance P_k matrices (2.2).

$$\hat{x}_{k+1}^- = F_k \hat{x}_k \tag{2.1}$$
$$P_{k+1}^- = F_k P_k F_k^T + Q_k \tag{2.2}$$

The update phase is used to correct the estimated state $\hat{x}_{k+1}^-$ with the observation z_k coming from, typically, sensors. A linear relation H_k, called the observation model, allows to compare the two with one another. Using the state covariance P_{k+1}^- and the observation covariance R_k, we can compute the gain (2.3) and the updated state $\hat{x}_{k+1}$ (2.4). Of course, we also update the state covariance in (2.5).

$$K = P_{k+1}^- H_k^T \left(H_k P_{k+1}^- H_k^T + R_k \right)^{-1} \tag{2.3}$$
$$\hat{x}_{k+1} = \hat{x}_{k+1}^- + K \left(z_k - H_k \hat{x}_{k+1}^- \right) \tag{2.4}$$
$$P_k = \left(I - K H_k \right) P_{k+1}^- \tag{2.5}$$

A nice property to emphasize is that the vectors and matrices can change between two iterations making the Kalman filter an excellent choice for asynchronous data fusion (see section 4.4). Moreover, there is no obligation to perform both the prediction and update phases every time. If needed, it is absolutely possible to call one of them multiple times in a row.

Fortunately, we are not limited to model only linear models. The extended Kalman filter (EKF) can work directly with a differentiable state transition function f_k to predict the next state (2.6) and its Jacobian evaluated around the estimate J_{f_k} to compute the state covariance by linearization (2.7).

$$\hat{x}_{k+1}^- = f_k \left(\hat{x}_k \right) \tag{2.6}$$
$$P_{k+1}^- = J_{f_k} P_k J_{f_k}^T + Q_k \tag{2.7}$$

Similarly, from the differentiable function h_k representing the observation model, we can perform the update (2.9) and from its Jacobian evaluated around the estimate J_{h_k} we compute the gain (2.8) and updated covariance (2.10).

$$K = P_{k+1}^- J_{h_k}^T \left(J_{h_k} P_{k+1}^- J_{h_k}^T + R_k \right)^{-1} \tag{2.8}$$
$$\hat{x}_{k+1} = \hat{x}_{k+1}^- + K \left(z_k - h_k \left(\hat{x}_{k+1}^- \right) \right) \tag{2.9}$$
$$P_k = \left(I - K J_{h_k} \right) P_{k+1}^- \tag{2.10}$$

2.4 Orientation representation

In this section, we give some background information about orientation representations that we use throughout this document.

2.4.1 Euler angles

One of the most popular way of representing an orientation is through Euler angles: θ, ϕ, and ψ (also known as pitch, roll and yaw). Each of them gives the rotation with respect to one axis and the complete orientation is obtained by composition. We can rotate further by addition: $\theta_1 = \theta_0 + \delta\theta$. However, rotations are not commutative and so we should always combine the angles in the same order. There are three well-known issues with Euler angles.

Ambiguities: Euler angles are ambiguous which means that the same orientation may be represented with different Euler angles. This is a very important issue when it comes to predict the evolution of the orientation. We have generated Fig.2.4 with the help of an online visualization tool [26] to show an example of this phenomenon. Imagine the orientation vector (Yaw, Pitch, Roll) in degrees is $\begin{pmatrix} 0 & 75 & 0 \end{pmatrix}$ (top left) and the body is rotating at a rate of $\begin{pmatrix} 0 & 30 & 0 \end{pmatrix}$ [°/s], we predict that after 1 [s], the orientation vector should be $\begin{pmatrix} 0 & 105 & 0 \end{pmatrix}$ (top right). However, we might measure $\begin{pmatrix} 180 & 75 & 180 \end{pmatrix}$. To understand why, we apply the rotation one angle at a time (bottom left to bottom right). As you can see, they are equivalent and it makes it difficult to compare a prediction with an observation. As we shall see later in this section, the solution for this problem is to use a non-ambiguous representation.

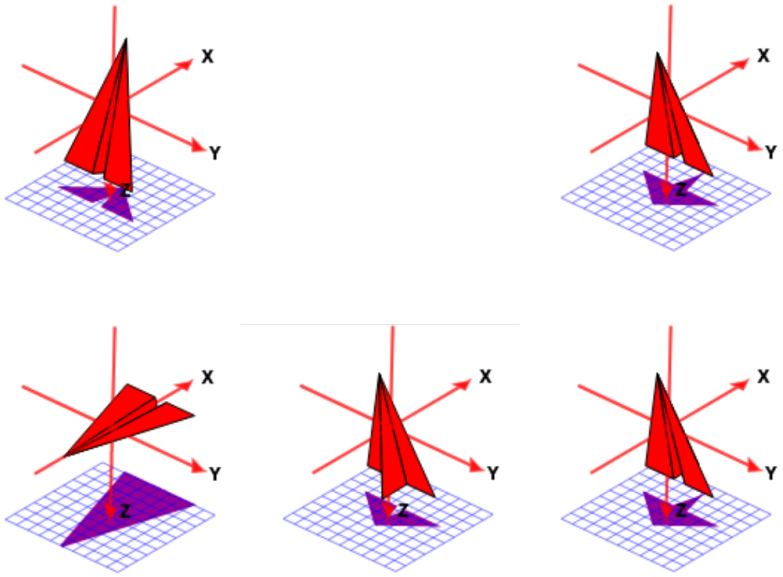

Figure 2.4: Ambiguities of Euler angles

The wrapping effect: A Euler angle is defined modulo 2π. Again, this problem makes it difficult to compare two angles because a non-null difference does not mean that the angles are different. It is easy to compare them by applying the modulo 2π first, but that is not enough. Assume that an angles ranges from $-180°$ to $180°$. What is the difference between $-170°$ and $170°$? It could be $340°$, but also $20°$ (Fig.2.5). If we are interested in the shortest path between to angles, some care must be taken when we compare the angles like, for example, inside of a Kalman filter.

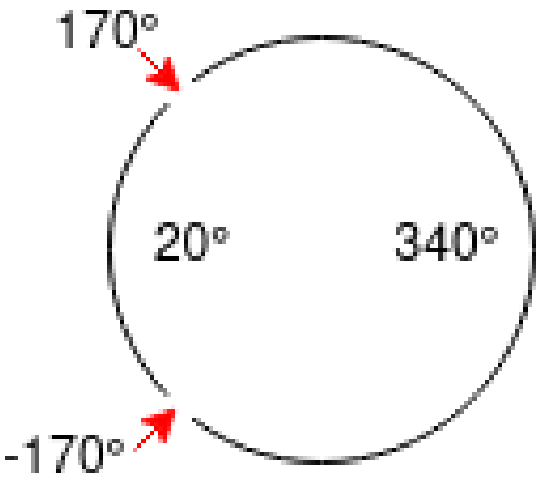

Figure 2.5: The wrapping effect

Gimbal lock: Euler angles are subject to gimbal lock which means that for certain values, we may lose one degree of freedom. This will happen when two gimbals are aligned. Of course, Euler angles are just numbers, they are not physical gimbals. However, they can be though of as actual gimbals because of the way they build on each other. The left scenario of Fig.2.6, from [36], shows 3 gimbals and the location of the axles are indicated by arrows. As you can see on the right scenario, if we increase the angle of the red gimbal by 90°, the blue and green gimbals are locked together because their axles are aligned. Indeed, turning the blue or the green gimbals only affects one dimension, not two. This problem is well-known, but is only an issue at certain values or if we want to interpolate Euler angles. The best solution, to avoid it, is simply to use an other representation.

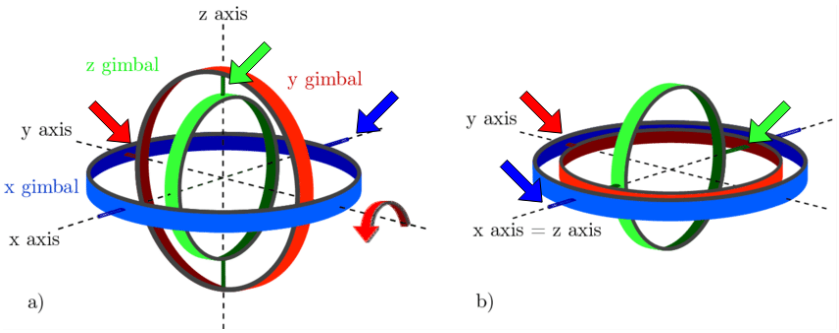

Figure 2.6: Gimbal lock: the blue and green axles are aligned

2.4.2 Rotation matrix

Any orientation can be represented by a rotation matrix also known as direct cosine matrix. A rotation by zero is represented by the identity matrix. To rotate further, we can multiple a matrix by another, but this operation is not commutative. Multiplying from the left side or the right side will change the sign of the rotation. A nice property of rotation matrices is that they are orthogonal and so, the inverse equals the transpose. We can obtain the rotation matrix from Euler angles by multiplying $R_x(\theta)$, $R_y(\phi)$, and $R_z(\psi)$ (2.11). However, the order must be preserved for all operations otherwise we would end-up with different matrices.

$$
\begin{aligned}
R_x(\theta) &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \\
R_y(\theta) &= \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \\
R_z(\theta) &= \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{2.11}
$$

It is also possible to find the Euler angles from a rotation matrix [9], but the solution is not always unique because of gimbal lock.

2.4.3 Quaternions

Quaternions are four-dimensional numbers made of one real number and three imaginary numbers and they give another way of representing orientation. They are a super-set of the complex numbers and they follow relation 2.12. A rotation by zero is represented by a unit quaternion without imaginary part. It is possible to rotate a quaternion by another with the Hamiltonian product. The only important thing to know is that this multiplication is not commutative.

Unit quaternions are great for rotations because they are not ambiguous and they require less computation than rotation matrix. Indeed when we work with rotation matrix it might be required to restore the anti-symmetry and to re-normalize because numerical computations often introduce errors which accumulate over time. Normalizing a quaternion is similar to normalizing a vector, once it has been written in vector form (2.13).

$$
i^2 = j^2 = ijk = -1 \tag{2.12}
$$

$$q = a + bi + cj + dk \Rightarrow q = \begin{pmatrix} a & b & c & d \end{pmatrix}^T \tag{2.13}$$

Because this representation was mostly unfamiliar to us, we decided to use them as little as possible and only to overcome the issues of Euler angles and rotation matrix that we have discussed in this section. Most of the time, we will convert a quaternion into a rotation matrix (2.14) [4] and vice-versa (2.15) [10]. Note that there are multiple ways of making this conversion.

$$R = \begin{pmatrix} 2(q_1q_1 + q_2q_2) - 1 & 2(q_2q_3 - q_1q_4) & 2(q_2q_4 + q_1q_3) \\ 2(q_2q_3 + q_1q_4) & 2(q_1q_1 + q_3q_3) - 1 & 2(q_3q_4 - q_1q_2) \\ 2(q_2q_4 - q_1q_3) & 2(q_3q_4 + q_1q_2) & 2(q_1q_1 + q_4q_4) - 1 \end{pmatrix} \tag{2.14}$$

$$\begin{aligned} q_1^2 &= \frac{1}{4}(1 + R_{11} + R_{22} + R_{33}) \\ q &= \frac{1}{4q_1} \begin{pmatrix} 4q_1^2 \\ R_{32} - R_{23} \\ R_{13} - R_{31} \\ R_{21} - R_{12} \end{pmatrix} \end{aligned} \tag{2.15}$$

Another benefit of working with quaternions is that they are easy to interpolate. In the literature, this is refereed as spherical interpolation (SLERP) [8]. We will not explain how to it, as it goes beyond the scope of our master thesis, but since we will compare two quaternions q_1 and q_2 inside of a Kalman filter, it is good to understand what we are actually doing. Moreover, there are two possible paths between two quaternions: the short (less than 180°) and the long (more than 180°). In our case, with the Kalman filter, we are interested in the short path. It is possible to ensure that the Kalman filter will compute the short path by changing the sign of one quaternion such that $q_1 \cdot q_2 > 0$.

Chapter 3

An introduction to inertial navigation and tracking

3.1 Principle of inertial navigation

Inertial navigation is an autonomous technique that consist of estimating the position of a moving body from inertial sensors like an accelerometer or a gyroscope. In theory, it is possible to estimate the position by integrating the acceleration twice, and for the orientation, we must integrate the angular velocity once. This technique is known as Strapdown inertial navigation [32].

The derivative of a function f around t is given by (3.1) and illustrated by Fig.3.1 [31].

$$f'(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{3.1}$$

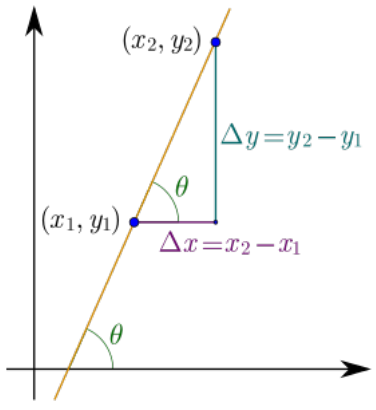

Figure 3.1: Slope of a linear function

As the acceleration is the derivative of the velocity and the velocity is the derivative of the position, we can write (3.2) and (3.3). Then, if we suppose Δt is small enough, we can isolate $v(t + \Delta t)$ and $p(t + \Delta t)$ which gives (3.4) and (3.5). The precision of this numerical integration depends on the value of Δt. In practice, this method is not accurate because errors quickly accumulate over time. We will demonstrate that in section 3.2.

$$a(t) = \lim_{\Delta t \to 0} \frac{v(t + \Delta t) - v(t)}{\Delta t} \tag{3.2}$$

$$v(t) = \lim_{\Delta t \to 0} \frac{p(t + \Delta t) - p(t)}{\Delta t} \tag{3.3}$$

$$v(t + \Delta t) = v(t) + a(t)\Delta t \tag{3.4}$$

$$p(t + \Delta t) = p(t) + v(t)\Delta t \tag{3.5}$$

It is also possible to perform the double integration analytically if we can make hypothesis on the acceleration. For instance, if the acceleration remains constant we can write 3.6.

$$\begin{aligned} p(t) &= \int_0^{\Delta t} a \, dt \\ &= p_0 + v_0 \Delta t + \frac{a}{2} \Delta t^2 \end{aligned} \tag{3.6}$$

3.2 Example with a MEMS accelerometer

In this section we try to apply the principle of inertial navigation with a real MEMS accelerometer (see section 2.1). However, before we can do anything with it, some pre-processing is required.

For the data that we show in this section, we have simply created a recursive function in Erlang in which we fetch data from the sensor driver and then apply our calculation before directly writing the result to the GRiSP SD card.

3.2.1 Extracting the linear acceleration

The raw output of a MEMS accelerometer cannot directly be used for integration because the output signal is subject to a bias and sensitivity. The bias is an offset and the sensitivity is a multiplicative factor (Fig.3.2). The sensitivity is provided by the sensor data-sheet and luckily for us, it is already applied in the pmod NAV

driver. Removing the bias can easily be done be averaging the noisy output signal when the sensor is not accelerating, but doing so will remove gravity.

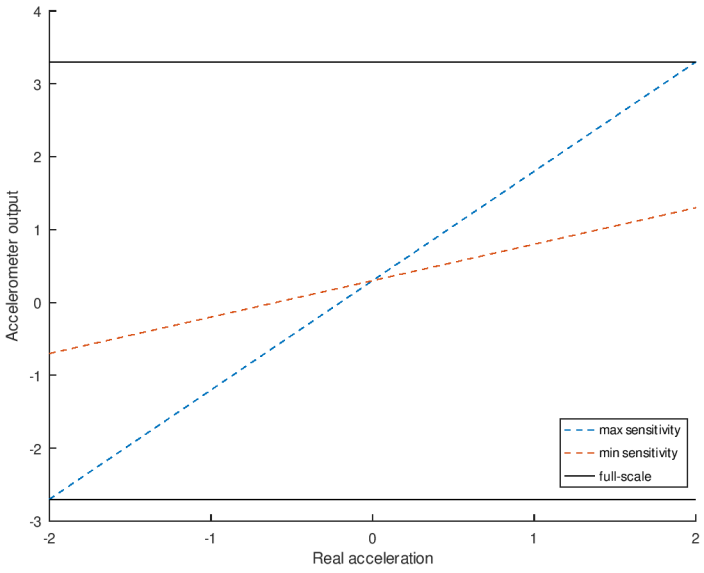

Figure 3.2: Sensitivity

Moreover, in reality the sensor is not perfectly aligned with the axis on which we measure the acceleration (Fig.B.1). Because of this, if we only accelerate the device on the "true" x axis by $\vec{a_x}$, the accelerometer $\vec{A_t}$ will also read the effect of gravity $\vec{g}$. Fortunately, we can use gravity to find the angle between the "true" x axis and the axis of the sensor. In order to get the linear acceleration, we put (3.7) and (3.8) into (3.9) and then isolate $\vec{A_x}$ which gives (3.10). Fig.3.3 illustrates the situation.

$$\vec{A_g} = \vec{g} \sin\theta \tag{3.7}$$

$$\vec{A_x} = \vec{a_x} \cos\theta \tag{3.8}$$

$$\vec{A_t} = \vec{A_g} + \vec{A_x} \tag{3.9}$$

$$\vec{A_x} = \vec{A_t} - \vec{g} \sin\theta \tag{3.10}$$

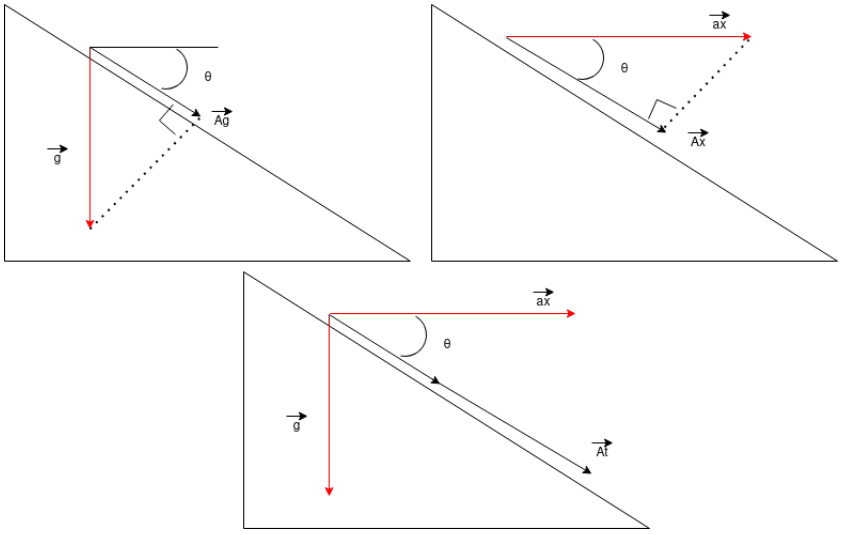

Figure 3.3: Gravity correction

If we assume only a rotation around the y axis then we can rotate the linear acceleration vector by multiplying it by R_y. Note that placing the rotation matrix on the left side or the right side, merely changes the sign of the rotation. One could easily find the appropriate sign by trial and error.

$$\vec{a_\theta} = R_y(\theta)\vec{a_l} \tag{3.11}$$

3.2.2 Integration of a MEMS accelerometer signal

The first thing we will try is to estimate the position and velocity when the accelerometer is standing still. As you can see on Fig.3.4, there is some noise in the output signal. Of course, it is perfectly natural for any real data to contain noise, but integrating noise makes the velocity (Fig.3.5) and position (Fig.3.6) drift. This phenomenon is called random walk.

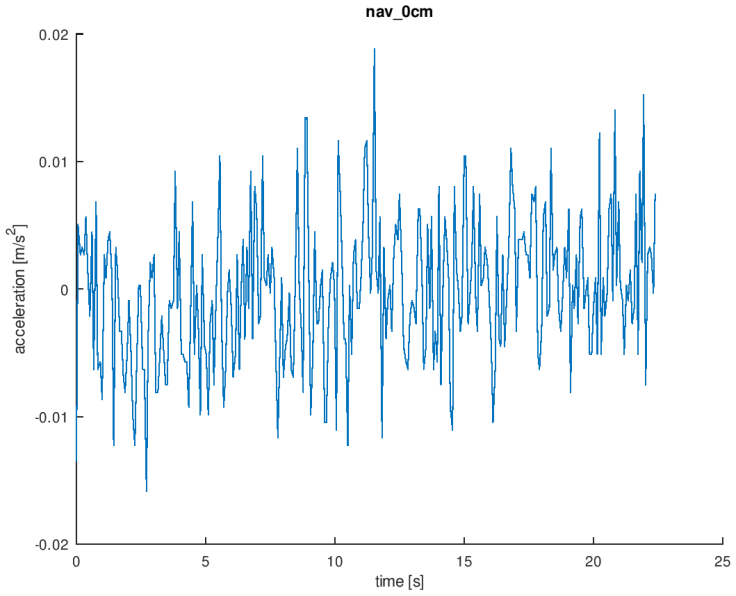

Figure 3.4: Linear acceleration when standing still

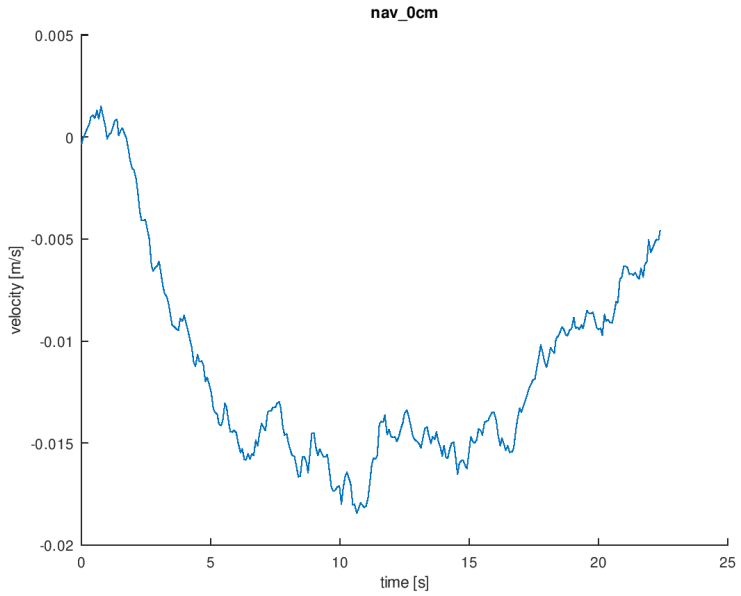

Figure 3.5: Velocity when standing still

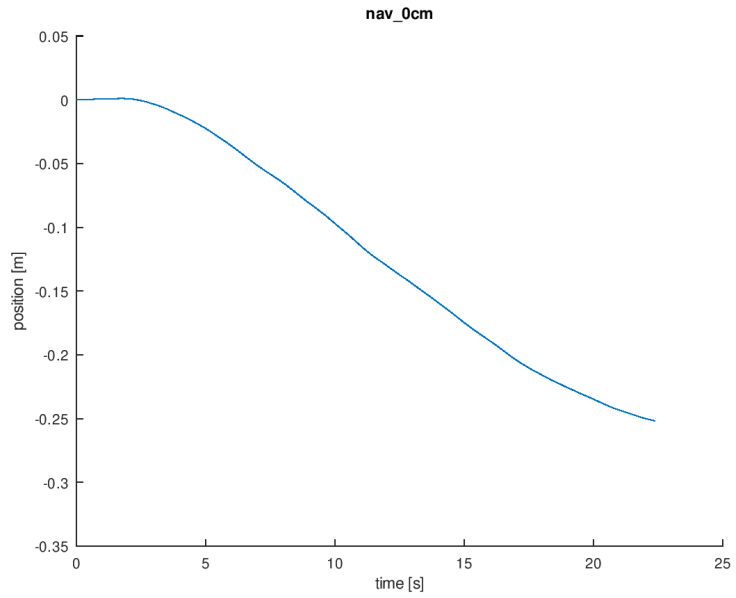

Figure 3.6: Position when standing still

Unfortunately, as you can see on Fig.3.7, Fig.3.8 and Fig.3.9 when the MEMS is being accelerated, the accumulation of errors gets even worse and makes this method impractical. In this experiment, we move the accelerometer over a distance of 110 [cm] and then we stop (velocity = 0). The drift in position is significant since we estimate 2.5 [m]. It is clear that even a small error in velocity can quickly lead to a large error in position when the sensor starts to move fast.

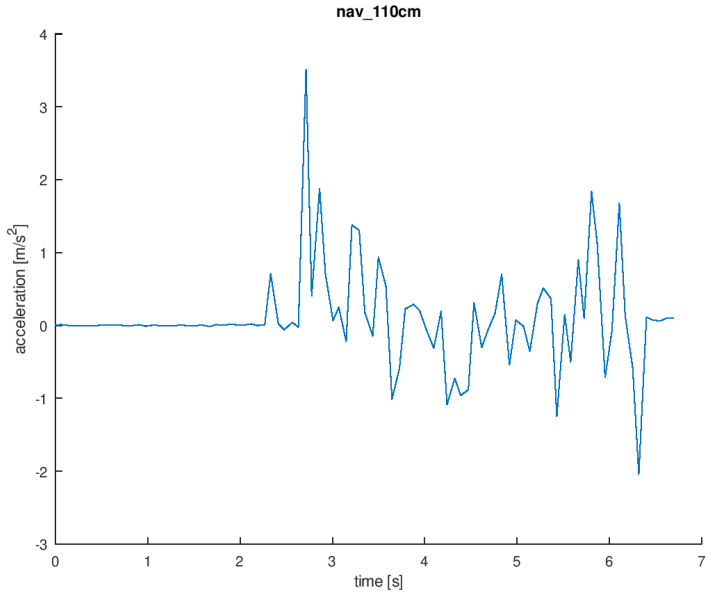

Figure 3.7: Linear acceleration when moving over 110 [cm]

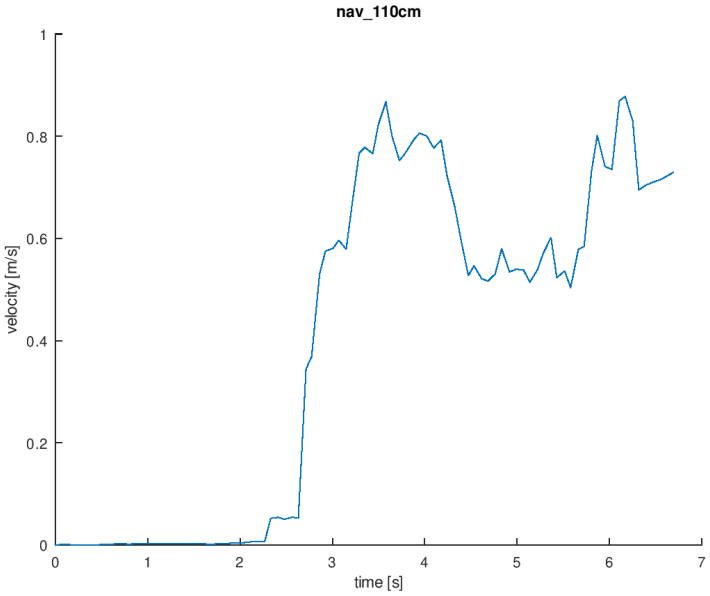

Figure 3.8: Velocity when moving over 110 [cm]

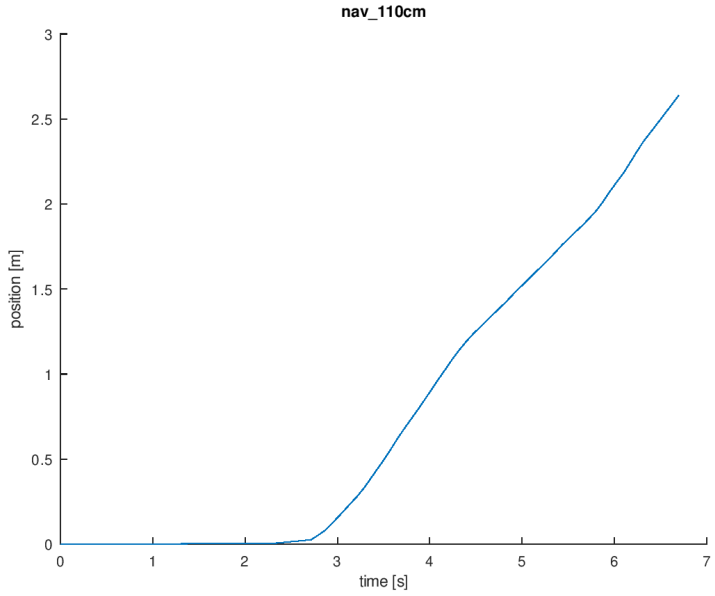

Figure 3.9: Position when moving over 110 [cm]

3.3 Absolute positioning with true-range multilateration

As we have seen in the previous section, inertial navigation is subject to drift. Finding the absolute position of a moving body is possible with a technique called "true-range multilateration" which consist of computing the position of a target based on distinct ranges measured by multiple observers. This approach was used in [21] for sonar-based tracking and thus, we investigated it to later cancel the drift introduced by the double integration of the MEMS accelerometer. The next scenario uses two sonars, without other sensors.

Assuming the target is at (x, y), a first sonar is at $(0, 0)$ and a second sonar is at $(u, 0)$. Sonar i measures a range $s_i = r_i + e_i$ where r_i is the true range and e_i is an error mostly due to inaccuracies about the geometry of the target. By solving (3.12), we compute 2 positions for the target (3.13).

$$\begin{cases} x^2 + y^2 = r_1^2 \\ (u - x)^2 + (y)^2 = r_2^2 \end{cases} \tag{3.12}$$

$$\begin{cases} x = \frac{r_1^2 - r_2^2 + u^2}{2u} \\ \\ y = \pm\sqrt{r_1^2 - \frac{(r_1^2 - r_2^2 + u^2)^2}{4u^2}} \end{cases} \tag{3.13}$$

In most cases, one can be rejected with some context knowledge like, for example, if we assume the target y coordinate is always superior to the y coordinate of the sonars. The problem we are trying to solve here, is to find the intersection of two circles. In practice, we will use s_i instead of r_i which will introduce errors in the process. In fact, the two circles might not even intersect. In such cases, the position is an imaginary number and must therefore be discarded. Fortunately, this does not append very often in practice. However, as shown on Fig.3.10, because of the non linear formulas in (3.13), the computed position will sometimes be way off. In this simulation, we used $\sigma_{s_i} = 0.25$ [m] which roughly models the width of a person (50 [cm] from arm to arm).

Another disadvantage of this method is that the target must be in sight of multiple observers (2 for the bilateration, 3 for the trilateration, ...). While this might not look very problematic, in practice, a method that can directly take into account an independent observation is more convenient (see section 6.3).

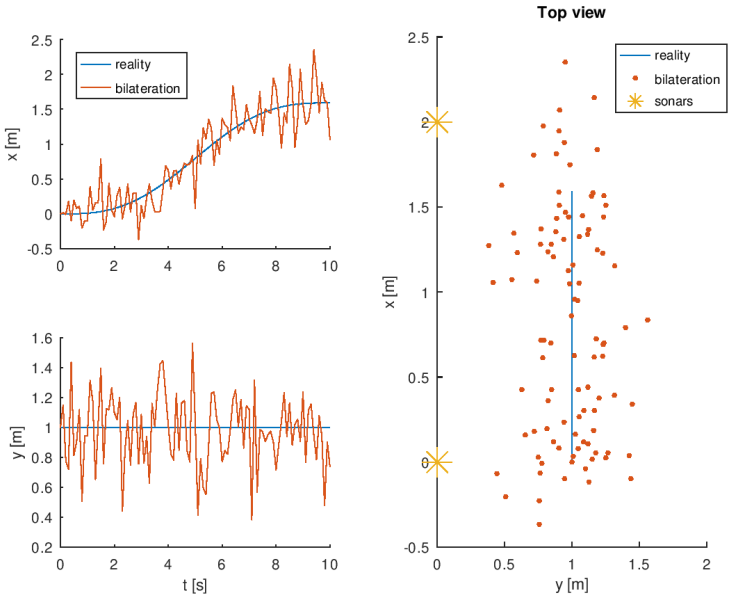

Figure 3.10: Bilateration simulation

3.4 Kalman filter-based tracking

In the previous section, we have seen how we can use "true-range multilateration" to find the absolute position of a target with a least two sonars and no other sensor. We now show a superior data fusion technique, the linear Kalman filter, which allows to combine both the inertial model based on an accelerometer, as described by (3.4) and (3.5), with a sonar. We use two GRiSP boards, one with the Pmod NAV and one with the Pmod MAXSONAR. It is important to note that in practice, inertial navigation is often combined with absolute measurements to cancel drift.

Fig.A.1 illustrates the experiment. As you can see, we will move only on one axis and therefore the sonar range can be interpreted as the absolute position (minus an offset). For this experiment, we have logged the sensors data directly on each GRiSP SD card and the fusion computation was performed offline, after the experiment, with GNU Octave.

The Kalman filter model (3.14) assumes the acceleration remains constant. The model is translated into matrix notation in (3.15) where s is the range measured by the sonar and acc is the linear acceleration extracted from the accelerometer.

$$\begin{aligned} p_t &= p_{t-1} + v_{t-1}\Delta t + \frac{1}{2}a_{t-1}\Delta t^2 \\ v_t &= v_{t-1} + a_{t-1}\Delta t^2 \\ a_t &= a_{t-1} \end{aligned} \tag{3.14}$$

$$\begin{aligned} &X = \begin{pmatrix} p & v & a \end{pmatrix}^T && F = \begin{pmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{pmatrix} && \begin{cases} \sigma_a = 0.1 \\ \sigma_s^2 = 0.01 \\ \sigma_{\text{acc}} = 0.2 \end{cases} \\ &G = \begin{pmatrix} \frac{1}{2}\Delta t^2 & \Delta t & 1 \end{pmatrix}^T && Q = G\sigma_a^2 G^T \\ &H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} && Z = \begin{pmatrix} s & \text{acc} \end{pmatrix}^T && R = \begin{pmatrix} \sigma_s^2 & 0 \\ 0 & \sigma_{\text{acc}}^2 \end{pmatrix} \end{aligned} \tag{3.15}$$

From Fig.3.11, Fig.3.12 and Fig.3.13, it is clear that the fusion of the accelerometer with the sonar gives superior results than the "pure" inertial approach. The drift is canceled with the help of the sonar and the "staircase" effect of the sonar is smoothed by the inertial model. However, good results like these requires a high[1] sampling frequency because the constant acceleration hypothesis must hold between two iterations. In this case, since most of the computations were performed

[1]Relative to the rate at which the acceleration changes.

offline[2] and because we only used one sonar and one accelerometer[3], we achieved a frequency of 17 [Hz]. However, this approach does not perform well if the frequency is too low or if the acceleration changes rapidly. Nevertheless, as we explain in appendix D, it is still possible to obtain good results in such situations with a more faithful model of reality.

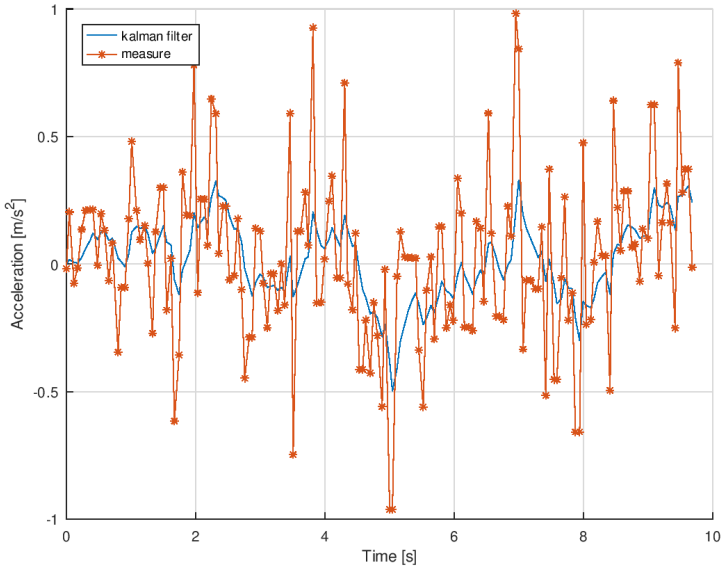

Figure 3.11: Linear acceleration

[2]The data was collected on the GRiSP and it is only after the experiment that we ran the Kalman filter on a computer with GNU Octave.

[3]The 3 axis were sampled, but only one was used for the calculation.

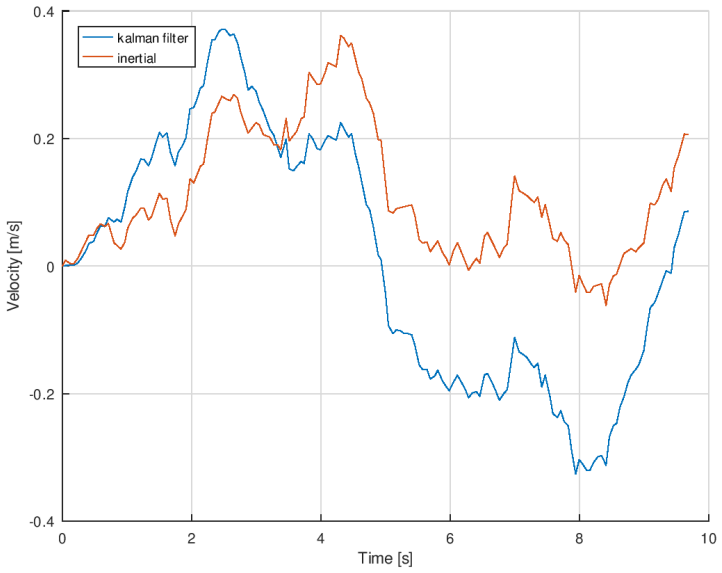

Figure 3.12: Velocity estimation

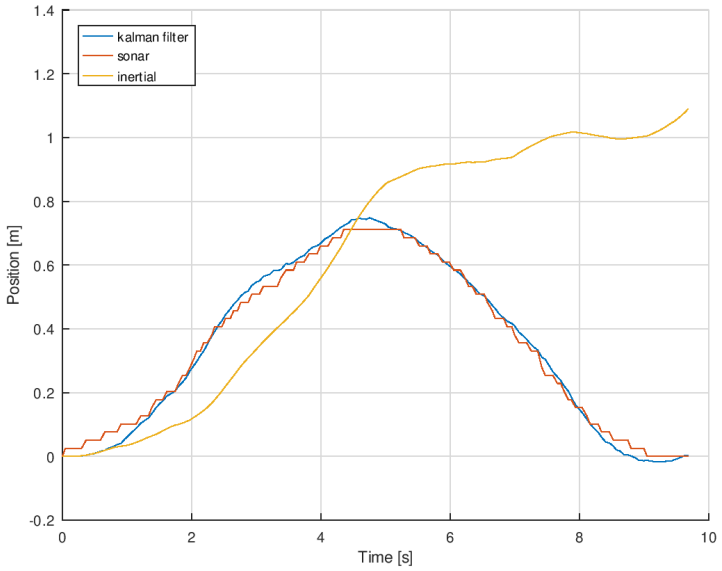

Figure 3.13: Position estimation

3.5 Possible improvements

In the section we explore two possible improvements for tracking that we have investigated, but that we could not implement due to a lack of time.

3.5.1 Multi-target tracking: data association

There are two main problems to address to achieve multi-target tracking: data association and track maintenance. The latter is not really an issue if we assume the number of targets to be known.

Assuming we have a collaborative system, we can associate each sonar measure to a specific target by comparing the measured range with the estimated range (computed from the estimated target position). We say the system is collaborative because the estimated range is provided by the target and the observer improves the accuracy. If the target and the observer were not collaborating, the observer could not distinguish multiple targets and the target would not benefit from the absolute measure of the observer.

Fig.3.14 illustrates this principle. The triangular shape represents an observer (i.e. a sonar) and there are two targets. The top scenario places a cross at an estimated position and the color allows to associate it with a target. It seems natural to associate the cross with the closest target. In practice, however, we do not have the true location, but an estimate provided by the system. The bottom scenario shows the estimated position (the stick man) and the measured range (the green arc). When an observer measures a range, we can compare it with the estimated ranges (the blue and red arrows) which we compute as the distance between the known position of the observer and the position of the stick man. In this example, the observer is seeing the blue target. There are other association techniques based on the same principle like, probabilistic data association (PDA), joint probabilistic data association (JPDA), ...

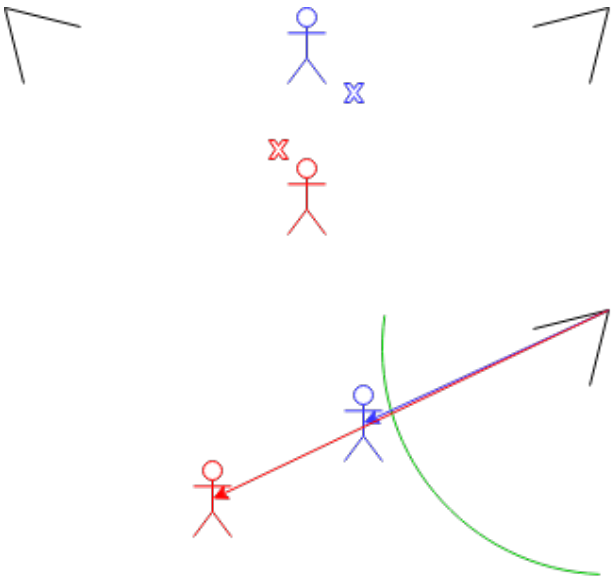

Figure 3.14: Multi-target tracking

3.5.2 Modeling complex trajectories

In this chapter, we have seen that Kalman filters are a powerful technique for inertial navigation because of their ability to both predict the next state and correct themselves. However, as we have explained, the accuracy heavily depends on the quality of the physical model. When the model does no longer reflect reality, the results of the Kalman filter are poor. On systems with very high update frequency, it is possible to model an arbitrary motion with the Strapdown method, but if this rate cannot be achieved, there exists an alternative.

The Kalman filter not only provides a state estimate, but also estimates the state covariance. We can use this information as an accuracy estimation: the greater the covariance, the less accurate we are. The interacting multiple model (IMM) [12] algorithm takes advantage of this accuracy information to combine the estimates of different models. We can model specific behaviours like a constant speed motion, a constant acceleration motion, a constant turn motion, etc and let the IMM select the best one depending on the situation.

This technique, however, requires additional computation and extra modeling efforts. Moreover, the delay introduced by this computation affects even further the update frequency on low-power devices. One should be careful not to deteriorate too much the frequency of the system and only model a few behaviours of the target.

Chapter 4

Software architecture

4.1 Approach

Our approach to perform sensor fusion is based on the separation of concerns principle. The hardware and sensor drivers are provided by the GRiSP platform and all the boards are connected via standard WIFI protocol. We built a GRiSP application[1] that uses Hera, a distributed framework designed for sensor fusion with Erlang. You can get access to the application via Github[2].

Fig.4.1 gives an overview of the data flow. Each GRiSP board uses the Hera framework which provides a data storage, handles the data sharing via broadcast and provides a generic measurement behaviour. Then, on the client side, we implemented "sensor interface"(s) on top of the provided Pmod drivers as well as a "fusion engine" module in which we both describe the Kalman filter parameters and fetch the appropriate data from the local data store.

The overall approach of Hera is to collect data on each node, exchange the data via UDP broadcast, run the computation on each node, and again share the results. A particularity is that only the most recent data are available for each sensor and so, the application only offers a temporary data storage. Since the data is simply shared by broadcast, each node computes independently from each other. We are not trying to achieve a consensus, however, the results of all the nodes should be close because of the real-time nature of the system. The difference is due to small data loss because UDP is not reliable, small changes in the data arrival time/processing time as well as small differences in the state of each Kalman filter.

[1] An application targeting the GRiSP platform

[2] `https://github.com/sebkm/sensor_fusion`

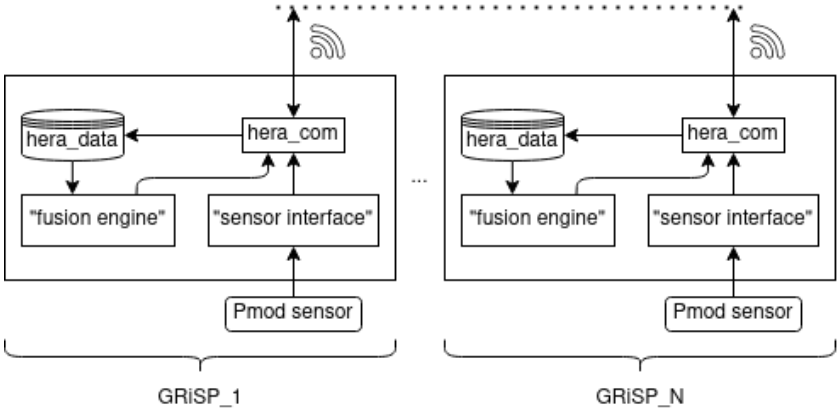

Figure 4.1: Data flow overview

The full system is fault tolerant thanks to the dynamic and asynchronous model provided by Hera, but also because of redundancy achieved by having multiple GRiSP boards with sensors and by running the same computation on multiple nodes. What this mean is that only one node is required for the system to be operational, but obviously, having multiple nodes (and sensors) improves the accuracy. The dynamic and asynchronous nature of the system is an essential property for fault tolerance because the only consequence of a failure will be less data being broadcast. The only thing to do then, is to restart the crashed node and it will reintegrate the system immediately, as if it was never gone.

A last point worth mentioning is the use of a replicated ETS table working on top of the RPC module. This data storage was imagined to ensure that mandatory information (e.g. a sensor calibration) remains available even if certain nodes die and recover.

4.2 Hera a framework for sensor fusion

To facilitate sensor fusion, we designed Hera, an Erlang/OTP framework for handling asynchronous and dynamic measurements. Hera was developed with a special focus on simplicity because we expect it to be used by other students in the future and we want them to be able to understand it quickly. You can get access to Hera via Github[3].

We use a small supervision tree (Fig.4.2). The top level supervisor `hera_sup` supervises three processes: `hera_data`, `hera_com`, and `hera_measure_sup`. These processes are considered vital for the application and are not expected to fail often. Moreover they are considered independent.

[3] `https://github.com/sebkm/hera`

The second supervisor `hera_measure_sup` supervises `hera_measure` processes. Theses processes are considered independent of each other and should be dynamically started. Because a measure process may interact with real world components such as sensors, we expect them to fail more often. This supervisor is not designed to supervise a very large number of processes because there is a natural limit to the number of `hera_measure` that can run simultaneously (network congestion, computational limitations, fixed number of sensor connectors, ...). However, the design remains open by allowing the user to supervise its `hera_measure` processes himself instead of using our built-in supervisor. The only purpose of this supervisor is to ease the implementation effort of the user.

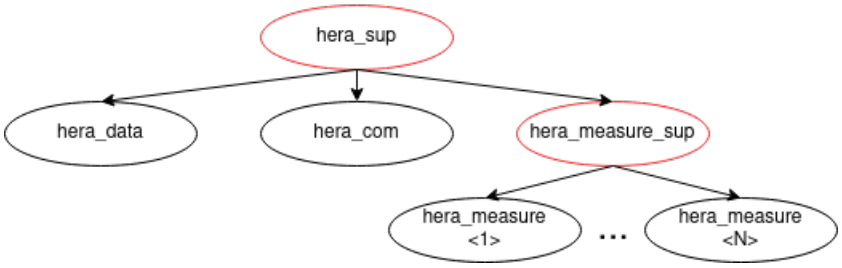

Figure 4.2: Hera supervision tree

`hera_data` is a `gen_server` used for storing measurements data. It stores only the most recent data identified by a name and the node who sent it and marks it by a timestamp. This way, the user knows when the data was received. If he also wishes to know when the measurement was performed he must send a timestamp along with its measure. By setting the `log_data` environment variable to `true`, it is also possible to log every measurement in a unique csv file for each data identifier (name and node).

`hera_com` is a communication process for sharing data across the network with other nodes. It uses a multi-cast UDP group for fast but unreliable data sharing. Sharing all the data enables to introduce redundancy by running the same computation on several nodes since the result will be broadcast. Another advantage is the ability to introduce "load balancing" capabilities by sharing the computational load. For example, the node that produces sensor measures may let another node perform the sensor fusion computation. In fact, this is how Fig.7.10 was produced.

The `hera_measure` module is a generic measurement behaviour. It allows the user to simply provide a few parameters and an initial state like a calibration in the `init(Args)` callback as well as a `measure(State)` callback to perform a measure. The expected result is a list of numbers (or the atom `undefined`) followed by the new state. This specification is flexible and allows, for instance, to discard certain

values or to implement a complementary filter stored in the callback state. If the measure must be synchronized, the process waits to receive an authorization from the synchronization process (section 4.3) before calling the `measure` callback and will manage the subscription itself. The `measure` callback will be invoked in a loop with a parametric delay[4] between each iteration. Later, we will refer to this as a cycle.

Fig.4.3 illustrates how the different processes interact. The top scenario shows how a measure is shared to all the nodes, the bottom left shows how data can be retrieved and the bottom right shows how the user can start a measure. We say the model is asynchronous because the data may arrive at any time and the `hera_measure` process that needs some data will fetch it from its local `hera_data`. There is no notion of event that would trigger a computation and the data cannot be consumed.

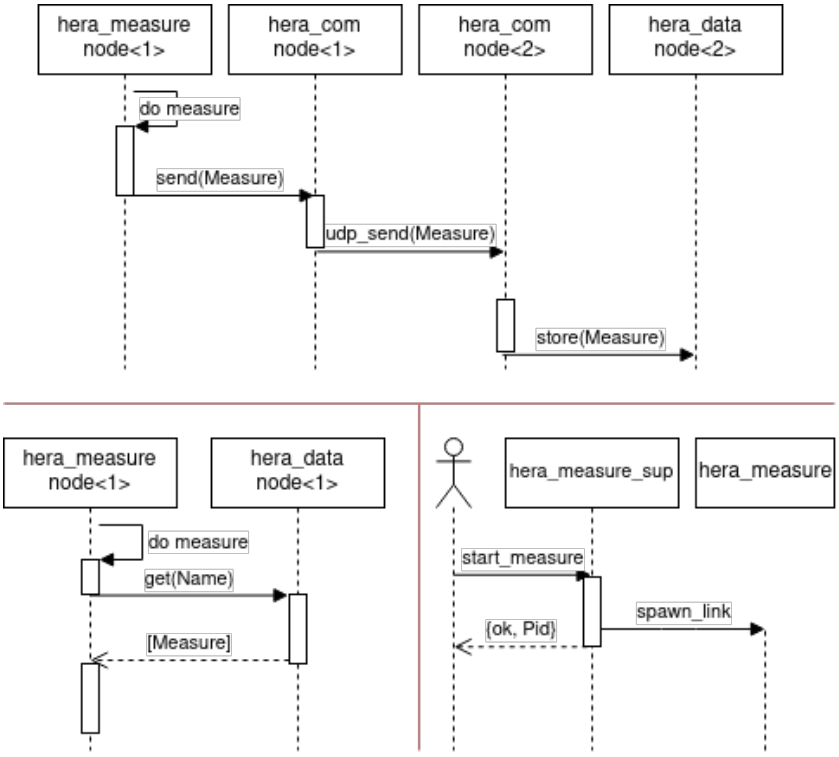

Figure 4.3: Interaction between processes

[4]Chosen by the user to reduce the sampling frequency in certain conditions. For example, if a measure is too fast and overloads the system. The delay may be nought.

4.3 A measurement synchronization extension

The use of sonars introduces the cross-talk problem occurring when multiple sonars interfere with each other. Avoiding cross-talk requires synchronization between the sonars. A mutual exclusion (mutex) solution has to be brought. We had 3 requirements: satisfying the mutex property, dynamic addition/removal of processes, and fault-tolerant synchronization. After reviewing several algorithms [34], we chose the centralized approach for simplicity, but we improved its fault tolerance with monitors and heartbeats. The extension is implemented in a separate application and is available on Github[5].

Fig.4.4 shows how the synchronization works. First, the measure subscribes itself to `hera_sub` which forwards the subscription to the dedicated[6] synchronization process which will then authorize the `hera_measure` in due time. The idea is to allow the system to dynamically synchronize measurements. Fig.4.5 illustrates how the processes are supervised or monitored and Fig.4.6 describes how the processes react upon reception of a "DOWN" message. When a `hera_measure` process crashes, `hera_sync` simply authorize the next process. If `hera_sub` fails, `hera_sync` exits because it cannot run independently. Last but not least, when `hera_sync` dies, the subscription server removes it and the measurement processes resubscribe themselves. If `hera_sub` is also crashed, the measurement processes wait and retry after 1 [s].

[5] `https://github.com/sebkm/hera_synchronization`

[6] A new process is created if needed.

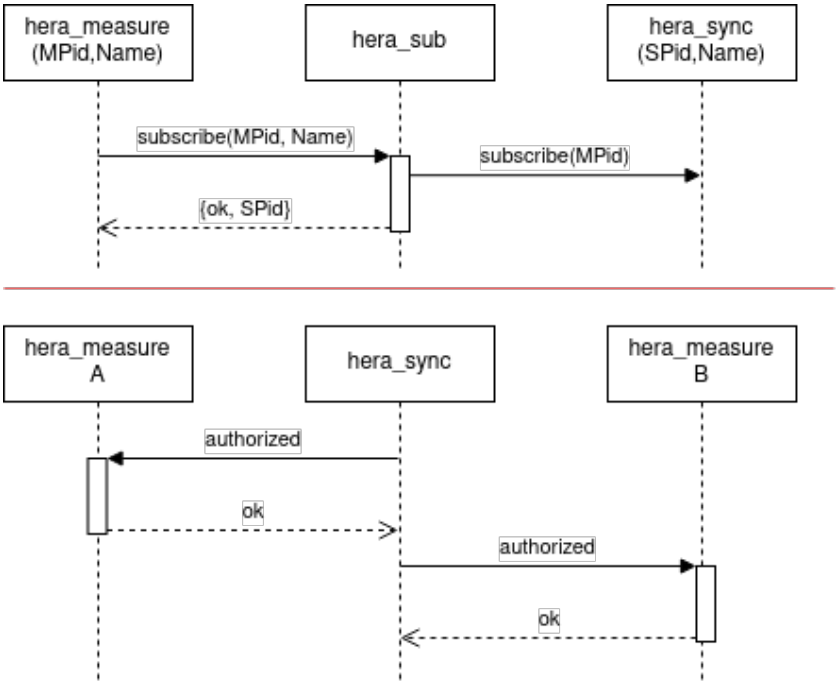

Figure 4.4: Synchronization principle

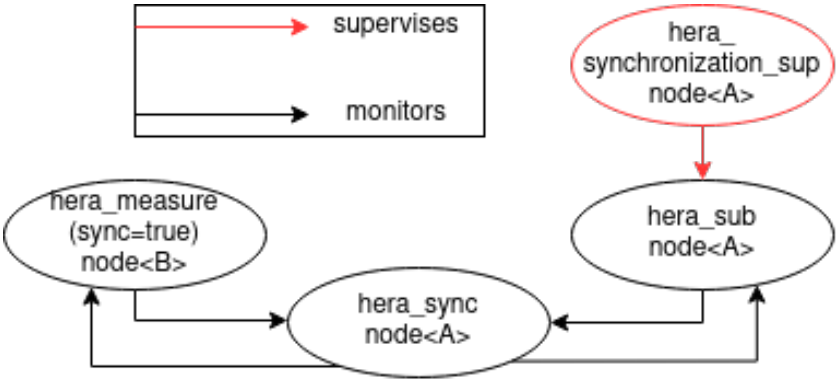

Figure 4.5: Synchronization monitoring

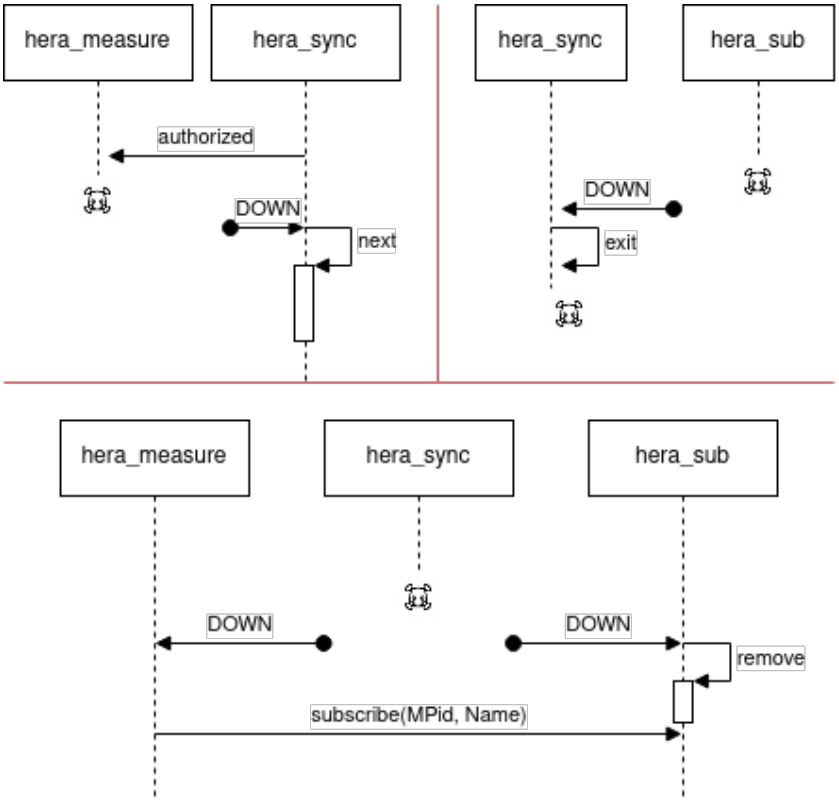

Figure 4.6: Reaction to "DOWN" messages

4.4 A matrix library with support for the asynchronous model

Along with the Kalman filters, we provide a matrix abstract data type (ADT) and a small library with basic matrix operations. It quickly became apparent that such a library was necessary because Erlang does not natively provide support for numerical computation. Moreover, we realised the importance of embedding the asynchronous arrival of data inside of the Kalman filter model itself. To do so, we imagined a way of writing variadic matrices with list comprehensions and by representing a matrix as a list of lists.

This representation has three flaws: inefficient random access, heavy memory consumption, and inefficient matrix operations in pure Erlang. However, the two first are of little importance. Indeed, we do not need random access considering we can pattern match the matrix and the "heavy memory consumption" turns out to be of no concern with the GRiSP boards. A long-term solution to the general inefficiency would be to use an Erlang NIF library. Actually, this project is already

ongoing (see chapter 8).

A benefit of the list of lists representation is to enable the use of list comprehensions to build variadic vectors. Listing 4.1 shows how we write the complete observation vector from (6.9) while preserving the asynchronous assumption. As we build the Z vector with list comprehensions, it is possible to write it as if all the data were present and then, at run time, the number of lines will depend on the available data. For example, if Mag = [] the corresponding line will not be present in Z and if Sonars contains 2 entries then the Z vector will have two lines for the two different ranges. This offers a lot of flexibility and gives some support to fault tolerance because the computation will continue to work despite the absence of certain data.

Listing 4.1: Variadic vector (Erlang code)

```
Z = [[-Ay    ] || [Ay,_] <- Nav] ++
    [[-Gz    ] || [_,Gz] <- Nav] ++
    [[-Theta] || [Theta] <- Mag] ++
    [[Range ] || [Range,_,_] <- Sonars].
```

4.5 A visualization tool

Any real application needs a graphical user interface (GUI) to communicate with the end-user. For a real-time application, we cannot simply collect the results in a CSV file and only review them later. We need to develop an application to visualize the data in real-time. In [21], they developed a web application based on the Phoenix framework to provide a minimal GUI. We tried to reuse this application, but quickly realized that this approach was inefficient for drawing a large variety of plots. Instead, we decided to build a small GUI with GNU Octave [7]. With it, it was easy to make a modular and extensible tool for displaying the data in real-time.

Octave is not made for networking and thus we could not easily read the data directly from Hera's UDP group. It is possible in theory, but requires installing the "instrument-control" package of Octave and parsing the binary "external term format" of Erlang. However, since Hera already provides logging capabilities, we decided to simply read the CSV file in real-time.

Fig.4.7 shows the main window of liveView. There is a view selection menu, a button to fetch the data source file generated by Hera and a replay mode to review the data with a "real-time" feel. Fig.4.8 shows the "train tracking" view. We have three graphs that shows information about the position of a train (see chapter 6) and there is a colored indicator that shows if data is being received in real-time (green) or not (red). In replay mode (used for the screenshot), the indicator is red.

With this application, it is easy for the user to add its own views. This is what we have done with Fig.7.3.

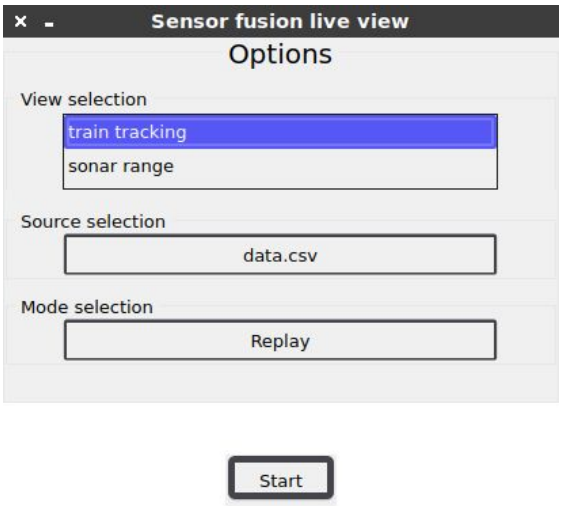

Figure 4.7: LiveView main window

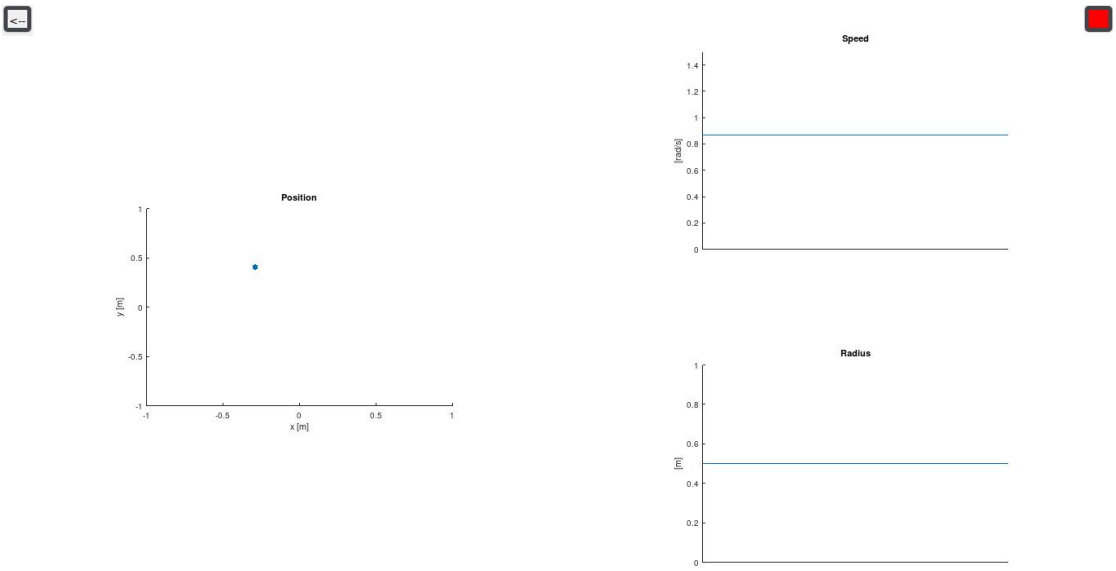

Figure 4.8: LiveView train tracking window

4.6 Properties

4.6.1 Asynchronous

Hera uses an asynchronous computation model. Concretely, computations and data gathering are separated from each other. A computation is a process running independently inside of its own loop. At the beginning of each loop, a computation process tries to fetch data from its local data store and incorporates the available data. As a result, only one value is enough to perform a computation.

The reason we were able to offer such flexibility is two fold. First, the Kalman filter is very flexible on its own because the observations (data) can be incorporated one at a time. Second, our matrix abstract data type supports "variadic definition", meaning that it is possible to write all the Kalman filter equations once, and let the program pick the ones it needs.

4.6.2 Dynamic

Our software is made to be dynamic on multiple levels. First, any measurement (sensor or computation) can be started at any time and this is also true for synchronised measurements.

Second, any known GRiSP board can join or leave (restart or crash) the system while it is running. This allows for even more flexibility and is mainly possible because of the asynchronous computation model.

Third, measurements supports hot code reloading. This functionally almost entirely comes for free with Erlang, but we made sure that the changes will be persistent even after a reboot. This feature is extremely valuable for developers because they can test and debug their code without having to deploy the whole system every time.

4.6.3 Fault-tolerant

We have shown the fault-tolerance achieved by Hera with a small supervision tree and monitors. This property is extremely important because no-one should ever assume their code is bug free. There are many reasons that could provoke an occasional crash and the restarting strategy allows the system to keep running despite such events. Of course, the dynamic and asynchronous nature of our software allows for this restarting strategy.

4.6.4 Modular

The modularity takes several forms.

First, there is a hardware modularity from the Pmod sensors because they have been imagined as a plug-and-play system. We can choose what Pmod we want on a certain GRiSP and even use different Pmod modules without having to change the whole system. The only change to make is around the sensor interface.

Second, the measurement modules are modular because they can be started or modified dynamically. The only requirement is to perform the necessary calibrations, which typically takes place when the system is installed.

Third, the Kalman filter can be called modular because the computation takes the available data instead of waiting for all them. In fact, this is an extremely valuable property of the Kalman filter. Moreover, it is possible to perform more predictions than update (or the other way around) depending the system frequencies.

4.6.5 Soft real-time

As one could expect, all the data is processed in soft real-time and the results can be visualized on LiveView. Nevertheless, this property must be emphasis because nowadays, many edge devices still delegates the computation to the cloud and are merely used as data gathering devices. With such architecture, the latency between data collection and result visualization increases too much for real-time tracking.

4.7 Where does Hera come from and the need for refactoring

Hera was originally developed in [21] and the authors, Julien Bastin and Guillaume Neirinckx, demonstrated its capabilities for a person tracking application based on true-range multilateration with sonars. It was a first attempt to provide a distributed framework designed for sensor fusion on the GRiSP platform with Erlang. Nevertheless, the ambitions of the authors were large and they tried to make the framework as generic as possible by providing many features like filters, worker pools, synchronized measurements, warm-up phase and data logger. The result was more than convincing and so, we planned to use their application as foundation for further sensor fusion experiments in the field of inertial navigation.

The original version of Hera was an attempt at sensor fusion and was used to build a demonstrator. The authors have done a great job, but the application was only at its first iteration and could be improved on. An analysis of the application source code revealed a few problems. This does not mean that the application has not reached its goal of genericity nor that it does not work, but these problems had to be addressed in some ways. We decided to write a new version of the framework to solve these problems, but also to simplify the code because we realised that it was important since other students may end-up using Hera for their own work. If

they need to understand how Hera works or to modify it for some reasons, they should be able to do so with a minimal effort. However, our purpose was not to reinvent Hera and so, we tried to keep the core ideas of the original version. Some of them could have been subject to a redesign, but since our predecessors have had a lot of time to consider their options and obtained good results, we decided to trust their choices. Still, we will discuss one alternative in section 8.3.

Let us now have a look at the issues we found. Note that for the rest of this chapter, we refer to the original version of Hera, not the redesign.

- **Synchronization:** The synchronization mechanism offered by Hera consists of authorizing processes to work one at a time to solve a distributed mutual exclusion problem. However, if a node is too slow to perform a measurement or if the network is too slow, an other node will be allowed to perform a measurement and the late reply will still be accepted. As a consequence, the algorithm does not provide mutual exclusion.

  Moreover, during the synchronization, a worker is expected to finish its measure in a certain time. This duration is fixed, but it should be a parameter because different measures require different timings. Ideally, an implementation that does not require prior knowledge on the system would be better because finding this parameter might be difficult for the user.

  About prior knowledge on the system, it would be nice that synchronized measurement could also be dynamic because currently, it is required to "declare" them in the application environment.

- **Worker pool:** In an attempt to provide a dynamic supervision system while limiting the number of workers running at the same time, the application uses a pool of workers. Unfortunately, a mistake in the workers supervisor specification introduces a leak in the pool. Indeed, when a worker crashes, the pool server will remove it from the pool and put a queued worker instead, but the supervisor will restart the dead worker. The problem is that the server is suppose to limit the pool size and because of the restarting, there is a leakage.

- **Workers restarting:** The current restarting strategy for the measurements, calculations and filters is to always restarted them. In case of crash, this is problematic because we cannot allow the "warm up phase" while the system is running. For example, in the case of a sonar, a person could be in the way. A better approach would be to start a measurement with its warm up value a.k.a. calibration as argument, so that it could be reused in case of crash.

- **Workers hibernation:** The application features a "clean" restart mode that enables a worker to restart without killing it. This is achieved by making the

process entering in hibernation instead of exiting normally when the work is done. This is not a very good idea because it is clearly conflicting with the pool system since once the maximal number of worker will be reached, new workers will never be allowed to run. Additionally, there is no real benefit because we can simply start a new worker (the same but as a different process). The only requirement is to find the appropriate sequence number when the worker starts.

- **Hera is limited to GRiSP:** Currently, the application can only work with GRiSP because of an OS type check. This must be removed because ideally, we want Hera to work on any device.

We also noticed some weird design choices that makes the application more complicated than it should:

- **The supervision tree:** It represents a large part of the code and it is kind of a maze around the "pools" because a lot of "back and forth" are required to understand how it works.

- **The pool system:** The purpose of this system is to allow dynamic insertion of workers. This could be an application on its own and should be abstracted or simplified. Moreover, this system is also suppose to limit the number of workers per pool. Yet, one can wonder if this feature even makes sense in the context of sensor fusion because there are already hardware limitations (e.g. we cannot have 1000 measurements at 50 [Hz] each on the same sonar). Still, dynamically starting workers is very useful in a real life scenario and therefore, a compromise should be taken there: dynamically starting workers for a minimal complexity.

- **Restarting a measure:** This feature is conflicting with the pool system. Indeed, when the measurements are done, the process goes in hibernation and thus, remains in the pool. Because of this, once the maximal number of workers will be reached, new workers will never run.

- **Filtering:** The idea of a filter is very simple, make a measurement and decide whether or not it should be taken into account. In Hera, the approach is quite different really. A filter is a server with its own pool system and supervision tree, and the filter is in charge of sending the measure if it is valid. This is just over engineering. A filter should be a simple function because there is no need for multi threading.

- **Calculations:** A calculation is very similar to a measurement and these two features could be merged into the same kind of process.

The analysis took three days and the code really felt cumbersome, especially the supervision tree. The application is not doing that much and should not be so large. We already revealed a few problems and since the code is quite complicated, fixing them is not the most appealing solution.

As first real users of the application, we felt some bad design choices:

- **Launching interface:** Hera requires to pass a measurement function as callback along with all the parameters which must be formatted by the application via additional function calls. This make the launching process complicated and dirty. We would have proffered a cleaner interface with a proper callback module for each worker process.

- **A stateless interface:** Allowing the client to have its own "state" like in a generic OTP server for a measurement process is necessary in case the user wants to implement something like averaging or a Kalman filter. A possible hack is to store it under some name inside of Hera or to use an ets table, but this is not very clean.

- **Data retrieval:** We did not like so much the retrieval of data. The framework groups the data per name and returns a dictionary where the key is the name of a node and the entry is the actual data. We believe that all required processing to extract the desired data should be handled by the framework itself because it just makes the whole process unnecessarily cumbersome.

- **Data return type and logging:** The framework expects the callback function to provide an Erlang term as "sensor data". However, sometimes the output is not just one value and it must be packet into, for instance, a tuple. Unfortunately, the consequence is that the logged data now requires additional parsing. This could all be avoided by directly logging everything in csv format and changing the return type specification.

With all these points in mind, it was time to refactor!

Chapter 5

Fault tolerance analysis

We validate the fault tolerance of the application by fault injection. Concretely, we disable certain parts of the system and observe the behaviour. To do so, we create dedicated `hera_measure` processes, called "observers", that measure useful information. We use 4 GRiSP boards and 1 computer for all the tests. There are 4 measures running per node which makes a total of $20 = 4 * (4 + 1)$ observers running at the same time. The exact moment of the fault injection is unknown, but we approximately know when it happens and we will mark it on every graph. We insist on the fact that these "observers" are used only for fault tolerance validation and are therefore not running on the system under normal circumstances. In the following paragraphs, the word "cycle" refers to the parametric constant time between two successive callbacks (see section 4.2).

Two kinds of observers are used: "counter" and "elapsed". The "counter" fetches from `hera_data` the number of observers who recently (less than 1.5 cycles ago) sent a value with the extra condition that these observers are at least 2 cycles old. This condition is used to be sure that a killed `hera_measure` is visible on the graphs because it should not be counted for at least two cycles. The "elapsed" is a synchronized observer used to analyse the resilience of the synchronization mechanism by measuring Δt, the time elapsed between two successive authorizations (for the same observer).

5.1 Process crashes

Fig.5.1 shows the results of the fault injection on `hera_measure` processes ("counter" and "elapsed"). The percentage indicates how many processes per node are killed. For a "counter", a cycle lasts for 1 [s], but only 100 [ms] for the "elapsed" (or 200 [ms] if it is alone). As the "elapsed" are synchronized, they are only monitoring 5 processes and not 20 because a key (name, node) must be unique and the

synchronization is based on names (recall that each node is running 4 observers with different names). So, there are 4 concurrent synchronizations with 5 measures per synchronization. We can clearly see that the count gets lower after the fault injection and then returns to normal a few seconds later. We also see that Δt decreases since it takes less time to receive an authorization when a participant is gone. On the two graphs, it takes roughly 2 [s] to restart all the processes. On the top graph, we should subtract up to 3 [s] (3 cycles) because of the way we count [1].

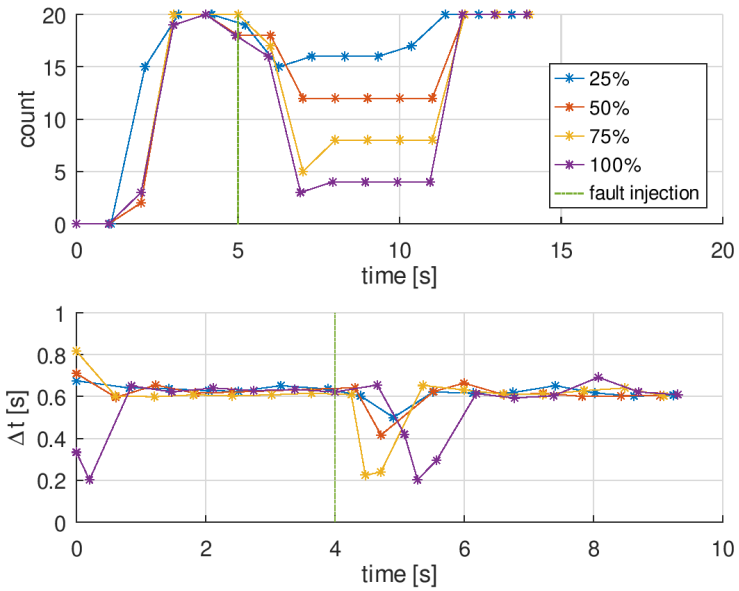

Figure 5.1: Fault injection on "counter" (top) and "elapsed" (bottom)

Fig.5.2 shows what happens when we kill `hera_data` and `hera_com` multiple times in a row. Killing the data storage process should result in a loss of information and therefore, the count should get lower. But, the loss is only partial because only the data stored before the kill is gone. 5 cycles after the last kill, the situation is restored. Observing the fault injection on `hera_com` is more difficult and we have to lower a cycle to 300 [ms]. Again, we can clearly see a loss of information after the kills and the situation returns to normal 1 [s] later.

[1] In fact, you can see that same delay at the beginning.

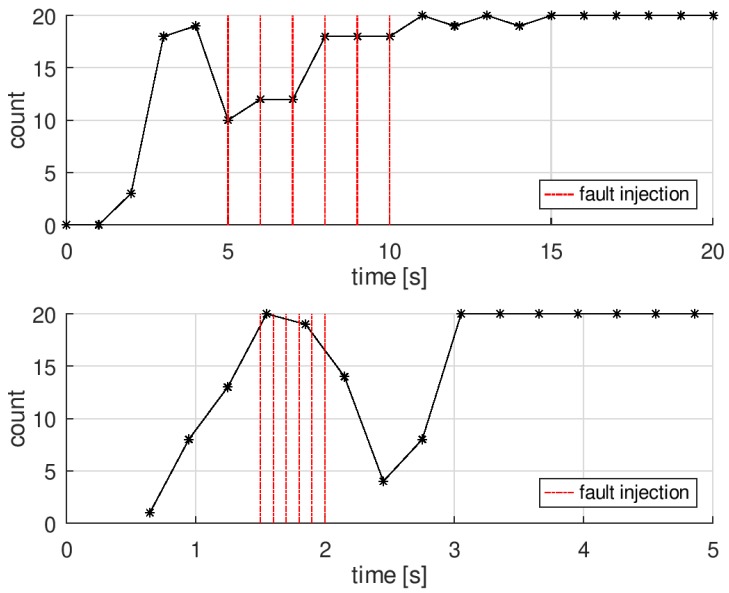

Figure 5.2: Fault injection on `hera_data` (top) and `hera_com` (bottom)

5.2 Synchronization resilience

For the resilience test of the synchronization mechanism, since an "elapsed" cycle is 100 [ms], it means that in total it takes ≈ 0.5 [s] to authorize every node. As explained previously, when the `hera_sync` process dies, the subscribed measures receive a "DOWN" message and must resubscribe. Therefore, the order in which the processes are authorized before the fault injection might be different than the order after (first come, first served policy). On Fig.5.3, you can see that when we kill the `hera_sync` process, the 5 subscribed measures are perturbed for 1 cycle and there is a gap similar to the one at the beginning. When we kill the `hera_sub` process, not only will the 20 measures send a subscription request, but they might also end-up sleeping for 1 [s] (or more) if the server is dead when they try to contact it. As a result, there is quite a significant gap after the kill.

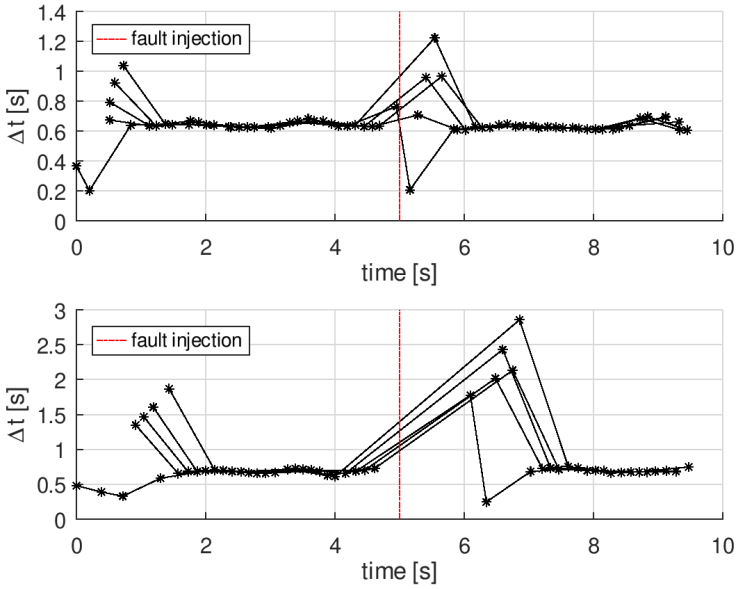

Figure 5.3: Fault injection on `hera_sync` (top) and `hera_sub` (bottom)

5.3 GRiSP board crashes

Finally, we try to power-off a few GRiSP boards to see how long it takes before a dead node is detected. According to the Erlang reference manual, the detection should take TickTime $\pm\frac{1}{4}$ TickTime. In our configuration, TickTime is set to 8 [s]. Hence, the detection should take between 6 and 10 [s] and from what we observe in Fig.5.4, it is respected. We can measure the detection time by looking at the difference between the normal Δt and the spike. For example, the first spike is at $\Delta t \approx 10$ [s] and then goes down to $\Delta t \approx 2$ [s]. Therefore, the detection takes $10 - 2 = 8$ [s]. When we power-off the node that is running the synchronization application, a "failover" takes place. As a consequence, the last spike is slightly higher because the "observer" processes only try to subscribe once per second and the first attempt is likely to fail since there will be no `hera_sub` server alive while the application is being restarted on another node.

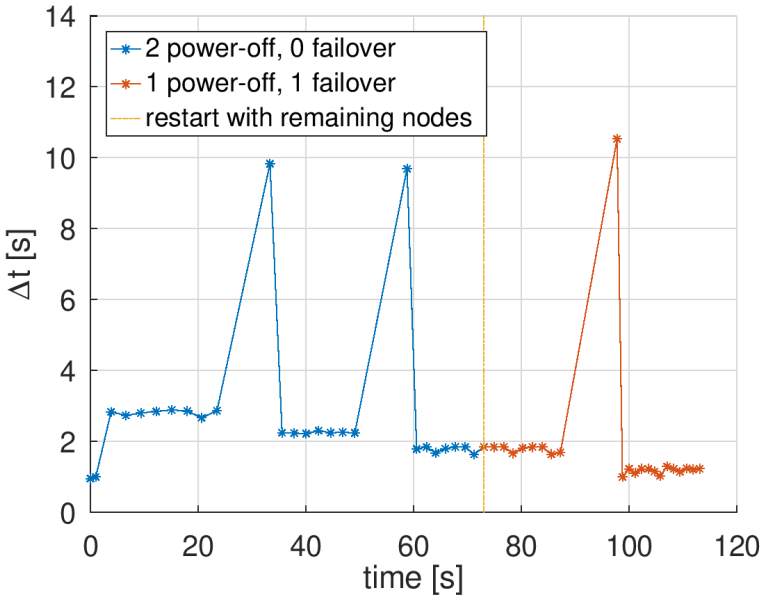

Figure 5.4: Hardware failure with synchronization

5.4 Verdict

We conclude that the system continues to run correctly as long as one board is running at every time instant. This property is very strong and allows to continuously do sensor fusion, despite some crashes. There is only a temporary interruption when a GRiSP board encounters a hardware failure and is restricted to synchronized measures (the rest of the system is not affected). This interruption directly depends on the TickTime (4 times the heartbeat) which can be changed via the `net_ticktime` environment variable, but a too low value may result in a false positive. In our settings, a heartbeat is sent every 2 [s] and hardware failures are detected in at most 10 [s].

Chapter 6

First phase: experimental sensor fusion with Hera

In order to verify if sensor fusion with Erlang and GRiSP is a viable option, we first developed an experimental model based on the extended Kalman filter (EKF), by gradually adding new sensors or complicating the fusion computation. This gave us the opportunity to understand how each sensor works as well as testing the software. In this section, we explain how we built that model and review some of the results. Note that to avoid any confusion, X is used to denote the state vector instead of x which represents the position along an axis. Additionally, for conciseness, we omit to give the Jacobian of the f and h functions.

6.1 Setup

The experimental setup is a train toy with a circular path (Fig.6.1). The train carries a battery and a GRiSP board equipped with a Pmod NAV. Except for the first model (section 6.2), we also use two Pmods MAXSONAR. They are installed on a wooden support with a small tilt to prevent them from seeing the rails (Fig.B.2). The angular velocity "true ω" of the train is constant and was measured by counting the number of turns the train performed over 60 [s], but the precision of this method remains limited. To better visualize the benefit of sensor fusion, we compare the estimated ω for each version of the model (Fig.6.5).

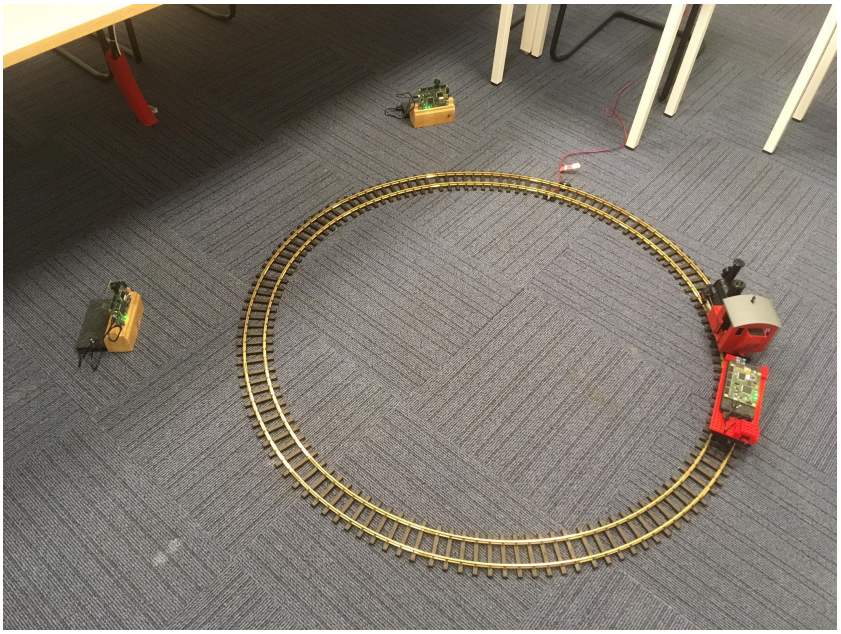

Figure 6.1: Setup for the experimental model

6.2 Angular velocity estimation

The first experimental model consists of estimating the angular velocity ω of the train with an accelerometer by measuring the constant centripetal acceleration a_c [35]. In the EKF parameters (6.1), we assume the radius $r = 0.57$ [m] is known. Since a_c is a constant, we compute $\sigma^2_{a_c} = 0.8$ directly from the accelerometer data (Fig.C.1).

$$\begin{aligned} X &= \omega & f &= \omega & Q &= 0 \\ h &= r\omega^2 & z &= a_c & R &= \sigma^2_{a_c} \end{aligned} \tag{6.1}$$

The "a_c only" curve on Fig.6.5 shows that the estimation is difficult mostly because the train introduces lots of vibrations. We also observe that the value converges towards a too high number. It cannot be the bias of the accelerometer because it was removed by an initial calibration, but a slightly higher r would already reduce the error. Actually, there is a small uncertainty on the value of the "true" r[1]. We could try to make the estimation as good as possible by optimizing the parameters, but instead we will improve it by adding other sensors.

[1]We do not exactly know the position of the MEMS accelerometer on the Pmod NAV chip.

6.3 Position estimation

In this section, we show how to estimate the position of the train with an accelerometer and at least one sonar. As a reminder, from now on until the end of this chapter, we use two Pmods MAXSONAR.

We can estimate the position of the train from the inertial equation $\theta_{k+1} = \theta_k + \omega\Delta t$ and the drift can be corrected with the absolute measurement d of a sonar. From the estimated position (x,y), it is possible to estimate the distance between the train and a sonar located at (P_x,P_y) with (6.2). The error $\sigma_d = 0.25$ is defined as half the length of the train.

$$\hat{d}(x, y, P_x, P_y) = \sqrt{(x - P_x)^2 + (y - P_y)^2} \tag{6.2}$$

$$\begin{aligned} X &= \begin{pmatrix} x & y & \theta & \omega \end{pmatrix}^T & R &= \mathrm{diag}(\sigma_{a_c}^2, \sigma_d^2) \\ f &= \begin{pmatrix} r\cos\theta & r\sin\theta & \theta + \omega\Delta t & \omega \end{pmatrix}^T & Q &= O_{4\times 4} \\ h &= \begin{pmatrix} r\omega^2 & \hat{d}(x, y, P_x, P_y) \end{pmatrix}^T & z &= \begin{pmatrix} a_c & d \end{pmatrix}^T \end{aligned} \tag{6.3}$$

Fig.6.2 clearly shows when a sonar measurement is used to correct the estimation. The sonars data is shown at Fig.C.2. The major corrections take place at the beginning and become invisible after two turns. By that time, the ω ("+sonars" curve on Fig.6.5) also converges towards the correct value and when we compare the shape of the curve with the previous configuration, it is obvious that the sonars give a major improvement. Compared with the previous model, the final value is already much closer to the "true" ω.

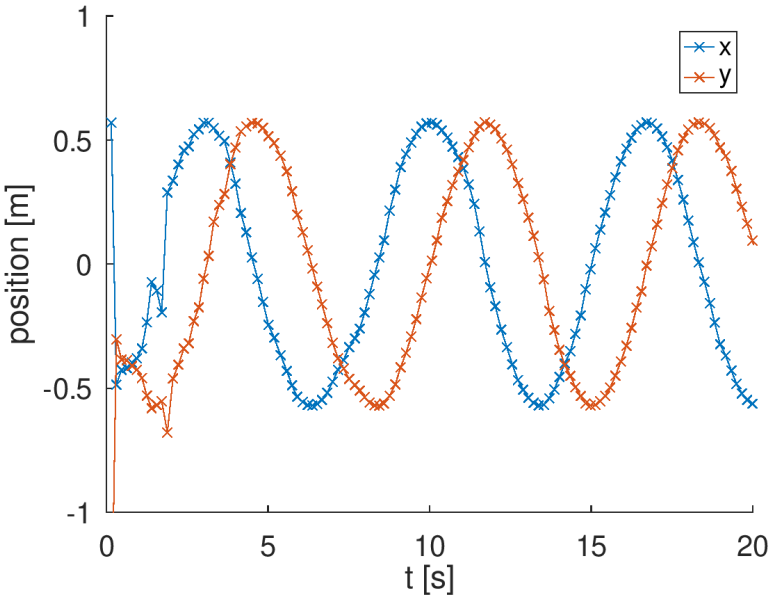

Figure 6.2: Estimation of the position

6.4 Adding a gyroscope

We can improve the angular velocity estimation with a gyroscope g. The h, z and R parameters in (6.3) are adapted in (6.4). $\sigma_g^2 = 0.005$ can be computed from the gyroscope data (Fig.C.3) because the angular velocity is constant.

$$h = \begin{pmatrix} r\omega^2 & \omega & \hat{d}(x, y, P_x, P_y) \end{pmatrix}^T \qquad z = \begin{pmatrix} a_c & g & d \end{pmatrix}^T$$
$$R = \text{diag}(\sigma_{a_c}^2, \sigma_g^2, \sigma_d^2) \tag{6.4}$$

The much cleaner gyroscope signal allows to reach a stable estimation in only 2 [s] instead of 15 [s] for the previous model. This time, the final value converges to the "true" value ("+gyro" curve on Fig.6.5). From this result, we learn the importance of having a gyroscope.

6.5 Radius estimation

Since the gyroscope provides information regarding the angular velocity, we can use the accelerometer to estimate the radius and therefore make the tracking more difficult. Of course, we keep the same data sources as the previous configuration, which are an accelerometer, a gyroscope, and two sonars.

The f, X and Q parameters in (6.3) are adapted in (6.5). Measuring the "true" radius is difficult because the train has a certain width and it is unclear on which point the system will converge. So, instead, we measure the radius of the inner circle $r_{in} = 0.57$ [m] and the outer circle $r_{out} = 0.615$ [m] of the railway track. The estimated radius should fall between these two bounds.

$$f = \begin{pmatrix} r\cos\theta & r\sin\theta & \theta + \omega\Delta t & \omega & r \end{pmatrix}^T$$
$$X = \begin{pmatrix} x & y & \theta & \omega & r \end{pmatrix}^T \qquad Q = O_{5\times5} \tag{6.5}$$

As you can see on Fig.6.3, it takes ≈ 5 [s] before the estimated radius stabilizes around the correct value. Later, the curve continues to oscillate slightly between the two bounds, but a bit less every time. These oscillations appear to be synchronized with the period of the circular motion and are probably due to the sonars systematically seeing different parts of the train. The estimation of ω ("+radius estimation" curve on Fig.6.5) shows that we are now solving a much harder problem since the convergence takes significantly longer.

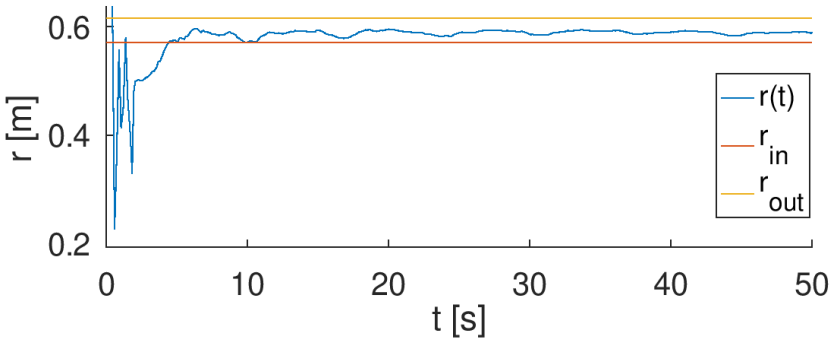

Figure 6.3: Estimation of r

6.6 Adding a magnetometer

A magnetometer provides information about the heading θ_m which can be computed from (6.6) where m_x and m_y are the measured magnetic field corrected for hard-iron bias. Fig.6.4 shows the raw magnetic field while the train is running on the railway. As you can see, the hard-iron bias is a constant offset applied on the magnetic field (see section 7.1.1 for more details).

$$\theta_m = \arctan 2(m_y, m_x) \tag{6.6}$$

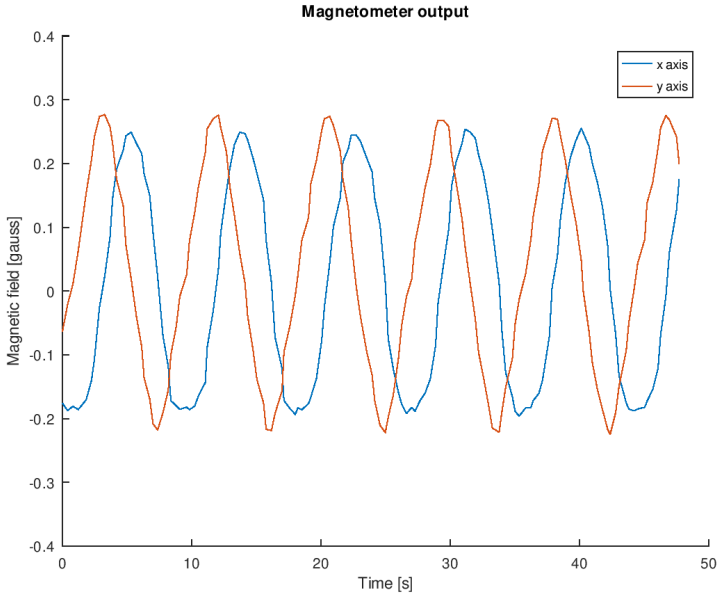

Figure 6.4: Measured magnetic field while the train is running on the railway

The hard-iron bias can be removed by a simple calibration:

1. Take two measures m_{x1}, m_{y1}
2. Turn the magnetometer 180° around the z axis
3. Take two measures m_{x2}, m_{y2}
4. Compute the bias $b_x = \frac{1}{2}(m_{x1} + m_{x2})$, $b_y = \frac{1}{2}(m_{y1} + m_{y2})$

The h, z and R parameters in (6.4) are adapted in (6.9) where $\theta^{'}$ is modified by (6.7). This allows $\theta^{'}$ to be compared with θ_m as described in [17] because of the wrapping effect of the SO(2) group (see section 2.4.1).

$$\begin{aligned} \theta^{'} &= (\theta \,\%\, 2\pi) - 2k\pi \quad \text{where} \\ k &= \underset{k\in\{0,1\}}{\operatorname{argmin}} \left(|\theta_m - \theta^{'} + 2k\pi| \right) \end{aligned} \tag{6.7}$$

$\sigma^2_{\theta_m} = 0.015$ cannot be computed directly from the magnetometer data because the heading is not a constant. However, we can compare it with the "true" heading obtained from the "true" ω and the known initial position θ_0 using (6.8).

$$\hat{\theta}(t) = \theta_0 + \omega t \tag{6.8}$$

$$h = \begin{pmatrix} r\omega^2 & \omega & \theta' & \hat{d}(x, y, P_x, P_y) \end{pmatrix}^T$$

$$z = \begin{pmatrix} a_c & g & \theta_m & d \end{pmatrix}^T \qquad R = \text{diag}(\sigma_{a_c}^2, \sigma_g^2, \sigma_{\theta_m}^2, \sigma_d^2) \tag{6.9}$$

Fig.C.4 shows a comparison of the heading computed from the magnetometer data and the Kalman filter estimation. The magnetometer gives very useful information and so we are able to restore the precision of the estimated ω ("+mag" curve on Fig.6.5) to what we achieved before adding the radius as state variable.

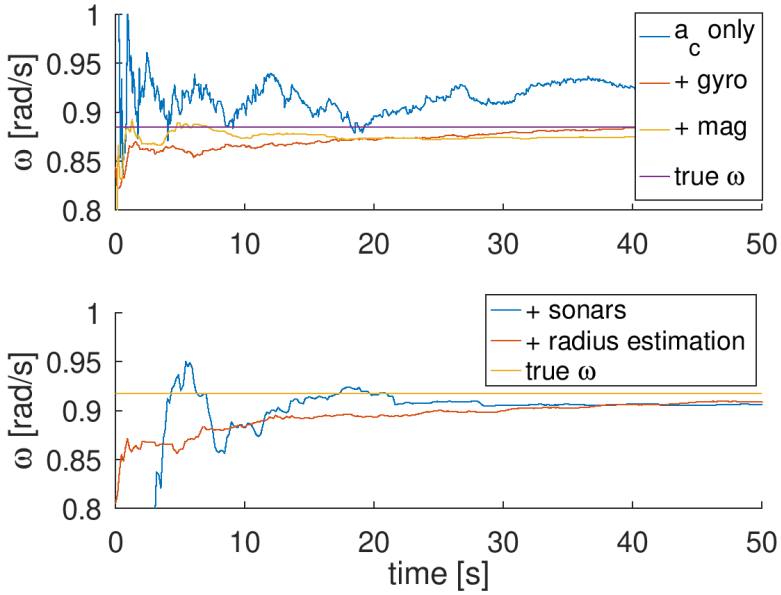

Figure 6.5: Estimation of ω

6.7 Verdict

For us, the conclusion is clear. Sensor fusion with a Kalman filter (EKF in this case) using GRiSP and Erlang is not only viable, but is also able to match soft real-time expectations. The early results showed that some time was required to get a decent estimation of the system state, but with the help of valuable sensors, like a gyroscope or a magnetometer, we are able to significantly lower that delay. From those results, we also infer a graceful degradation of the sensor fusion quality under hardware failure because this would be similar as removing sensors.

Chapter 7

Second phase: a six degrees of freedom inertial measurement unit

7.1 Orientation tracking with an AHRS

In the first phase, we analyzed the feasibility of sensor fusion with Hera and GRiSP. In the second phase, we developed a more useful application: an attitude and heading reference system (AHRS).

Orientation tracking requires to combine information coming from multiple sensors and is typically performed at a high frequency. State of the art systems achieve this by working very close to the hardware which adds a lot of complexity to an already non-trivial problem. Moreover, working close to the hardware makes it difficult to provide a distributed and fault-tolerant system. In this section, we show that it is possible to get a decent orientation tracking at a much higher level of abstraction. Due to sensor drivers and processor limitations, the update frequency is bounded to ≈ 3.75 [Hz].

7.1.1 Calibration of the magnetometer

In the previous chapter, we have seen how a magnetometer can be used to find the orientation in two dimensions, but tracking the orientation in three dimensions is different. Before explaining how to do it, we first need to examine the magnetic field in three dimensions to calibrate the magnetometer. If we plot the magnetic field vector $\begin{pmatrix} m_x & m_y & m_z \end{pmatrix}$ while turning the sensor in all directions, we should obtain a sphere. However, as you can see on Fig.7.1 and Fig.7.2, the raw output (in blue) is not centered. This distortion is call "the hard-iron bias" and it is due to

the presence of an additional[1] magnetic field near the sensor and generated by the electrical components of the board or by permanently magnetized ferromagnetic components. We can correct this bias with a calibration:

1. Take 3 measures m_{x1}, m_{y1}, m_{z1}
2. Turn the magnetometer 180° around the z axis
3. Take 2 measures m_{x2}, m_{y2}
4. Turn the magnetometer 180° around the x axis
5. Take 1 measure m_{z2}
6. Compute the bias $b_a = \frac{1}{2}(m_{a1} + m_{a2})$, where $a \in \{x, y, z\}$

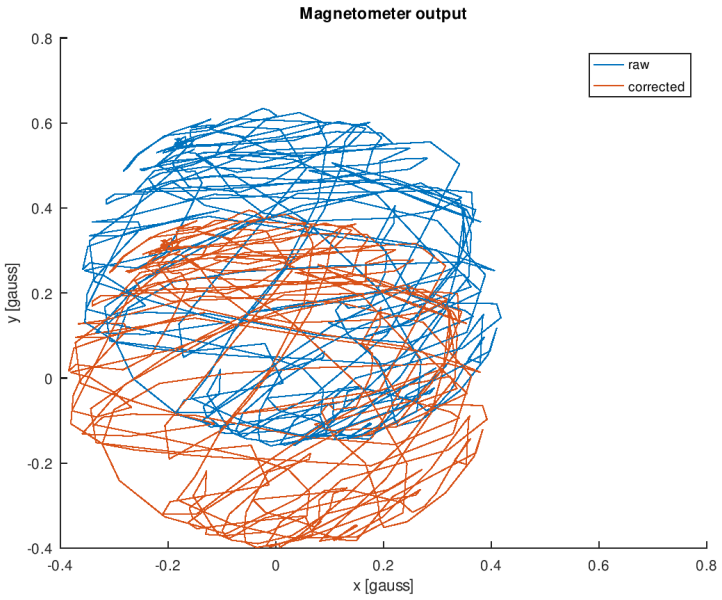

Figure 7.1: Magnetic field as we turn the sensor in all directions (XY)

[1]We are interested in the Earth's magnetic field only because we use it as a reference to find our current orientation. Much like the way a compass works.

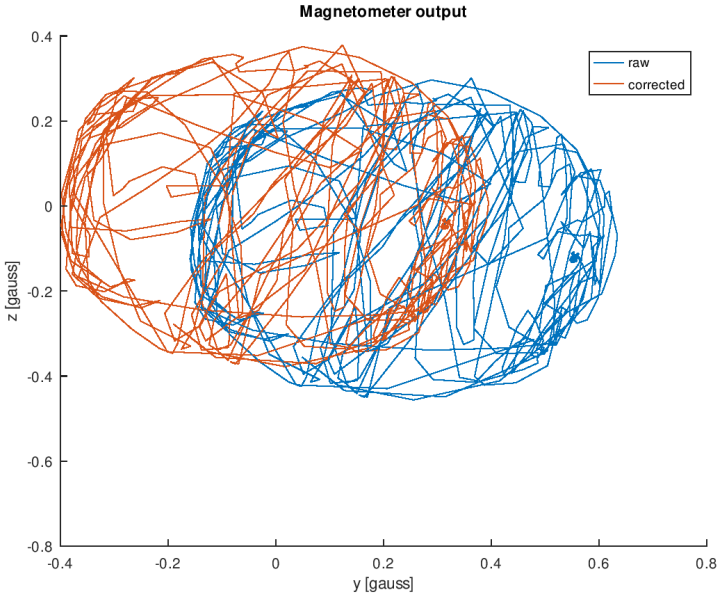

Figure 7.2: Magnetic field as we turn the sensor in all directions (YZ)

Another distortion that we observe it that the "sphere" is a bit squashed and is in fact, an ellipse. This distortion is call "the soft-iron bias" and is due to the presence of metal, near the sensor, that alter the magnetic field. Again, this is perfectly normal since the board is made of metal components. We can correct this distortion by mapping the ellipse to a sphere by multiplying the measured magnetic field with a 3×3 matrix. The application note [22] gives all the necessary information to perform this calibration. However, the soft-iron bias has a much lower impact than the hard-iron bias and because of limited time, we decided not to do it for a first implementation. Still, this could only be beneficial for a more accurate orientation tracking.

7.1.2 Orientation estimation with 3-axis accelerometer, 3-axis magnetometer, and 3-axis gyroscope

We can determine the orientation from the direction of gravity and the north [6] using an accelerometer and a magnetometer. When there is no linear acceleration, the accelerometer measures gravity. The magnetometer m points to the magnetic north which is not perfectly parallel to the ground. Therefore, we can extract the vectors of interest with cross products and obtain the orientation in the form of a direct cosine matrix (DCM) a.k.a rotation matrix (7.1).

$$
\begin{aligned}
\text{East} &= \hat{\text{Down}} \times \hat{m} \\
\text{North} &= \hat{\text{East}} \times \hat{\text{Down}} \\
\text{DCM} &= \begin{pmatrix} \hat{\text{North}}^T & \hat{\text{East}}^T & \hat{\text{Down}}^T \end{pmatrix}
\end{aligned} \tag{7.1}
$$

As shown in [24], it is possible to build a Kalman filter with the DCM. However, we will use the quaternion representation because this approach requires less operations. We cannot use Euler angles because they are ambiguous [2]. The DCM can be converted into a quaternion with (7.2) [10].

$$
\begin{aligned}
q_1^2 &= \frac{1}{4}(1 + R_{11} + R_{22} + R_{33}) \\
q_{am} &= \frac{1}{4q_1} \begin{pmatrix} 4q_1^2 \\ R_{32} - R_{23} \\ R_{13} - R_{31} \\ R_{21} - R_{12} \end{pmatrix}
\end{aligned} \tag{7.2}
$$

The quaternion-based Kalman filter (7.4) is inspired from [11]. From this point, we can add the gyroscope data in the Kalman filter. Injecting the gyroscope signal directly into the state transition matrix F has the benefit of making the system more reactive to brutal changes. This is extremely important considering the low sampling frequency. Another way would be to include ω in the state, but since we do not know how it changes, a constant hypothesis would be chosen which would result in an overall less reactive model. The variances $\sigma_Q^2 = 10^{-3}$ and $\sigma_R^2 = 10^{-2}$ were defined by trial-and-error and we did not try to optimize these parameters as much as possible because we are only focus on demonstrating the feasibility.

$$
\Omega(\omega) = \begin{pmatrix} 0 & \omega_x & \omega_y & \omega_z \\ -\omega_x & 0 & -\omega_z & \omega_y \\ -\omega_y & \omega_z & 0 & -\omega_x \\ -\omega_z & -\omega_y & \omega_x & 0 \end{pmatrix} \tag{7.3}
$$

$$
\begin{aligned}
F &= I_{4x4} + \frac{\Delta t}{2}\Omega(\omega) & & \\
H &= I_{4x4} & Z &= q_{am} \\
Q &= \sigma_Q^2 I_{4x4} & R &= \sigma_R^2 I_{4x4}
\end{aligned} \tag{7.4}
$$

[2] The same orientation can be represented with different combinations.

However, the sign of the predicted quaternion $\hat{q}^-$ may be changed such that (7.5) is valid, to ensure that the difference computed by the Kalman filter is the smallest (see section 2.4.3).

$$q_{am} \cdot \hat{q}^- > 0 \tag{7.5}$$

Moreover, because the $\Delta t \Omega(\omega)$ term is not always constant between two iterations, the estimated quaternion requires a normalization to fix the error introduced by the Taylor expansion.

7.1.3 Validation of the AHRS

A proper analysis of an AHRS demands equipment, like a high-precision tri-axis turntable, that we do not have. Instead, we created 3 manually reproducible benchmark tasks to validate the model: a slow 360° rotation, a fast 360° rotation, and shaking without rotation.

The "shaking" benchmark was performed concurrently on a computer[3] because the shaking motion could not be reproduced 3 times with enough similarities.

Quantitative demonstration whether the estimation is smooth and correct is difficult. During the experiments, we could visualize the rotation in real time with our modular visualization tool (Fig.7.3). However, to report our results, we represent the orientation by showing the coordinate of the point $(1, 0, 0)$ from the reference frame to the body frame.

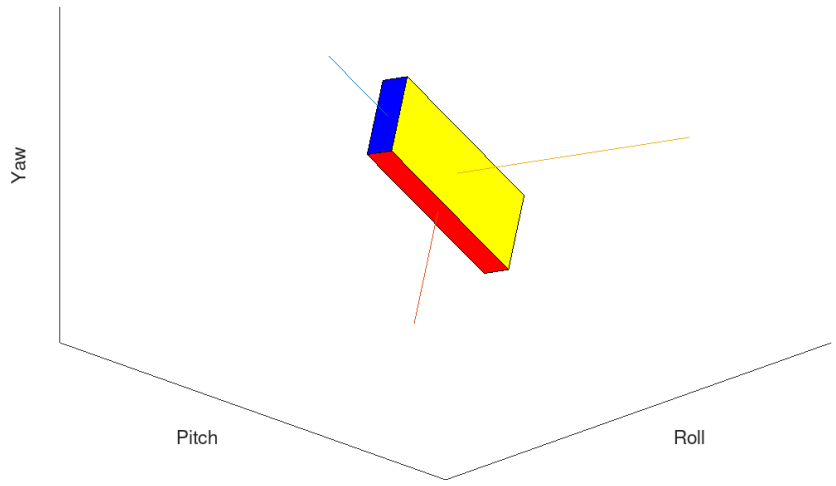

Figure 7.3: Orientation in soft real-time with LiveView

[3]And not on the GRiSP board because it does not have enough computational power for the three models at the same time.

On Fig.7.4 and Fig.7.5, you can see the orientation provided by the DCM from 7.1 (without the gyroscope). It is shaky and not smooth, especially when we move faster.

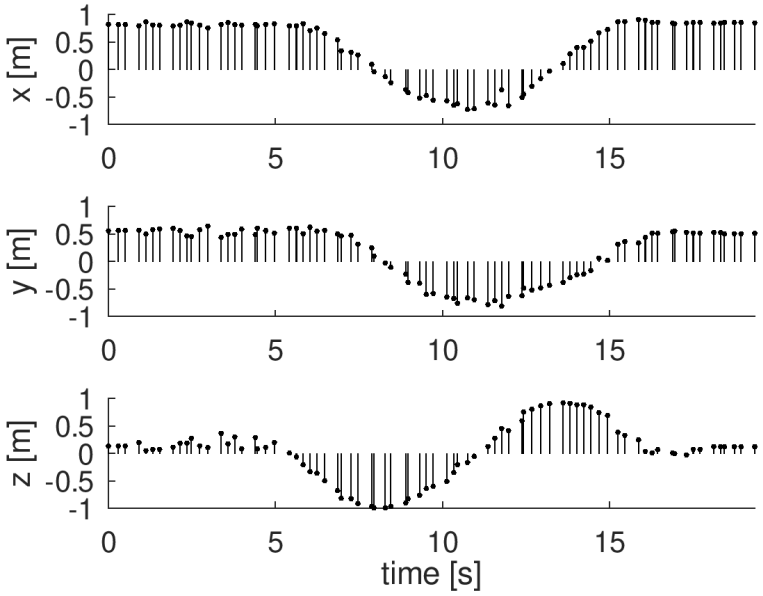

Figure 7.4: Orientation from the accelerometer and the magnetometer, slow rotation

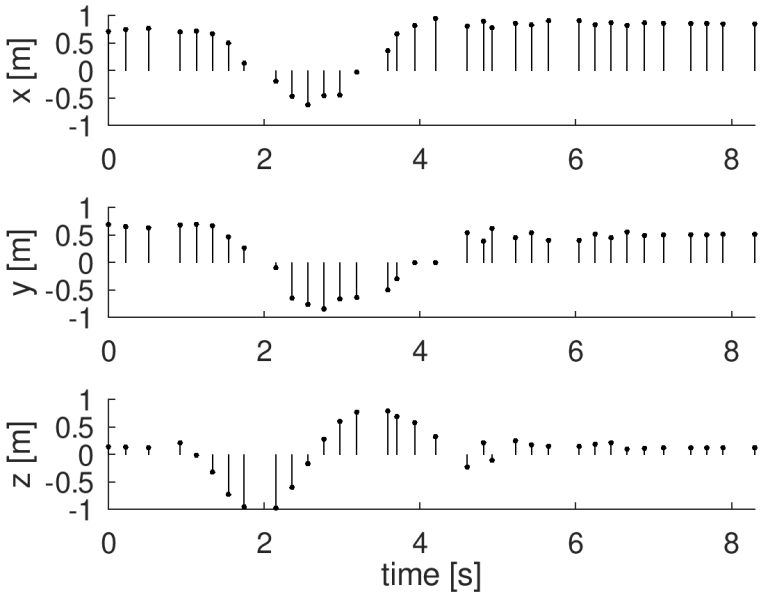

Figure 7.5: Orientation from the accelerometer and the magnetometer, quick rotation

Fig.7.6 and Fig.7.7 give the pure inertial orientation we get from the prediction model of the quaternion-based Kalman filter, thanks to the gyroscope signal but without the accelerometer and the magnetometer. The gyroscope really helps to predict the next orientation with a lot of smoothness. However, after a complete 360° rotation, we can see that it is subject to an important drift.

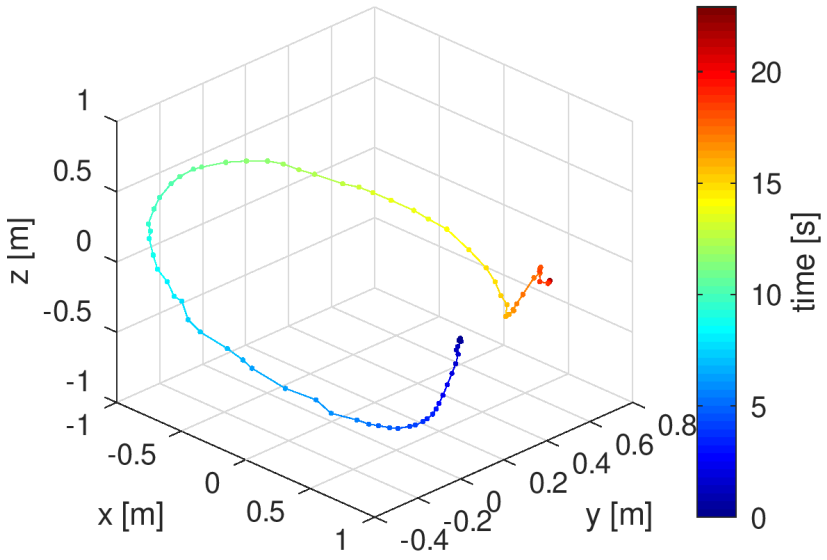

Figure 7.6: Orientation with a gyroscope only, slow rotation

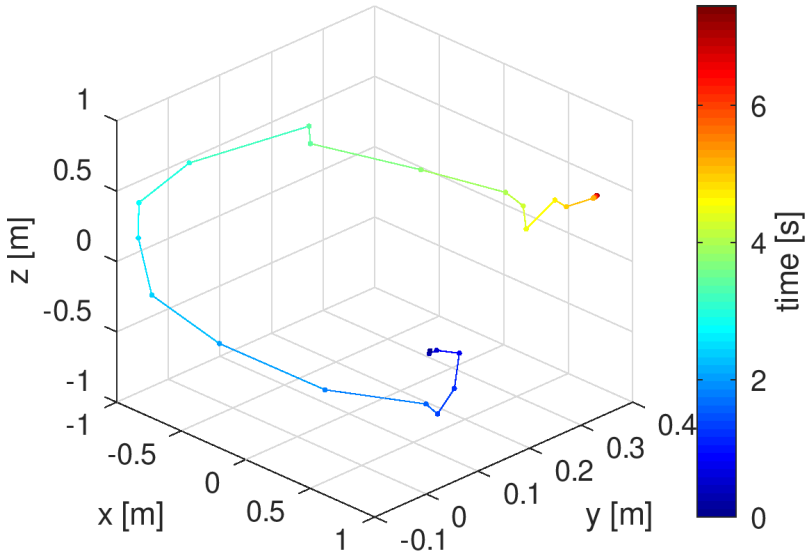

Figure 7.7: Orientation with a gyroscope only, quick rotation

It is clear from Fig.7.8 and Fig.7.9 that the Kalman filter allows to combine the absolute information from the magnetometer and accelerometer and the smooth

inertial prediction from the gyroscope to get the best of both without the disadvantages.

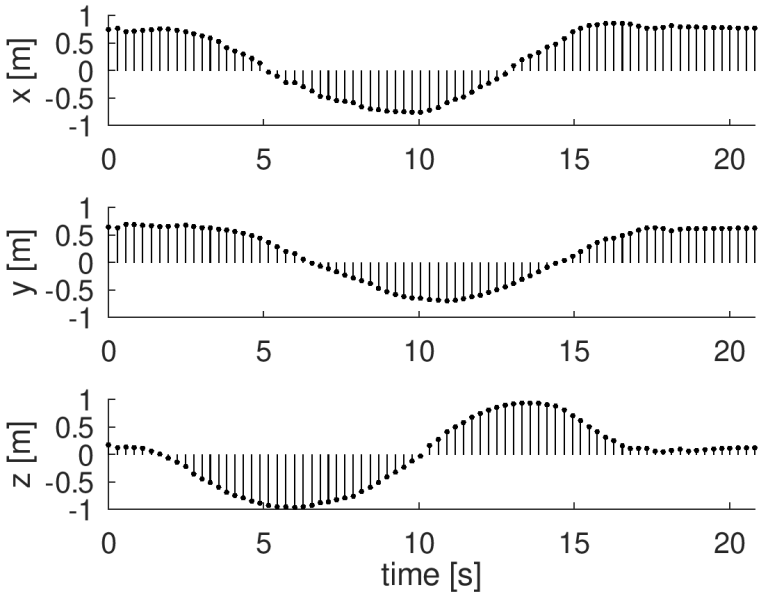

Figure 7.8: Orientation from the fusion of accelerometer, magnetometer, and gyroscope via Kalman filter, slow rotation

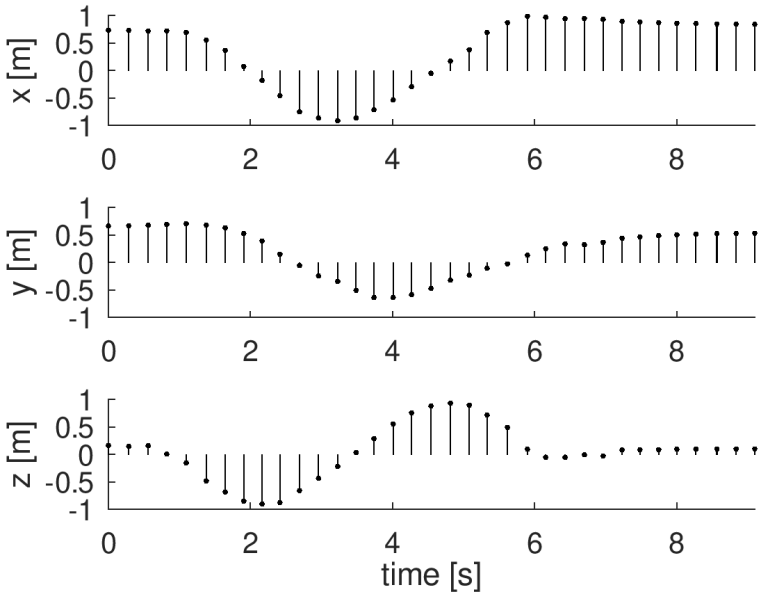

Figure 7.9: Orientation from the fusion of accelerometer, magnetometer, and gyroscope via Kalman filter, quick rotation

Since we assume the accelerometer gives the direction of the floor, the computation is incorrect when the device is subject to an acceleration. As a result, while under a shaking motion, the DCM from (7.1) indicates a changing orientation even when there is none in reality. Fortunately, the gyroscope is not subject to that issue and thus, it should help to reduce this effect. Fig.7.10 reports the "deviation from previous position" which is the normalized distance between two consecutive positions. For example, if the orientation goes from 0° to 360° and then 180° in two iterations, we compute 100% and then 50%. The graph confirms the effect we described with a spike reaching the 80% for the DCM. We also see that the pure inertial method is not completely spared with a deviation going up to 20%. It means there is a small angular velocity $\delta\omega$ picked up by the gyroscope which can lead to a visible deviation after the integration by $\theta_0 + \delta\omega\Delta t$. The Kalman filter is sometimes higher and other times a bit lower, but overall fairly close. In any case, the deviation never exceeds the 20% mark.

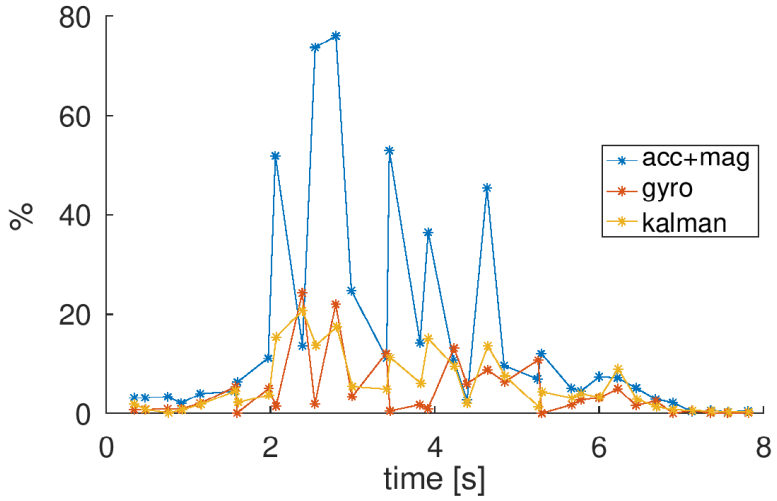

Figure 7.10: Deviation from previous position while shaking

7.2 Position tracking with 3 sonars

In section 3.4, we have seen how to combine an accelerometer with a sonar in order to cancel the drift of inertial navigation. In this section, we briefly revisit this method in a simplified way to ensure that we can provide an accurate position tracking in three dimensions with only three sonars without accelerometer, over a small area.

If the target is not too fast, we can assume a constant position and design a simple Kalman filter to fuse the data coming from multiple sonars (7.6). We have set $\sigma_p = 5$ [cm] and $\sigma_s = 10$ [cm]. The first is a small error on the constant assumption and the second represents the size of the GRiSP board with its support[4].

$$\begin{aligned} F &= I_{3x3} & H &= I_{3x3} \\ Q &= \sigma_p^2 I_{3x3} & R &= \sigma_s^2 I_{3x3} \end{aligned} \tag{7.6}$$

Fig.7.11 shows the position tracking achieved with three sonars aligned with the x,y,z axes. You can see how we placed the sonars on Fig.B.3. We tried a back-and-forth motion along one axis at a time. The sonars allow to get a high accuracy, but cannot follow a quick motion because the Kalman filter model assume the position is constant ($\pm\sigma_p$) and because the sonars are only measuring one after the other to avoid cross-talk.

[4]We have used the same kind of wooden support as for the sonars (Fig.B.2).

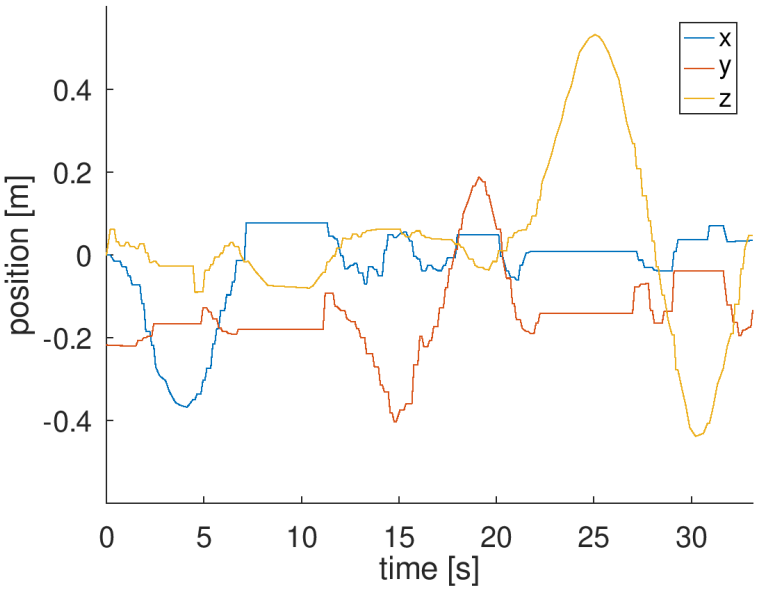

Figure 7.11: Position tracking with sonars

7.3 Combining orientation and position

Now that we have a working orientation tracking with accelerometer, magnetometer and gyroscope, and that we can track the position on 3 axis with sonars, we can try to combine both. As explained previously, the major problem with the AHRS is that we rely on the accelerometer to get the direction of the floor. When there is a linear motion, the acceleration measured by the accelerometer contains both gravity and the linear acceleration. As a result, we can no longer measure the down direction. For example, if the board is in a car that accelerates, then the estimated orientation would be a bit bias since we directly use the accelerometer to find the direction of the floor (gravity).

From [6], we know that one solution may be to separate the linear acceleration from gravity, allowing to recover the direction of the floor. We can accomplish this with 2 Kalman filters: the first one, an inertial navigation-based Kalman filter as explained in chapter 3 and the second one, a quaternion-based Kalman filter as explained in section 7.1.2. By merging the two, we obtain a 6 DOF IMU.

As you can see on Fig.7.12, first, we process the accelerometer data by expressing the acceleration $\vec{a}'_k$ in the reference frame with the previously estimated orientation R_k (7.7). Then, we extract the linear acceleration $\vec{a}_{l_k}$ by removing gravity (7.8). Next, we feed it, along with the sonar measurements, to the first Kalman filter (the inertial navigation model). After, we use the corrected linear acceleration $\vec{a}_{l_{k+1}}$,

computed by the model, to transfer it back to the body frame (7.9) and finally we subtract it from the accelerometer measurement in order to obtain the floor direction (7.10). Once this information is retrieved, we feed it into the second Kalman filter (orientation estimation model) with the magnetometer and gyroscope measurements to update the orientation.

$$\vec{a}_k = \vec{a}'_k R_k^T \tag{7.7}$$

$$\vec{a}_{l_k} = \vec{a}_k - \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{7.8}$$

$$\vec{a}'_{l_{k+1}} = \vec{a}_{l_{k+1}} R_k \tag{7.9}$$

$$\vec{g}'_{k+1} = \vec{a}'_k - \vec{a}'_{l_{k+1}} \tag{7.10}$$

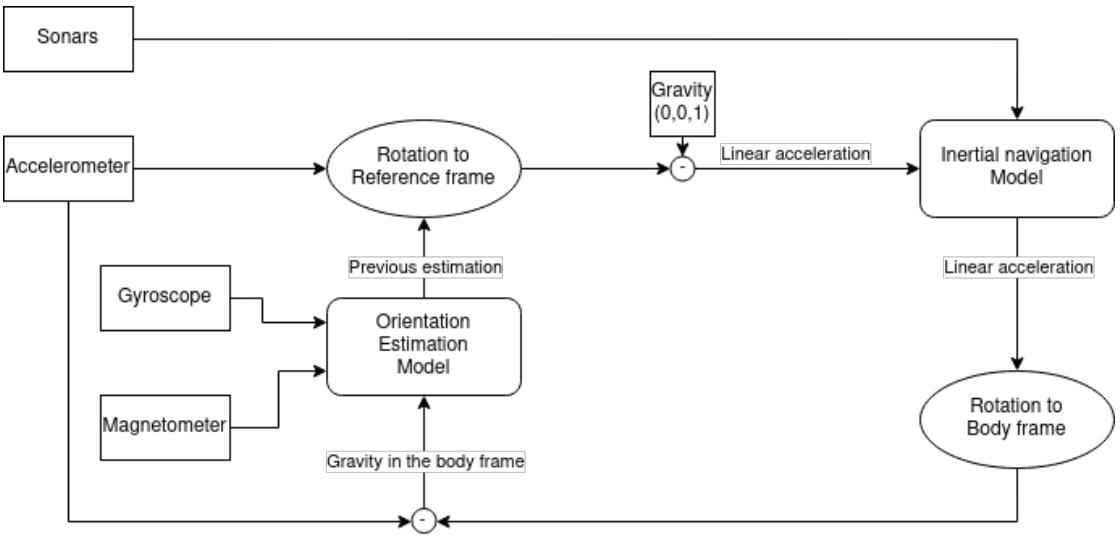

Figure 7.12: The 6 DOF IMU

Unfortunately, despite our attempt to recover the direction of the ground by removing the linear acceleration, the vector representing the direction of the floor never gets totally clean. Indeed, instead of being close to $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$, a bias was present on each axis and creates problems for the orientation estimation. The idea was good, but some limitations prevented us from obtaining good results.

7.4 Verdict

We have shown that orientation and position tracking in 3 dimensions with inertial navigation and sonars can be achieved with Hera and the GRiSP platform.

In chapter 3 we have shown how an accelerometer can be combined with a sonar for position tracking along one axis and in this chapter, we have shown that tracking in 3 dimensions can also be achieved with 3 sonars only. Despite a low frequency of 3.75 [Hz], the orientation tracking is surprisingly smooth because of the inertial model (gyroscope) and do not suffer from drift because of the absolute reference system (magnetometer and accelerometer).

Even though we could not directly combine orientation and position tracking into a single model, alternatives can be found. Indeed, it is still possible to use the approach of section 3.4 for non-rotating targets. In case the target is rotating, one can independently perform position tracking with just sonars and use our AHRS to determine their orientation.

This limitation is caused by an accumulation of minor problems.

First, the system[5] is too slow. With the orientation tracking only, we could still have a decent update frequency, but now, the computational power required for the merged model has increased too much because of the operations for the inertial navigation and the additional traffic generated by the three sonars. Nevertheless, for testing purposes, we ran the full 6 DOF IMU on a computer, considered as part of the network.

Second, the vector representing the floor direction is close to $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$, but never gets totally clean. In addition to errors directly related to the noisy measurements, changing the acceleration vector from one frame to another leads to an accumulation of error. Since we reuse the, slightly off, previous orientation estimate to transfer the acceleration vector from the body frame to the reference frame, the result can only be as accurate as the previous estimate. When the GRiSP board was standing still on a table, we observed that the linear acceleration derived from the subtraction of the acceleration by gravity (in the reference frame) gets close to $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$, but is never perfectly equal to zero as it should, when the device does not move. Of course, the same problem occurs when we transfer the corrected linear acceleration from the reference frame to the body frame.

Third, as already mentioned in section 7.1.1, we did not correct the soft-iron bias. Since the system is too slow anyway, we decided that the effort was pointless, but we suspect that the soft-iron bias could also be a source of error.

Finally, there is always a small uncertainty about the exact moment at which a measure is taken compared to the moment at which we do the computation with it. This small time delay can also lead to small errors added to the ones already present. Because the inertial navigation relies on integration, a small error on Δt can actually quickly generate a lot of drift. The best way of solving this problem is

[5]The GRiSP board, the Pmod NAV driver and the matrix library written in pure Erlang.

simply to increase the update frequency of the system, but at the moment we have already reached the limit.

Again, the limitations are minor and only prevent us to accurately estimate the orientation while the target is moving, i.e. accelerating (linearly). We think that the problem could be resolved with performance improvements since it would increase the precision by reducing the impact of approximations, and last but not least, it would allow us to run the full 6 DOF IMU directly on the GRiSP board instead of using a computer.

Chapter 8

Conclusion

8.1 Results

Throughout this master thesis, we showed that sensor fusion at the extreme edge of an IoT network is not only feasible, but also surprisingly efficient. The GRiSP environment is an excellent choice for IoT applications because it provides a WiFi antenna for communication and sufficient performance for simple sensor fusion. Moreover, GRiSP makes prototyping easy with the use of Digilent Pmod sensors, drivers and Erlang/OTP. With these tools, we developed Hera, a fault-tolerant and distributed framework for asynchronous sensor fusion. We analysed the fault-tolerance of Hera by fault injection and we conducted experiments on sensor fusion with Kalman filters to test Hera on the GRiSP platform. The results were very encouraging and so, we built an attitude and heading reference system (AHRS) as well as a 6 degrees of freedom (DOF) inertial measurement unit (IMU). While the latter requires major performance improvements, the former gave astonishing results despite reaching the limits of the system.

Inertial navigation and tracking: We started this master thesis by exploring inertial navigation and tracking. We concluded that this technique could be used on the GRiSP platform and we showed that a Kalman filter-based sensor fusion with an accelerometer and a sonar has the potential to remove drift while increasing the overall precision.

Hera: We decided to redesign the Hera framework and we tested it on a network of GRiSP boards. Our full system, which consists of both hardware and software, is: asynchronous, dynamic, fault-tolerant, modular, and soft real-time. It allows nodes to join or leave the network without problem while executing sensor fusion with easily modifiable modules. The built-in Kalman filters perfectly fit these properties

because, helped with our matrix library, they can achieve asynchronous sensor fusion of noisy data. Thanks to the asynchronous computation model, the dynamic nature of the system, and a restarting strategy, we made the system fault-tolerant.

Fault-tolerance analysis: To analyse the fine-grained behaviour of Hera under failure, we designed small software agents running on Hera itself with the purpose of observing the availability and the resilience of our framework. We performed fault injection on the different parts of the system and explained the observed results. The conclusion is positive. Hera keeps working even in the presence of failures, as long as one board is running. We also showed that the distributed mutex or synchronization extension presents a weak spot in case of hardware failure (network partition, power failure, ...) because it takes up to 10 [s] for the system to recover. Fortunately, this is restricted to the extension only and the rest of the system works perfectly.

The experimental model: With a train toy on a circular path, we showed that it is possible to combine different sensors (accelerometer, sonar, gyroscope, and magnetometer) with a Kalman filter to achieve high precision tracking. We also proved that sensor fusion using Hera on a network of GRiSP boards is viable and matches soft real-time expectations.

AHRS and 6 DOF IMU: Finally, we implemented an AHRS as well as a 6 DOF IMU. The AHRS, with the quaternion-based Kalman filter, exceeds our expectations despite a low sampling frequency. Not only is the orientation estimation correct compared to reality, but it also comes along with a relatively smooth motion tracking without latency. Our empirical tests showed that the system reacts well to brutal changes thanks to the gyroscope in the Kalman filter and the error does not exceed 20% when the board experiences shaking.

We also showed how sonars can be used for position tracking in 3 dimensions. We could not estimate the orientation while the board is under a continuous linear acceleration that prevents us from recovering the gravity vector.

There are still possibilities for future work to combine orientation tracking and position tracking with an inertial model. However, at the current time, we believe that the system (GRiSP hardware, Pmod NAV driver, and numerical computation with Erlang) is too slow and improvements should be brought before further attempts.

8.2 Performance improvements

The Hera framework shows that low-cost, accurate, and fault-tolerant sensor fusion can be achieved directly at the edge. The current implementation pushes the GRiSP computation power and Pmod sensor drivers to their limit. Future improvements in GRiSP, Pmod and Erlang numeric computation performance will be directly usable to achieve a higher update frequency and increased accuracy. These improvements are very important for the full 6 DOF IMU because the current system is simply too slow for it.

8.2.1 GRiSP 2

The current system is limited by the low clock frequency of the GRiSP-base boards resulting in a low computation speed, by the low measurement frequency of the Pmod drivers and by the use of Erlang for matrix computation. Despite these limitations, it provides good accuracy and is fast enough for soft real-time applications. The GRiSP 2, a second-generation GRiSP board, is planned to become available in June 2021 and will provide a 10× improvement in computation speed.

8.2.2 A NIF matrix library

A native matrix library is being implemented as part of a separate master thesis [16]. It consists of native implemented functions (NIF) for Erlang and specifically compiled for the GRiSP platform. According to preliminary results, this much more efficient library should provide an additional 10× to 100× improvement in speed for the matrix operations of the Kalman filter.

8.2.3 A new driver for the Pmod NAV

GRiSP already provides Erlang drivers for many Pmod sensor modules. Simply having to plug a sensor and directly benefiting from an existing driver is incredibly useful when it comes to prototyping or quick development. We greatly appreciated the Pmod NAV driver and it seems quite complete, but we think that some improvements could be done:

- It is not possible to read registers from different components (accelerometer, magnetometer or altimeter) at the same time.
- Requests are treated synchronously which implies that we cannot access different registers concurrently (i.e. requests are being serialized).

- In case multiple registers are read, we could also want a timestamp along with each value to have a more accurate information in order to increase the tracking precision.

However, there is a much more important problem with the current driver: it is too slow. During a discussion with Peer Stritzinger, we learnt that the Pmod NAV could be running in continuous mode and supports interrupts to read the 32 last values at once. Since the current driver does not allow that, a new version is planned for summer 2021. This new version has a lot of potential and we hope to gain a significant boost in sampling rate.

8.3 Event-based computation: an alternative design for Hera

In the current model, the sensor fusion engine[1] has its own internal loop that begins by fetching data from the local data store before performing an iteration of the Kalman filter. This creates unnecessary "spinning" when no new data has arrived since the previous iteration. We managed to keep that under control with a parametric delay[2] between each iteration, to avoid overloading the system.

Another way we could approach this issue is by triggering a computation upon data reception. We decided to keep the same model as the original version of Hera because this method was proven successful. Nevertheless, we think the event-based version could be a better option. The local data store could be replaced by an event manager and the computation modules could become handlers running in their own process. The downside of this simplified approach is that the event manager could start lagging behind if the computation is too slow. Therefore, some kind of flushing would be necessary to prevent the accumulation of similar events. In short, if two data coming from the same source are not treated yet, the oldest one should be discarded. This is somewhat similar to what we did with the local data storage except that currently, we do not trigger any computation upon reception of data.

8.4 Future work

We now give some of the possibilities for future work based on Hera.

[1] An `hera_measure` process.

[2] During which the process is sleeping.

8.4.1 Orientation estimation under magnetic distortions

Magnetic distortions are very likely to occur in a real-life scenario like a person moving inside a building. During our experiments, we noticed the great sensitivity of the magnetometer. For example, if we were to place the sensor on a table supported by metal beams, then the computed orientation would be completely wrong. Therefore, developing an AHRS resilient to magnetic distortions could be very useful. In the recent years, multiple works tried to provide accurate orientation estimation despite the presence of magnetic distortions. The approach of [11] could be used as a starting point for future work in this domain.

8.4.2 Multi-target tracking

In section 3.5, we briefly talked about multi-target tracking which involves track maintenance and data association. In section 1.4, we also cited the theoretical study [14] that could be used for future work in this field.

8.4.3 Modeling complex trajectories of maneuvering targets

At multiple occasions, we mentioned that the major problem of inertial navigation is drift caused by an accumulation of small errors, amplified by integration. A naive model that assumes a constant acceleration can only work if the frequency of the system is high enough with respect to the rate at which the acceleration actually changes.

Position tracking for completely arbitrary motion with inertial navigation requires a frequency of multiple hundreds of Hertz that the current Pmod sensor drivers cannot provide. In a simplified scenario and at 17 [Hz], we demonstrated that good results are possible, but in this precise case, the tracking does not react well to brutal changes in acceleration (i.e. the acceleration is not constant).

In low frequency systems, good results can still be achieved with more accurate models. We illustrated this in appendix D and this approach proved to be very successful in chapter 6. However, a more accurate model cannot cope with complex trajectories[3] and this is precisely what the IMM algorithm is made for (section 3.5). The algorithm uses a Markov chain to dynamically select the most appropriate model at any time instant and is therefore suited to cope with discontinuities in the target trajectory. We believe that with the planned performance improvements, it would be very interesting to experiment on this topic with Hera.

[3] A composition of different types of simple trajectories like a constant turn, a brutal turn, a constant linear acceleration, a constant velocity motion, ...

8.4.4 A hand-over system for large coverage

The current system is designed to run in a small cluster which is a set of fully connected nodes in the same environment. The nodes do not need to be physically close to each other, but this is probably a requirement in order for multiple sensors to observe the same region (i.e. environment). Because of that, Hera cannot be used to cover a large area.

The idea of a hand-over system, is to maintain the identity of an object that moves between clusters. For example, Hera could be used to track the position of a person equipped with a Pmod NAV inside of a building where each room would have its own cluster. It would then be interesting to transfer the node to the next cluster as the person leaves the room and enters in sight of other sensors. Concretely, the system should automatically handle the transition from a cluster A to a cluster B, as soon as data coming from sensors of the cluster A can no longer be associated to the moving node. Of course, a transition implies to disconnect the node from the previous cluster. Such a hand-over mechanism would allow Hera to be used in large coverage applications.

8.4.5 Combination with machine learning and data mining

Libraries for machine learning and data mining give new abilities, such as object recognition, which can be added to Hera in a straightforward manner. From the Hera viewpoint, these libraries can be implemented as `hera_measure` behaviours in the same way we implemented the different Kalman models or completely replace the Kalman engine depending on the needs. It is clear that these libraries require increased computation speed, but this should become possible with the planned improvements in speed.

8.4.6 Targeting rugged terrains

Because of its high fault tolerance, the Hera framework is suitable for IoT experiments in a rugged real-world terrain. Future work can use Hera for IoT prototyping in such situations.

8.4.7 Controlling physical devices

The current version of Hera does not attempt to control a physical device. Adding control is a straightforward extension of the Kalman filter computations in the sensor fusion engine. There exists Pmod actuators for controlling the external world, like proportional motor control.

8.5 Final word

Simple and easy to use, the Hera framework is perfect for IoT prototyping. In addition to its very interesting properties, it offers a good basis for sensor fusion at the extreme edge. Indeed, despite the low sampling frequency and the performance limitations, we obtained good experimental results.

A user manual can be found at appendix F and the complete software along with all the results presented in this master thesis are available on Github[4]. Hera is also freely available as open-source software on Github[5,6] and can be used with GRiSP-base boards from Stritzinger GmbH. Our software is a base that can be used for many improvements and extensions in fields like surveillance, tracking, and games. We hope that Hera will be used for both IoT education and product development.

[4] `https://github.com/sebkm/sensor_fusion`
[5] `https://github.com/sebkm/hera`
[6] `https://github.com/sebkm/hera_synchronization`

Bibliography

[1] Ienkaran Arasaratnam and Simon Haykin. Cubature Kalman Filters. *IEEE Transactions on Automatic Control*, 54(6):1254 – 1269, 2009.

[2] Federico Castanedo. A Review of Data Fusion Techniques. *The Scientific World Journal*, 2013.

[3] Puput Dani Prasetyo Adi and Akio Kitagawa. ZigBee Radio Frequency (RF) Performance on Raspberry Pi 3 for Internet of Things (IoT) based Blood Pressure Sensors Monitoring. *International Journal of Advanced Computer Science and Applications*, 10(5):10, 2019.

[4] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15):35, 2006.

[5] Digilent. *Pmod^{TM} Interface Specification*. National Instruments, 2020.

[6] Brian Douglas. Understanding Sensor Fusion and Tracking, Part 2: Fusing a Mag, Accel, and Gyro to Estimate Orientation. https://www.mathworks.com/videos/sensor-fusion-part-2-fusing-a-mag-accel-and-gyro-to-estimate-orientation-1569411056638.html, 2019.

[7] John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. *GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations*, 2020. URL `https://www.gnu.org/software/octave/doc/v5.2.0/`.

[8] David Eberly. *Quaternion algebra and calculus*. Magic Software Inc, 2002.

[9] David Eberly. *Euler Angle Formulas*. www.geometrictools.com, 2020.

[10] Jay A Farrell. Computation of the Quaternion from a Rotation Matrix. Technical report, University of California, 2015.

[11] Kaiqiang Feng, Jie Li, Xiaoming Zhang, Chong Shen, Yu Bi, Tao Zheng, and Jun Liu. A new quaternion-based Kalman filter for real-time attitude estimation using the two-step geometrically-intuitive correction algorithm. *Sensors*, 17(9):6, 2017.

[12] Anthony F. Genovese. The Interacting Multiple Model Algorithm for Accurate State Estimation of Maneuvering Targets. *Johns Hopkins APL Technical Digest*, 22(4):614–623, 2001.

[13] Felix Govaers. *Introduction and Implementations of the Kalman Filter*. IntechOpen, 2019.

[14] Junjun Guo, Xianghui Yuan, and Chongzhao Han. Sensor selection based on maximum entropy fuzzy clustering for target tracking in large-scale sensor networks. *IET Signal Processing*, 11(5):613–621, 2017.

[15] LightKone. Lightweight Computations on the Edge, 2020. URL `https://www.lightkone.eu/`.

[16] Tanguy Losseau. Concurrent Matrix and Vector Functions for Erlang. Master's thesis, UCLouvain, 2021 (to appear).

[17] Ivan Marković, Josip Ćesić, and Ivan Petrović. On wrapping the Kalman filter and estimating with the SO(2) group. In *19th International Conference on Information Fusion*, page 2, Heidelberg, Germany, 2016. ISIF, IEEE.

[18] Joshua Morales, Joe Khalife, and Zaher M. Kassas. Simultaneous Tracking of Orbcomm LEO Satellites and Inertial Navigation System Aiding Using Doppler Measurements. In *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, pages 1–6. IEEE, 2019.

[19] Muhammad Muzammal, Romana Talat, Ali Hassan Sodhro, and Sandeep Pirbhulal. A multi-sensor data fusion enabled ensemble approach for medical data from body sensor networks. *Information Fusion*, 53:155–164, 2020.

[20] MReza Naeemabadi, Birthe Dinesen, Samira Najafi, Mikkel Thøgersen, and John Hansen. Feasibility of employing AHRS algorithms in the real-time estimation of sensor orientation using low-cost and low sampling rate wearable sensors in IoT application. In *2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, 2018.

[21] Guillaume Neirinckx and Julien Bastin. Sensor fusion at the extreme edge of an internet of things network. Master's thesis, UCLouvain, 2020.

[22] Talat Ozyagcilar. *Calibrating an eCompass in the Presence of Hard- and Soft-Iron Interference*. Freescale Semiconductor, 2015.

[23] Holger Pirk. Dark silicon — a currency we do not control. https://plds.github.io/programme.html, 03 2020.

[24] William Premerlani and Paul Bizard. Direction Cosine Matrix IMU: Theory. Technical report, DIY DRONE: USA, 2009.

[25] Min-Shik Roh and Beom-Soo Kang. Dynamic Accuracy Improvement of a MEMS AHRS for Small UAVs. *International Journal of Precision Engineering and Manufacturing*, 19:1457–1466, 10 2018.

[26] D. Rose. Rotations in Three-Dimensions: Euler Angles and Rotation Matrices. `http://danceswithcode.net/engineeringnotes/rotations_in_3d/rotations_in_3d_part1.html`, 02 2015.

[27] Argyrios Samourkasidis and Ioannis N. Athanasiadis. A Miniature Data Repository on a Raspberry Pi. *Electronics*, 6(1):13, 2017.

[28] Jessica Velasco, Leandro Alberto, Henrick Dave Ambatali, Marlon Canilang, Vincent Daria, Jerome Bryan Liwanag, and Gilfred Allen Madrigal. Internet of things-based (iot) inventory monitoring refrigerator using arduino sensor network. *Indonesian Journal of Electrical Engineering and Computer Science*, 18(1):508–515, 04 2020.

[29] Eric A Wan and Rudolph Van Der Merwe. The unscented Kalman filter for nonlinear estimation. In *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No. 00EX373)*, pages 1–6. IEEE, 2000.

[30] Greg Welch and Gary Bishop. An introduction to the Kalman Filter. Technical report, University of North Carolina at Chapel Hill, 1995.

[31] Wikipedia contributors. Derivative — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Derivative`, 2021. [Online; accessed 12-April-2021].

[32] Oliver J. Woodman. An introduction to inertial navigation. Technical report, University of Cambridge, 2007.

[33] Nagesh Yadav and Chris Bleakley. Accurate Orientation Estimation Using AHRS under Conditions of Magnetic Distortion. *Sensors*, 14:20008–20024, 2014.

[34] Nisha Yadav, Sudha Yadav, and Sonam Mandiratta. A Review of various Mutual Exclusion Algorithms in Distributed Environment. *International Journal of Computer Applications*, 129(14):6, 2015.

[35] Hugh D Young and Roger A Freedman. *University Physics with Modern Physics, 14th edition*, pages 304,305. Pearson, 2016.

[36] Julian Zeitlhöfler. Nominal and observation-based attitude realization for precise orbit determination of the Jason satellites. Master's thesis, Technical University of Munich, 2019.

Appendix A

Description of experiments

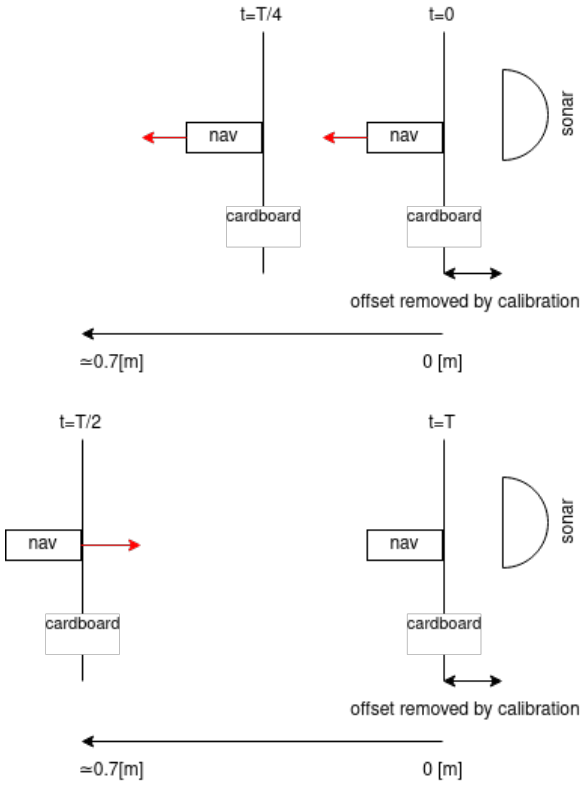

Figure A.1: Description of the experiment "Inertial navigation with Kalman filters"

Appendix B

Experimental setup

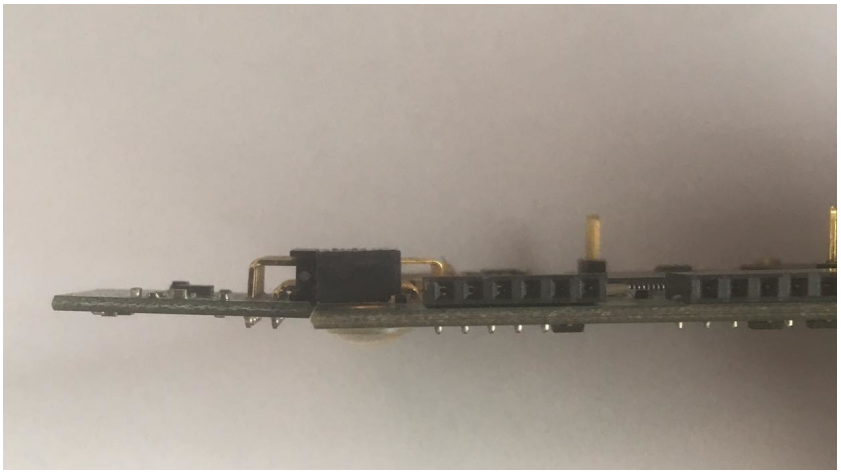

Figure B.1: Pmod NAV plugged in the GRiSP board with a tilt of $\approx 5°$

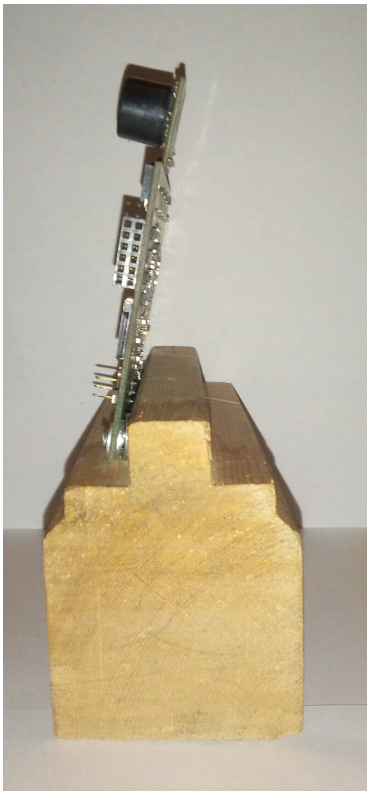

Figure B.2: Sonar support with tilt of $\approx 10°$

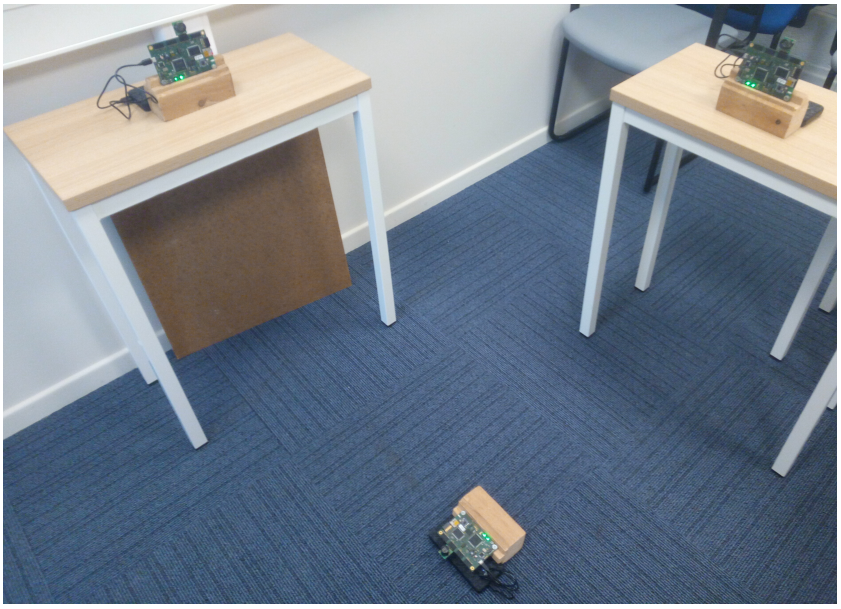

Figure B.3: Sonars setup for 3d tracking

Appendix C

Additional graphs

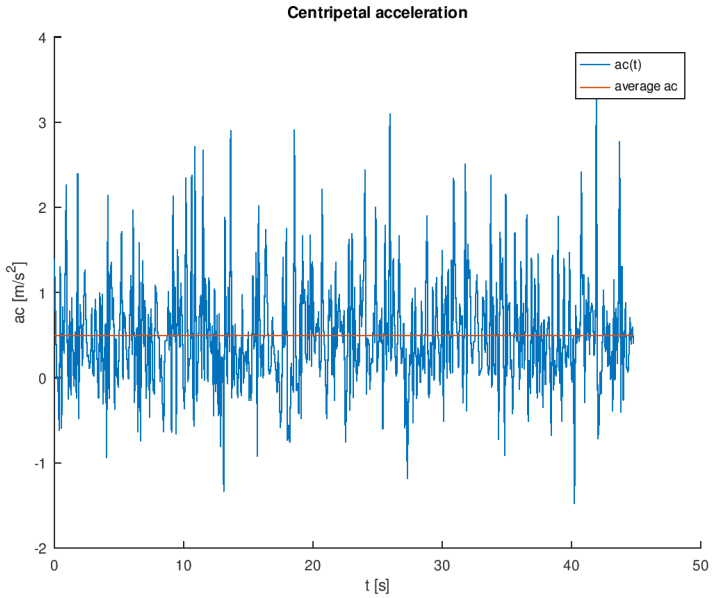

Figure C.1: Centripetal acceleration of the toy train

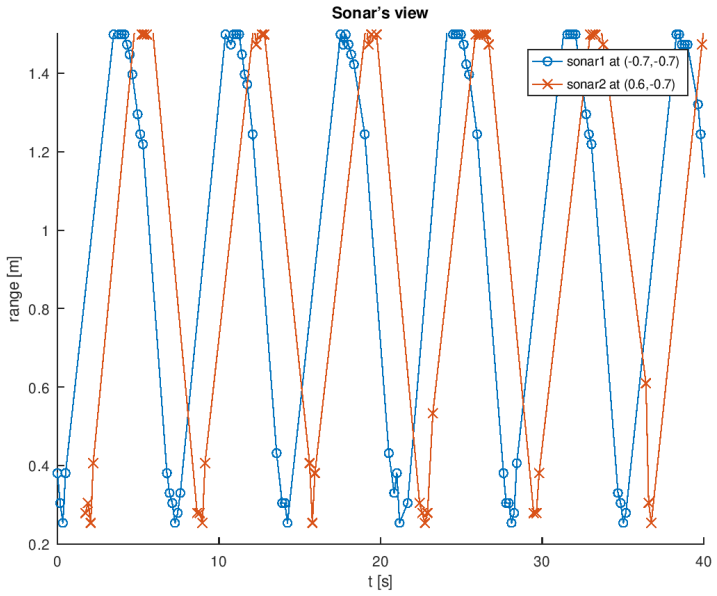

Figure C.2: Sonars view of the toy train

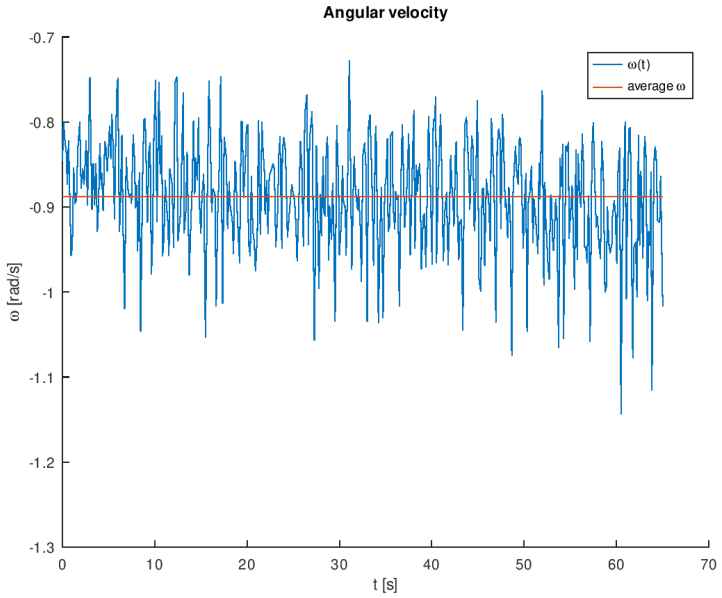

Figure C.3: Angular velocity of the toy train

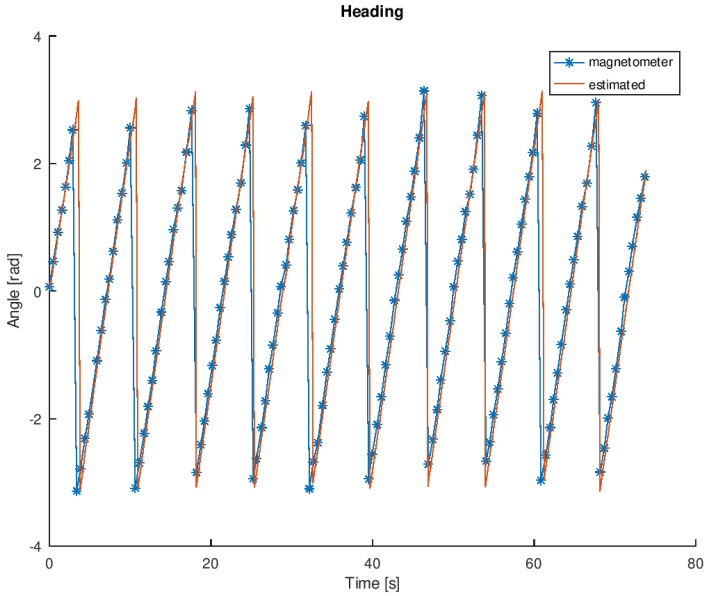

Figure C.4: Heading from the magnetometer and the Kalman filter

Appendix D

The importance of the physical model

When trying to track the position of a target by integrating the acceleration twice assuming it remains constant, as shown in chapter 3, may not give good results if this hypothesis does not hold. In this chapter, we show that with a more faithful model of reality, we can still achieve good results when the "naive" inertial navigation does not.

In the following experiment, we have performed a circular motion with a hand holding a GRiSP board equipped with a Pmod NAV. The period $T \approx 1.25$ [s] and the amplitude $A \approx 4$ [m/s]. Additionally, two sonars have been used to track the board with bilateration. Using a Kalman filter, we have tried to combine the absolute information of bilateration with the same inertial model as described in section 3.4. Fig.D.1 shows the result of this attempt. As you can see, initially, the blue line seems to describe a circular motion, but quickly drifts away and the tracking goes wrong. Of course, we could increase the weight of bilateration, but it would not solve the inherent problem which is that the constant acceleration hypothesis does not hold.

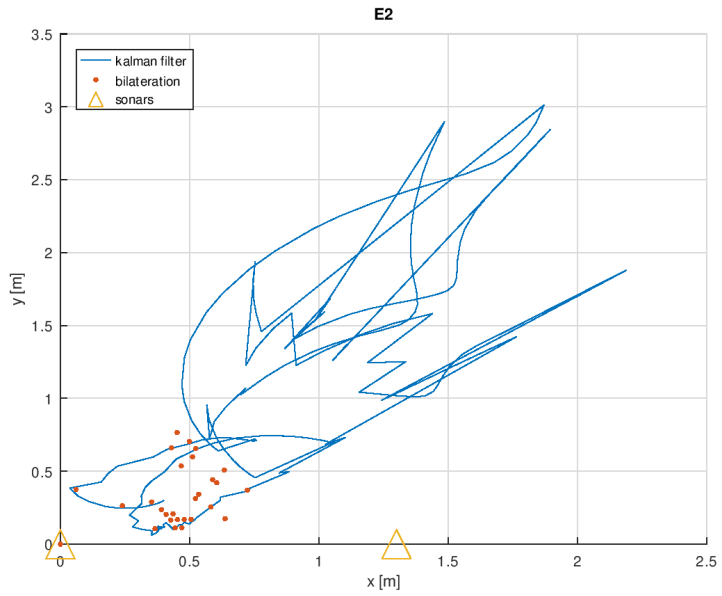

Figure D.1: Tracking while assuming constant acceleration

If we look more closely at the current situation, we can actually do much better. Since we are performing a circular motion (without changing the direction of the board), the acceleration along one axis is sinusoidal (D.1). The amplitude and the period (or angular velocity) can be estimated from the accelerometer signal itself. If we integrate $a(t)$ twice, we realize that the position $p(t)$ is a function of the acceleration (D.4), and in this expression is drift-free.

$$a(t) = A\sin(\omega t) \tag{D.1}$$

$$v(t) = -\frac{A}{\omega}\cos(\omega t) \tag{D.2}$$

$$p(t) = -\frac{A}{\omega^2}\sin(\omega t) \tag{D.3}$$

$$= -\frac{1}{\omega^2}a(t) \tag{D.4}$$

Fig.D.2 shows the position tracking achieved by (D.4) and it is clear that this approach is far superior to assuming a constant acceleration.

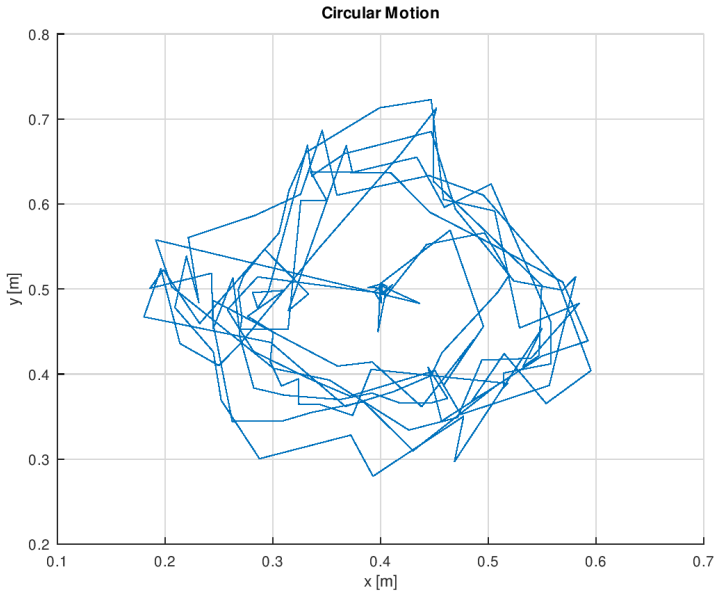

Figure D.2: Tracking with a faithful model of reality

Appendix E

Correction of a bug in the Pmod NAV driver

When we worked with the magnetometer, we noticed some strange behaviours that made no physical sense to us. At first, we could really explain them, but with the help of Peer Stritzinger, we could identified and fix it.

First, we observed a strange behaviour we when tried to retrieve the position of the train on the circle. As you can see on Fig.E.1, between 5 [s] and 7 [s], the value of m_y has an important offset. In fact, the offset was even greater than the maximal possible value according to the specification. If we compute the angle from these data, we end-up with a wrong value, periodically (Fig.E.2). At this point, we decided to simply discard the output when m_y was bigger than a certain threshold.

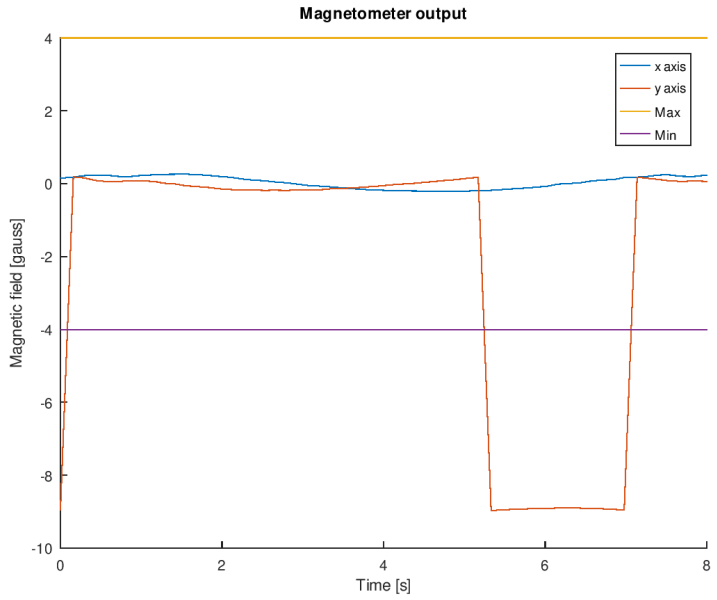

Figure E.1: Magnetometer output with a bug

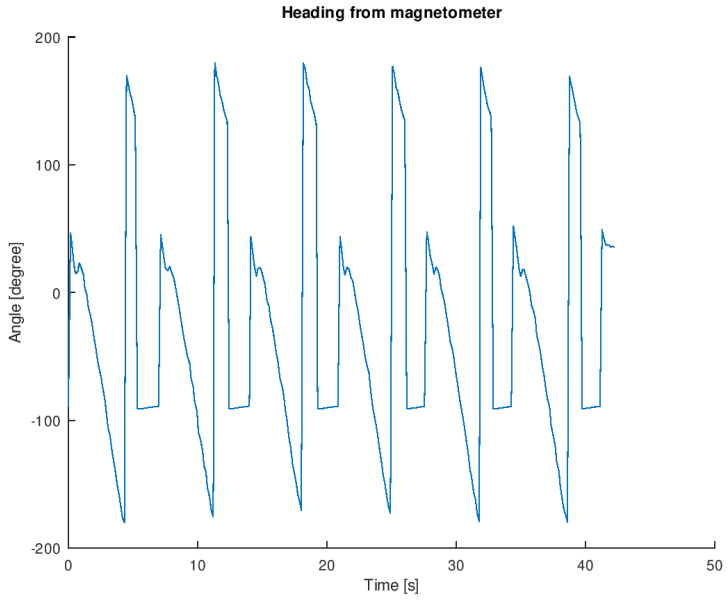

Figure E.2: Position of the train with a bug

However, when we tried to visualize the magnetometer output in 3d, the issue became more apparent. Instead of obtaining a sphere when we measure the

magnetic field while rotating the sensor in all directions, we got four quarter of a sphere placed at each corner of a square (Fig.E.3).

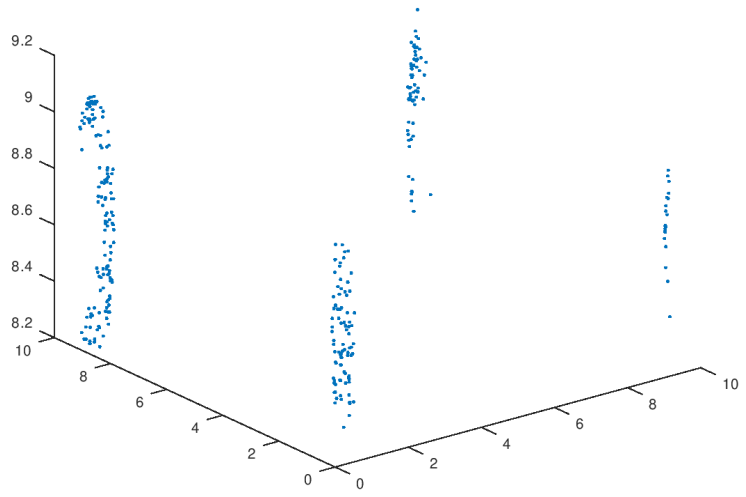

Figure E.3: Output of the magnetometer as we rotate the sensor in all directions

It appears that the problem came from an incorrect binary decoding A number was interpreted as an unsigned value, but should have been interpreted as a signed value. We made a pull request [1] to fix this small issue, and the problems disappeared.

[1] `https://github.com/grisp/grisp/pull/77`

Appendix F

User manual

This user manual is also available in our Github[1] and we advise you to read it there in case of change.

F.1 Required hardware

To use our system you need:

- 1 computer
- 1 wifi access point (a smartphone is enough)
- 4 GRiSP-base (with sd card)
- 1 Pmod NAV
- at least 3 Pmod MAXSONAR
- 4 batteries (with micro-usb connector)

You can find all GRiSP related hardware at `https://www.grisp.org/shop/`. Additionally, if you want to reproduce some of our experiments you might need to purchase a toy train [2].

F.2 Required software

To use our system you need to install on your computer:

[1] `https://github.com/sebkm/sensor_fusion/blob/master/README.md`
[2] `https://www.lgb.com/products/details/article/90463/`

- Erlang/OTP 22.0
- rebar3 3.13.0
- rebar3_hex 6.9.6
- rebar3_grisp 1.3.0
- GNU Octave (only for the visualization tool)

It is very important to install the specified versions because the most recent versions are not compatible. We advise you to work on GNU/Linux and to follow this tutorial `https://github.com/grisp/grisp/wiki` in case of problems. First, you can install Erlang/OTP. Then, you can install rebar3 and follow this tutorial `https://github.com/erlang/rebar3/#getting-started`. When this is done, you should specify the plugins in ~/.config/rebar3:

```
{plugins, [
  {rebar3_hex, "6.9.6"},
    {rebar3_grisp, "1.3.0"}
]}.
```

and run the following command to update the plugins and verify if they are correctly installed:

```
rebar3 update && rebar3 plugins list
```

F.3 Configuration files

F.3.1 Network configuration

To connect the GRiSP boards via existing wifi network, you first need put the information relative to your network in wpa_supplicant.conf. Then, you must indicate the IP addresses and hostnames in erl_inetrc. For example:

```
{host, {192,168,43,217}, ["sebastien"]}. % computer node
{host, {192,168,43,12}, ["sonar_1"]}.
{host, {192,168,43,142}, ["sonar_2"]}.
{host, {192,168,43,245}, ["sonar_3"]}.
{host, {192,168,43,90}, ["nav_1"]}.
{host, {192,168,43,206}, ["nav_2"]}.
```

To find the IP of a board, you can perform a network scan or follow the tutorial from grisp. Finally, you should write this information on your computer in /etc/hosts. The format is not the same. Here is an example:

```
127.0.1.1   sebastien
192.168.43.12 sonar_1
192.168.43.142 sonar_2
192.168.43.245 sonar_3
192.168.43.90 nav_1
192.168.43.206 nav_2
```

If you wish to use our system as is then you must follow the same hostnames as shown in the example (More on that later).

F.3.2 Other configurations

computer.config.src is used for the computer node. The log_data variable must be set to true if you wish to receive the data collected on the network.

```
{hera, [
    {log_data, true}
]}
```

The data will be written in **csv** format in measures/. The file names follow a specific nomenclature: *measureName_sensor_fusion@hostname.csv*.

sys.config is used for the GRiSP nodes. An error logger is setup to generate a report in LOGS/ERROR.1 on the sd card. If you suspect something has gone wrong with the system you should look there, but be aware that the information might be outdated as these files are never removed.

vm.args must contain:

```
## Name of the node
-sname sensor_fusion

## Cookie for distributed erlang
-setcookie MyCookie
```

Finally, rebar.config contains all the information to build and deploy the system. The only part you should modify is the path to the sd card on which you want to deploy the system. In this example, the sd card is names "GRISP".

```
{deploy, [
  {pre_script, "rm -rf /media/sebastien/GRISP/*"},
  {destination, "/media/sebastien/GRISP"},
  {post_script, "umount /media/sebastien/GRISP"}
]}
```

If you want to have more in-depth information about configuration files here are a few useful links:

- `https://github.com/grisp/grisp/wiki`
- `https://github.com/grisp/rebar3_grisp`
- `https://github.com/erlang/rebar3`
- `http://erlang.org/doc/man/config.html`
- `http://erlang.org/documentation/doc-5.9/doc/design_principles/distributed_applications.html`

F.4 Deployment

The first thing to do is to format each sd card as **fat32**. We also suggest to name it "GRISP". The easiest way to achieve that is to use a partitioning tool like "KDE Partition Manager" or similar. You only need to this once.

To ease the process we created a make file that we use as a command shortcut. You can deploy the software on each sd card with the `make deploy-hostname` command where hostname is the name of the GRiSP board. This will take some time. For example:

```
make deploy-nav_2
```

After that, you can plug the sd cards in their respective GRiSP boards. Then, you should plug the Pmod sensors:

Pmod	Slot
NAV	SPI1
MAXSONAR	UART

Finally, you can connect the board to the battery. During the boot phase you should see one green LED. After ≈5 min you should see two red LEDs or two green LEDs (see later). In case of problems we advise you to connect the GRiSP-base by serial[3]. You can use `make screen` once the cable is plugged in.

In parallel you can start the application on your computer. You can either have a clean start with:

```
make local_release && make run_local
```

or start in development mode with:

[3] `https://github.com/grisp/grisp/wiki/Connecting-over-Serial`

```
make shell
```

We advise you to start in development mode. Note that if you use the release the measures folder will be created in the release root directory. After a few seconds a message will be displayed telling you that the application is booted. You might need to press enter to get the shell prompt.

F.5 Launching the system

F.5.1 Calibration

Certain sensors require a calibration in order to be used. If the information is not present on the system, you should see two red LEDs. On the other hand, if the information is present, you should see two green LEDs. Note that in case of restart, as long as there is at least one node staying alive, the information will remain available.

The calibration routine depends on the node hostname as well as the type of measurements that you wish to perform. The first step we suggest is to open a remote shell. If you know what you are doing you can also perform the calibrations with the `rpc` module. When the calibration is done you can close the remote shell.

Calibration of a sonar node:

First, you should open a remote shell to the node. For instance:

```
make remote-sonar_1
```

Then you must call:

```
sensor_fusion:set_args(sonar, Arg1, Arg2, Arg3).
```

Where `Arg1` is the maximal range that can be measured by the sonar while `Arg2` and `Arg3` are either the x and y coordinate of the sonar or the distance to the origin (0,0,0) and the direction (-1 or +1) of the sonar with respect to the axis it is aligned with. The former is used for the experiments with the train while the latter is used for the 6 DOF IMU. The 6 DOF IMU requires 3 sonars and each of them must be placed on a different axis (x,y,z). If you need to, you can use `sonar:range/0` to see what the sonar is measuring.

Calibration of a nav node:

First, you should open a remote shell to the node. For instance:

```
make remote-nav_1
```

Then you must call:

```
sensor_fusion:set_args(Nav).
```

Where `Nav` is either `nav` or `nav3`. The former is used for the experiments with the train while the latter is used for the 6 DOF IMU. Once you press enter, instructions will appear on your screen. Follow these instructions.

F.5.2 Launching the measurements

You can launch the whole cluster from any node with:

```
sensor_fusion:launch_all().
```

Alternatively you can launch a single node from the remote shell with:

```
sensor_fusion:launch().
```

If the LEDs switch to green, you can consider the system to be launched. If they switch to red (or remain red) it means the calibration data was not found.

If you wish to stop the measurements on the whole cluster you can use:

```
sensor_fusion:stop_all().
```

F.5.3 LiveView

You can visualize the measurements in soft real time with the LiveView tool. To start it just do:

```
make liveView
```

This will open a small GUI. First, select the view you want. We have created 4:

- train tracking
- sonar range
- 3d orientation
- 3d position

Then, click on the button "data.csv" and select the csv file you want to read. Each csv file produced by the erlang application starts with the name of the measure. Here is the correspondence between the view selection and the measure names:

view name	measure name
train tracking	e5,e6,e7,e8,e9
sonar range	sonar
3d orientation	e11
3d position	e10

Then, click on the "start" button. You should see a colored square on the top right of the screen. If the square is green it means data is being received in soft real time, but if it is red then nothing is being received.

The tool also provides a "replay" mode that you can use to review the measures with a "real time" feel.

F.6 Development

Our application is made to be extended. The dynamic measurements are handled by the framework "hera" and the actual measurements as well as the calibration storage are handled by the application "sensor_fusion". The framework will not be explained in this document.

F.6.1 Creating a new measure process

You can easily add a new measure by creating a new hera_measure behaviour module. You only need to provide two functions: `init/1` and `measure/1`.

The `init/1` callback takes as argument any Erlang term and must return a tuple of the following type:

```
{ok, State :: term(), Spec :: measure_spec()}.
```

Where `Spec` is a map specified by:

```
-type measure_spec() :: #{
    name := atom(), % measure id
    iter := pos_integer() | infinity, % number of measures
        to perform
    sync => boolean(), % must the measure must be
        synchronized? (default: false)
    timeout => timeout() % min delay between two measures (
        default: 1)
}.
```

The argument of `measure/1` can be any Erlang term. You can use it for all sorts of things like a state variable or calibration data. This callback must return a tuple of the following type:

```
{ok, Values, NewState} | {undefined, NewState} when
Values :: [number(), ...],
NewState :: term().
```

To start the measure you should call:

```
hera:start_measure(Module, Args).
```

Where `Module` is your behaviour module and `Args` will be passed as argument to `init/1`. If the `Args` passed to `hera:start_measure/2` must be persistent you can store it in the system with:

```
sensor_fusion:update_table({{Key, node()}, Args}).
```

And later retrieve it with:

```
ets:lookup_element(args, {Key, node()}, 2).
```

For more information, we advise you to look at some examples in this application. You should also look in sensor_fusion.erl to see how we launch the system and manage the persistent data.

F.6.2 Adding a new sensor

The first thing to do, is to enable the driver of the Pmod sensor with:

```
grisp:add_device(Slot, Driver).
```

Where `Slot` is the lower case version of the slot name printed on the board itself and `Driver` is the name of the driver module [4].

Then, you can create a new measure process (section F.6.1). To read the sensor, you simply need to call the sensor driver *get* function in the `measure/1` callback of your `hera_measure` behaviour module. For instance:

```
measure(Calibration) ->
    RawData = pmod_nav:read(acc, [out_x_xl]),
    % ... do something with the RawData and the Calibration
    {ok, CorrectedData, Calibration}.
```

In this example, we receive a calibration as argument. Here are the steps to follow if you need a calibration:

1. Create and export a calibration function in your module.

[4] All the drivers can be found at `https://github.com/grisp/grisp/tree/master/src`

2. Before you start the measure, call your calibration function and store the output with `sensor_fusion:update_table/1`.

3. Retrieve the calibration data with `ets:lookup_element/3` and pass it as argument to `hera:start_measure/2`.

4. Your `init/1` callback should forward the data in the `State` variable.

If you need to, you can also update the state with the `NewState` variable in the output of your `measure/1` callback.

F.6.3 Adding a new sensor fusion model

In Hera, we perform sensor fusion via `hera_measure` modules. Each module has an `init/1` and a `measure/1` callback. In the `measure/1` callback, we can fetch the data from the local data store with `hera_data:get/1` and `hera_data:get/2`. You can discard old data with a simple timestamp filter. Then, you can write the parameters required for your sensor fusion model using list comprehensions and the fetched data. In our case, we use a Kalman filter and so, we write all the "variadic" matrices needed. Finally, we give these parameters as argument to the Kalman function, included as library in Hera, and recover the result.

Here is a short example where we fetch new (not used in the previous computation and arrived since at most 500 [ms]) sonar data, create the matrices and call the Kalman filter. Then, we serialize the matrix containing the state vector and we update the state of the model. As you can see, we update a timestamp to avoid reusing data multiple times.

```
measure(State = {T0, X0, P0}) ->
    DataSonars = hera_data:get(sonar),
    T1 = hera:timestamp(),
    Sonars = [{Node,Data} || {Node,Seqnum,Ts,Data} <-
       DataSonars, T0 < Ts, T1-Ts < 500],
    Matrix1 = ...
    Matrix2 = ...
    {X1, P1} = kalman:kf(State, Matrix1, Matrix2, ...),
    NewState = {T1, X1, P1},
    Values = lists:append(X1),
    {ok, Values, NewState}.
```

Our Kalman library offers two Kalman filters: the linear Kalman filter without control input and the extended Kalman filter without control input. We also separately implement the predict and update phase for the linear Kalman filter,

allowing to use them independently if needed. A matrix library is also included in Hera and offers common matrix operations in the `mat` module.

For more information, we advise you to look at some examples in this application.

F.6.4 Adding new libraries to Hera (Kalman filters, matrix operations, ...)

The Hera framework comes with 2 implementations of Kalman filters: the linear Kalman filter without control input and the extended Kalman filter without control input. For some applications, new Kalman filters or even different data fusion algorithm can be added to Hera in a straightforward manner. Hera also provide a matrix library that could be extended with new operations. The only constraint is that the algorithms must be implemented by pure functions. The purpose is simply to provide a reusable toolbox for the `hera_measure` modules where all the states and side effects should be embedded.

F.6.5 Updating the code

When you make a change to the code, you can compile it with `rebar3 compile` and even have additional type checks (recommended) with:

```
rebar3 dialyzer
```

In case you made changes to dependencies for example by placing it in _checkouts/, you should run `make clean` and remove build files in the dependency folder before compiling. If you change something in kalman.erl or in mat.erl you are encouraged to run the tests:

```
make test
```

If you are running the application with `make shell`, you can update the code live (without leaving the application) on all the nodes in the cluster with:

```
sensor_fusion:update_code(Application, Module).
```

Where `Application` and `Module` are the application and module you wish to update. This will make the update permanent (even after a reboot). The Application must already exist on each GRiSP-base, but the module may be new. However, if it is new, the module will not be loaded on start-up.

Note that if you update hera your changes will only be applied after a reboot of hera. You can force it with `sensor_fusion:stop_all/0`.

If you made lots of changes you should consider deploying the application again.

F.6.6 Adding a new view in LiveView

Creating your own view requires only 3 steps:

1. Create a figure initialization callback (e.g. initTrain.m) that should return handles to your axes.

2. Create a figure update callback (e.g. updateTrain.m) that receives the last measure as well as the handles. A data global variable can be used to store data in a more permanent way. The data will be erased upon leaving the view.

3. Adding a new entry in the view selection menu as well as your callbacks in liveView.m:

```
initView = {@initTrain, @initSonar};
updateView = {@updateTrain, @updateSonar};
views = {"train tracking", "sonar range"};
```

Currently, liveView only enables you to read from one file, but you can have multiple instances of liveView at the same time. This way, you can visualize data from multiples sources. If you need to visualize the data before the measurement is even started on the Erlang application, you can create the csv file yourself and start liveView. As soon as it will be written, the view will be updated.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**