The Hera Framework for Fault-Tolerant Sensor Fusion with Erlang and GRiSP on an IoT Network

Sébastien Kalbusch
kalbusch.sebastien@gmail.com
Université catholique de Louvain
Belgium

Vincent Verpoten
vincent.verpoten@hotmail.com
Université catholique de Louvain
Belgium

Peter Van Roy
peter.vanroy@uclouvain.be
Université catholique de Louvain
Belgium

Abstract

Classical sensor fusion approaches require to work directly with the hardware and involve a lot of low-level programming, which is not suited for reliable and user-friendly sensor fusion for Internet of Things (IoT) applications. In this paper, we propose and analyze Hera, a Kalman filter-based sensor fusion framework for Erlang. Hera offers a high-level approach for asynchronous 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 avoid soldering or dropping down to C. We emphasize on the importance of performing all the computations directly at the sensor-equipped devices themselves, completely removing the cloud necessity. We show that we can perform sensor fusion for position and orientation tracking at a high level of abstraction and with the strong guarantee that the system will keep running as long as one GRiSP board is alive. 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.

CCS Concepts: • **Computer systems organization** → **Sensor networks**; **Availability**; *Real-time system architecture*; • **Computing methodologies** → **Distributed programming languages**; • **Mathematics of computing** → **Kalman filters and hidden Markov models**; • **Human-centered computing** → *Visualization systems and tools.*

Keywords: IoT, fault tolerance, sensor fusion, Kalman filter, AHRS, Erlang, GRiSP, Pmod

ACM Reference Format:
Sébastien Kalbusch, Vincent Verpoten, and Peter Van Roy. 2021. The Hera Framework for Fault-Tolerant Sensor Fusion with Erlang and GRiSP on an IoT Network. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang (Erlang '21), August 26, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471871.3472962

Erlang '21, August 26, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8612-8/21/08…$15.00
https://doi.org/10.1145/3471871.3472962

1 Introduction

Sensor fusion is the ability to combine information from multiple sensors to give a single coherent view of a real-world situation. It 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 a 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 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 (13% per year versus 5% per year) [18].

We deliberately make no use of cloud because delegating computation to the cloud comes with significant disadvantages. Using the cloud significantly increases cost. Computing at the extreme edge not only has the potential to reduce costs, but also to largely simplify the infrastructure. Indeed, going to cloud requires a considerable amount of work and increases the overall complexity of IoT projects. Additionally, the non-negligible[2] latency of the edge-cloud connection prohibits the use of cloud computing for certain domains, like tracking, for which the slightest delays decrease the accuracy. Furthermore, the edge-cloud connection has an unpredictable reliability and goes down surprisingly often. Therefore, it is essential to bring the computation to the extreme edge instead of simply using the edge devices for data gathering.

In this paper, 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 platform to all interested parties.

The hardware consists of a network of GRiSP boards, each of which can host Digilent Pmod modules [2] 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

[1] In terms of structure and not physical distance.
[2] Usually tens of ms.

or product development. It was developed in the context of a long-term research in IoT applications at UCLouvain in collaboration with Stritzinger GmbH[3]. This research started in the European Horizon 2020 LightKone project [12] and led to the development of a first version of Hera in [16]. The framework presented in this paper has since been improved by two students during their master thesis [11] with a special focus on simplicity.

The contributions are:

1. Hera, an open-source framework for fault-tolerant sensor fusion running on a network of GRiSP boards and based on Kalman filters [21]. 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 allowing measurements (sensor or computation) to be started at any time. Moreover, any known GRiSP board can join or leave (restart or crash) the system while it is running.
   - A modular visualization tool built with GNU Octave [4].
2. Evaluation of the fault tolerance by performing fault injection and observing how the framework responds. Hera guarantees to continuously do sensor fusion as long as one GRiSP board in the network is running. If a sensor or a board fails, the software continues to work with a degraded accuracy.
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 attitude and heading reference system (AHRS) able to update its physical model at a rate of 3.75 Hz, running completely on a GRiSP board.

The paper is structured as follows. Section 2 gives background information about the GRiSP platform and the Kalman filters used in the sensor fusion engine. This section also compares our approach to related work. Section 3 presents the design of the Hera framework. Section 4 gives a practical evaluation of Hera's fault tolerance using fault injection. Section 5 evaluates a demonstrator model to better understand the impact of each sensor in the calculation. Section 6 explains and analyzes a microelectromechanical system (MEMS) AHRS to provide a useful example of what can be achieved with Hera. Finally, section 7 concludes by giving a summary and an overview of possible future developments.

2 Background

2.1 The GRiSP Platform

The GRiSP platform is an embedded system that allows IoT prototyping out of the box. A GRiSP board contains a full Erlang/OTP platform running on a RTEMS layer, is equipped with six sockets for Pmod modules, and has Erlang drivers for the most common Pmod modules. We use the Pmod MAXSONAR[4] allowing to detect objects and people at a distance up to 6.5 m with an accuracy of 2.5 cm and the Pmod NAV[5] which provides a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer.

The use of Erlang facilitates the development of IoT applications because of its native support for concurrency, fault tolerance, and hot code loading. It is largely because of the abilities provided by GRiSP and Erlang that it was possible to develop, test, and evaluate Hera within the time constraints of a master thesis.

2.2 Kalman Filter

The Kalman filter is a well-known data fusion technique for state estimation with multiple benefits: it is simple to implement, handles noisy data, and can be used for a variety of problems. Since the Hera framework provides two general variations of Kalman filters, we give a conceptual explanation of this technique. We invite unfamiliar readers to consult [8] for a tutorial-like description, but note that the generic discrete Kalman filters we propose do not use any control input.

The linear Kalman filter aims to estimate the normally distributed state of a system that can be modeled by a set of linear equations. It is composed of two phases called predict and update (Fig.1). The prediction phase allows to estimate the state of the system $\hat{x}_{k+1}^{-}$ from the previous known estimate $\hat{x}_k$ and its physical model F_k. 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. Additionally, the Kalman filter produces P_k, the state estimate covariance matrix, which gives information about the quality of the estimate.

While the prediction phase allows for accurate estimation of a well-modelled problem, the update phase serves to correct the state in a changing environment and prevents drift due to accumulation of small errors or inaccuracies in the model. 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. Moreover, there is no obligation to perform both the

[3] https://www.stritzinger.com

[4] https://reference.digilentinc.com/reference/pmod/pmodmaxsonar/start

[5] https://reference.digilentinc.com/reference/pmod/pmodnav/start

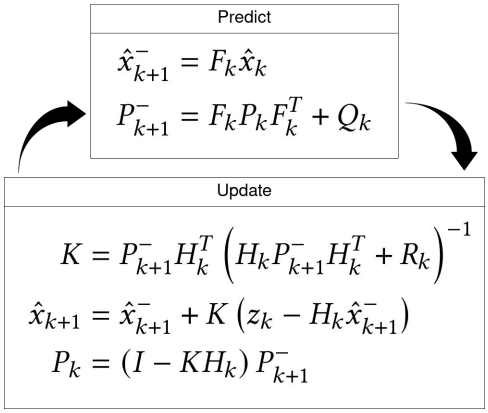

Figure 1. The two phases of the Kalman filter

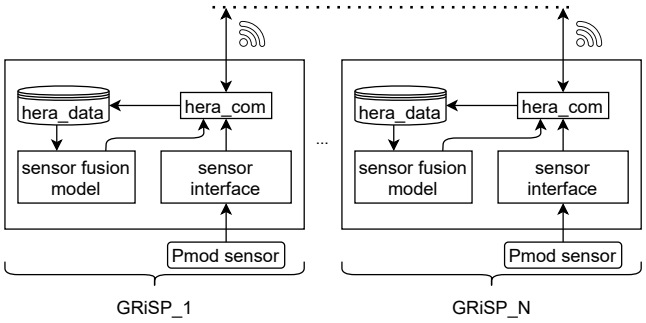

Figure 2. Data flow overview

prediction and update phases every time. If needed, it is possible to call one of them multiple times in a row.

Fortunately, we are not limited to linear problems. The extended Kalman filter (EKF) works with f and h, the differentiable functions for the state transition and the observation model as well as their Jacobian, evaluated around the estimate, to compute the state covariance by linearization.

2.3 Approach

Our approach to perform sensor fusion is based on the separation of concerns principle. The concerns are: the low-level interactions with the hardware, the edge computing model, and the high-level description of a sensor fusion model.

The hardware and sensor drivers are provided by the GRiSP platform and all the boards are connected via standard Wi-Fi protocol. Each GRiSP board uses the Hera framework which provides a generic measurement behaviour, a data storage, and shares data via broadcast. Then, on the client side, we implement "sensor interface(s)" on top of the provided Pmod drivers as well as a "sensor fusion model" module in which we both fetch the appropriate data from the local data store and define the Kalman filter parameters. Fig.2 gives an overview of the data flow.

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. Since we are not trying to achieve a consensus, each node computes independently from each other. However, the results should be close because of the real-time nature of the system[6]. As a consequence of sharing all the data between the nodes, Hera is designed to run in a small cluster, which is a set of fully connected nodes in the same environment. Note that there is no notion of event that would trigger a computation and the data cannot be deleted.

Hera tolerates failures thanks to supervision and restarting schemes made possible in an asynchronous and dynamic system, but also because of redundancy achieved by having multiple GRiSP boards with sensors and by running the same computation on multiple nodes. Of course, these properties are largely due to Erlang's concurrency model.

2.4 Related Work

To our knowledge, there exists no other platform like Hera that we could use for a direct comparison. Instead, we compare our system to low-cost sensor fusion on other platforms and to the architecture of related work on wireless sensor networks. 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.

Low-cost Sensor Fusion: In [1], a Raspberry Pi is used as a gateway for a blood pressure monitoring application and the data is sent to the cloud instead of processing the data directly at the edge as we do. In [20], two Arduino and sensors are used to monitor a refrigerator. The two Arduino are interconnected to exchange sensor information via I2C communications. One of them uses a Wi-Fi interface to send data to Dropbox, used as a cloud storage.

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

Wireless Sensor Networks: In [15], 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. The data is stored in a cloud server, but the machine learning computation takes place in a fog computing environment, 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

[6]The difference is due to occasional data loss in the network and small changes in the data arrival time or processing time.

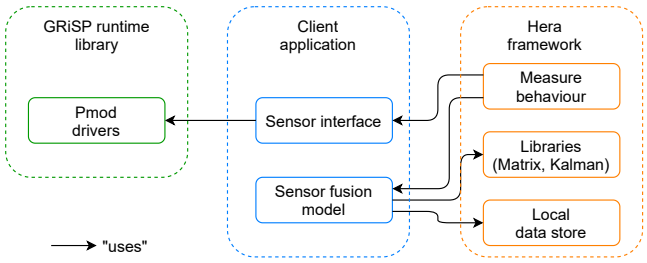

Figure 3. Software architecture

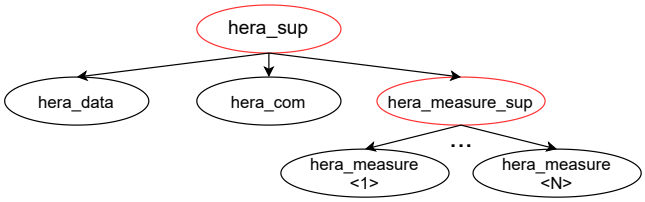

Figure 4. Hera supervision tree

sensor-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 [9] proposes a method for multi-target tracking in large-scale sensor networks based on maximum entropy fuzzy clustering. 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 which offers the possibility to use more sophisticated algorithms in large-scale environments.

3 The Hera Framework

3.1 Software Architecture

The software consists of 3 parts: the GRiSP Erlang runtime library[7], the Hera framework[8], and a GRiSP application[9].

Fig.3 illustrates the architecture and shows where the client (i.e. user) interacts. The client application a.k.a GRiSP application is used to bootstrap the system and to define the callback modules for the "Measure behaviour" of Hera. A "Sensor interface" aims at interfacing Hera with the Pmod drivers of the GRiSP runtime while the "Sensor fusion model" is where the user first fetches the appropriate data from the local data store, then defines the physical model and uses the "Libraries" for the computation. A complete example will be provided in section 5.7.

3.2 Application Design

Hera uses a small supervision tree (Fig.4). The top level supervisor `hera_sup` supervises 3 independent processes: `hera_data`, `hera_com`, and `hera_measure_sup`. These processes are vital for the application and are not expected to fail often.

The second supervisor `hera_measure_sup` supervises `hera_measure` processes. These processes are considered independent of each other and should be dynamically started. Because the measure processes execute the client code and interact with real-world components such as sensors, we expect them to fail more often. This supervisor is not designed to supervise a large number of processes considering natural limitations (network congestion, computational power, number of sensor connectors, ...).

`hera_data` is a `gen_server` used for storing measurement data. It stores only the most recent data identified by a name and the node which sent it, and marks it by a timestamp. Upon reception, it is also possible[10] to log every measure in a unique csv file for each data identifier (name and node). Note that Hera does not provide a fine-grained timing because it uses the arrival time instead of a real-time clock synchronization.

`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 redundancy by running the same computation on several nodes. Another advantage is the ability to introduce "load balancing" capabilities by sharing the computational load. For example, the node that produces sensor measurements may let another node perform the sensor fusion computation.

The `hera_measure` module is a stateful generic measure behaviour. The user is expected to provide a `measure(State)` callback to perform a measure. The specification is flexible and allows, for instance, to discard certain values or to implement a complementary filter stored in the callback state. The `measure/1` callback is invoked in a loop with a parametric delay[11] between each iteration.

Fig.5 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.

3.3 A Measurement Synchronization Extension

The use of sonars in our experiments 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

[7] https://github.com/grisp/grisp
[8] https://github.com/sebkm/hera
[9] https://github.com/sebkm/sensor_fusion

[10] By setting the `log_data` environment variable to `true`.
[11] The delay is chosen by the user and allows to reduce the sampling frequency if a measure is too fast and overloads the system. The delay may be zero.

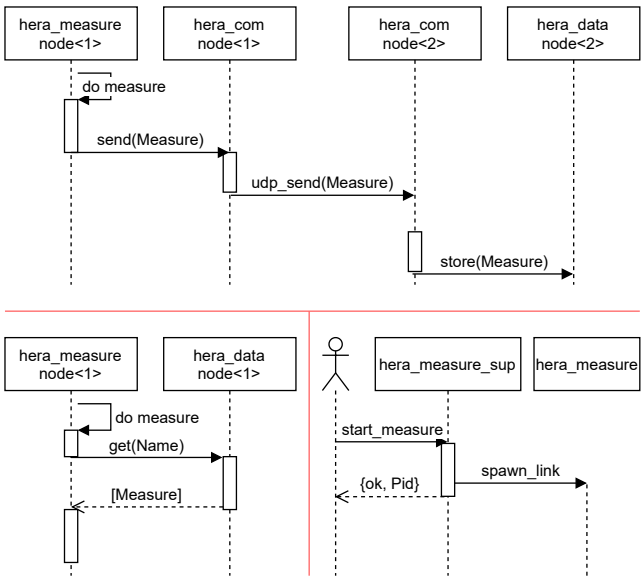

Figure 5. Interaction between processes

property, dynamic addition/removal of processes, and fault-tolerant synchronization. After reviewing several algorithms [22], we chose the centralized approach for simplicity, but we improved its fault tolerance with monitors and heartbeats. The extension is available on Github[12] and is implemented as an "Erlang Distributed Application". Therefore, if the node running the extension goes down, it will be automatically restarted at another node.

Fig.6 shows how the synchronization works. First, the `hera_measure` process subscribes itself to `hera_sub` which forwards the subscription to the dedicated[13] synchronization process which will then authorize the measure in due time.

Fig.7 illustrates how the processes are supervised or monitored and Fig.8 describes how the processes react upon reception of a "DOWN" message. When a measure crashes, the synchronization process simply sends the next authorization (top left). If `hera_sub` fails, `hera_sync` exits because it cannot run independently (top right). When `hera_sync` dies, the subscription server removes it and the measurement processes resubscribe themselves.

3.4 A Matrix Library

Along with the Kalman filters, Hera provides 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 support 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

[12] https://github.com/sebkm/hera_synchronization
[13] A new process is created if needed.

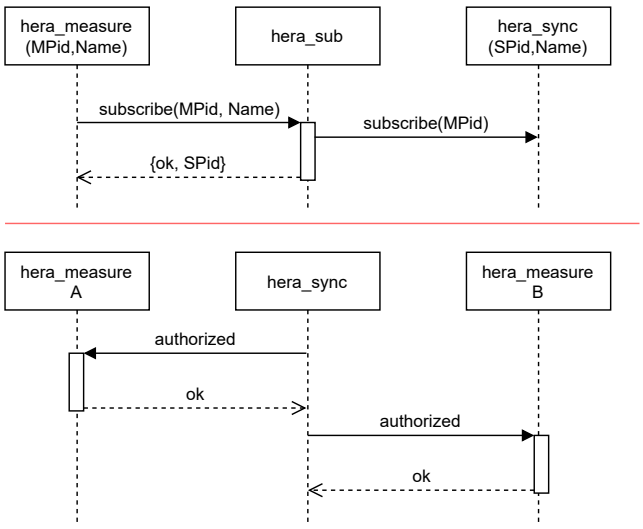

Figure 6. Synchronization principle

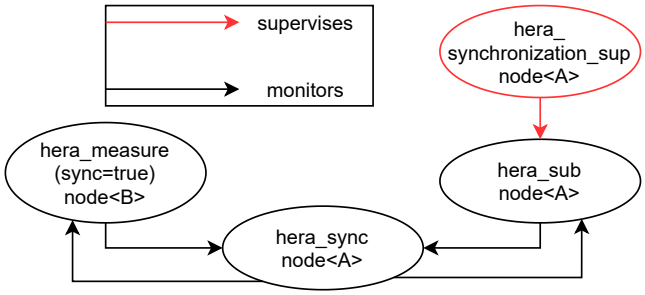

Figure 7. Synchronization monitoring

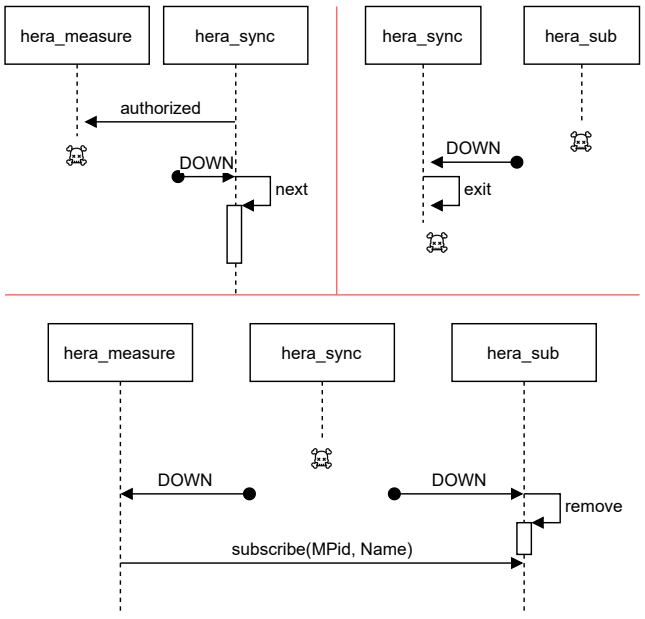

Figure 8. Reaction to "DOWN" messages

representing a matrix as a list of lists. An example will be provided in section 5.7. A long-term solution to the general inefficiency of matrix operations in pure Erlang would be to use an Erlang NIF library compatible with the GRiSP platform. Actually, this project is already ongoing [13].

4 Fault Tolerance Analysis

We validate the fault tolerance of the application by fault injection. Concretely, we simulate failures and observe the system behaviour. To do so, we create dedicated `hera_measure` processes that observe the activity of the system.

4.1 Setup

We use 4 GRiSP boards and 1 computer for all the tests. There are 4 measures running per node which makes a total of $4\times(4+1) = 20$ observers running at the same time. Since the observers do not consume a lot of resources, we intentionally set the parametric constant time between two successive callbacks (see section 3.2) to a non-zero value. In the following paragraphs, the word "cycle" refers to this duration. Note that the observers are only used for the fault tolerance validation and are therefore not running under normal circumstances. 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 duration of a cycle is set to 1 s.

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. As the synchronization is based on names, each "elapsed" is only monitoring 5 processes and not 20 because a key (name, node) must be unique. So, there are 4 concurrent synchronizations with 5 measures per synchronization. The duration of a cycle is set to 100 ms.

4.2 Measurement Process Crashes

Fig.9 shows the results of the fault injection on observer processes and the percentage indicates how many processes per node are killed. On the top graph, we can clearly see that the count gets lower after the fault injection and then returns to normal a few seconds later. On the bottom graph, we see that Δt decreases since it takes less time to receive an authorization when a participant is gone.

In both cases, it takes roughly 2 s to restart all the processes. Of course, for the top graph we should subtract up to 3 s (3 cycles) because of the way we count [14].

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

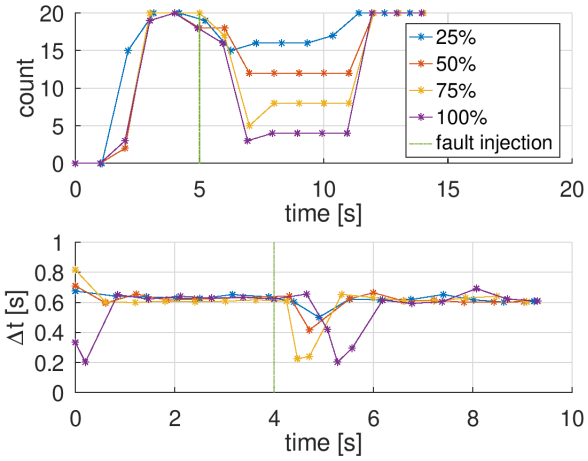

Figure 9. Fault injection on "counter" (top) and "elapsed" (bottom). The green lines indicate when the fault injection happened.

4.3 Transient Failure

Fig.10 shows what happens when we kill `hera_data` and `hera_com` multiple times in a row to simulate transient failures. Killing the data storage process should result in a loss of information and therefore, the count should get lower. This is reflected on the top graph and 5 cycles after the last kill, the situation is restored. Observing the fault injection on `hera_com` is more difficult and we had to lower a cycle to 300 ms. Again, we observe a loss of information after the kills and the situation returns to normal 1 s later.

What we show here is that `hera_measure` processes (i.e. observers) keep working despite transient failures. Of course, for an actual sensor fusion computation, missing data could result in a degraded accuracy, but would not prevent the system from working.

4.4 Synchronization Resilience

Fig.11 shows how the synchronization is affected by the failure of a centralized coordinator. 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 might change after the fault injection (first come, first served policy). As you can see on the top graph, 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 (bottom graph), 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.

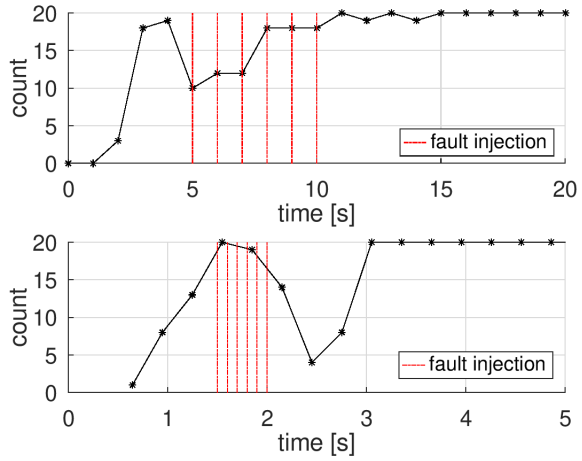

Figure 10. Fault injection on `hera_data` (top) and `hera_com` (bottom). The red lines indicate when the fault injection happened.

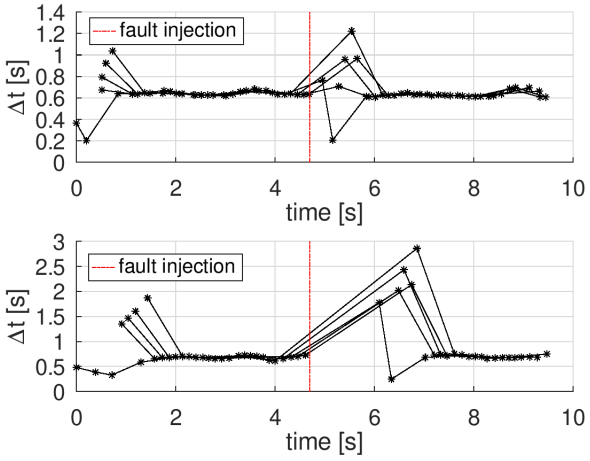

Figure 11. Fault injection on `hera_sync` (top) and `hera_sub` (bottom). The red lines indicate when the fault injection happened.

4.5 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 $TickTime \pm \frac{1}{4} TickTime$ where $TickTime$ is 4 times the delay between two heartbeats. In our configuration, a heartbeat is sent every 2 s hence, the detection should take between 6 and 10 s. From what we observe in Fig.12, the delay 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 took $10 - 2 = 8$ s. Since the

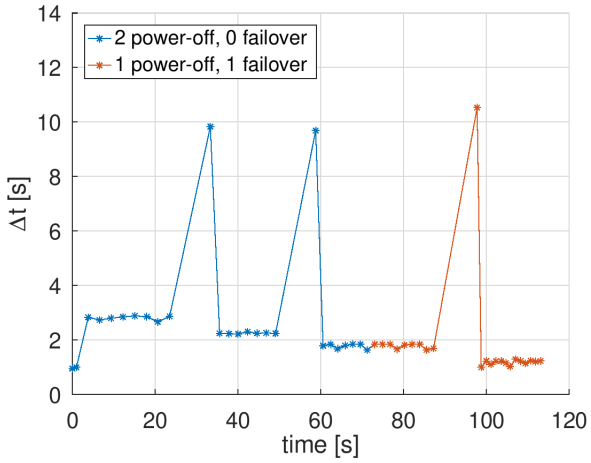

Figure 12. Hardware failure with synchronization

synchronization extension is an "Erlang Distributed Application", when we power-off the node at which it is running, 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.

4.6 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 failures. In particular, we are not concerned by occasional loss of data since the system has an asynchronous model. The temporary interruption that occurs when a GRiSP board encounters a hardware failure is restricted to synchronized measures only (the rest of the system is not affected). The duration of the interruption directly depends on the *TickTime* which can be changed via the `net_ticktime` environment variable, but a too low value may result in a false positive. In our settings, hardware failures are detected in at most 10 s.

5 First Phase: Experimental Model

In order to verify if sensor fusion with Erlang and GRiSP gives satisfactory results, 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 the contribution of each sensor as well as testing the software. In this section, we explain how we built the model and review some of the results. For conciseness, we omit to give all the EKF parameters, but we invite the reader to consult [11] for a more detailed description of the model. Still, we

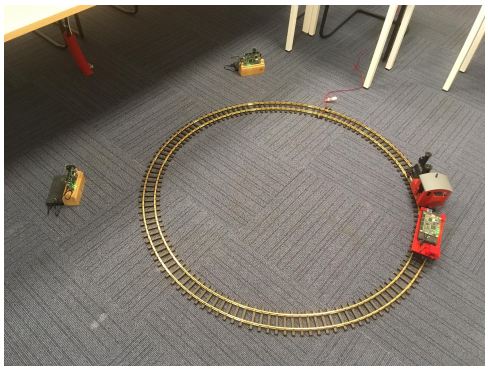

Figure 13. Setup for the experimental model

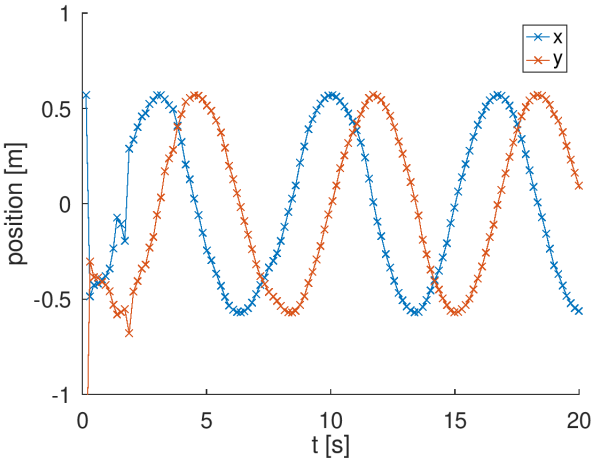

Figure 14. Estimation of the position

give the variance σ^2 for each sensor because this information is crucial to compute the gain K of the Kalman filter.

5.1 Setup

The experimental setup is a toy train with a circular path (Fig.13). The train carries a battery and a GRiSP board equipped with a Pmod NAV. Except for the first model (section 5.2), we also use two Pmods MAXSONAR. The true angular velocity ω of the train is constant and was estimated by measuring how long it took for the train to complete a fixed number of cycles over $\approx$ 60 s. To better visualize the benefit of sensor fusion, we compare the estimated angular velocity $\hat{\omega}$ for each version of the model (Fig.16).

5.2 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 [23]. In the EKF parameters, $h = r\hat{\omega}^2$ and we assume the radius $r = 0.57$ m is known. Since a_c is a constant, we computed $\sigma^2_{a_c} = 0.8$ directly from the accelerometer data.

The "a_c only" curve on Fig.16 shows that the estimation is difficult mostly because the train introduces lots of vibrations. We also observe that ω is overestimated, but a slightly higher r would already reduce the error. Actually, there is a small uncertainty on the value of r because we do not exactly know the position of the MEMS accelerometer on the Pmod NAV chip. 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.

5.3 Position Estimation

We can estimate the position of the train with (1) and the drift can be corrected with the absolute measurement d of a sonar. Indeed, from the estimated position ($\hat{x}$,$\hat{y}$), it is possible to estimate the distance between the train and the sonar located at (P_x,P_y) with (2). The error $\sigma_d = 0.25$ is defined as half the length of the train.

$$\hat{\theta}_{k+1} = \hat{\theta}_k + \hat{\omega}\Delta t \quad \hat{x}_{k+1} = r\cos\hat{\theta}_k \quad \hat{y}_{k+1} = r\sin\hat{\theta}_k \tag{1}$$

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

Fig.14 clearly shows when a sonar measurement is used to correct the estimation. The major corrections take place at the beginning and convergence is complete after two cycles. By that time, $\hat{\omega}$ ("+sonars" curve on Fig.16) also converges towards the "true" value and when we compare the shape of the curve with the previous configuration, it is obvious that the sonars give a major improvement.

5.4 Adding a Gyroscope

We can improve the angular velocity estimation by adding a gyroscope g in the observation vector z and $\sigma^2_g = 0.005$ can be computed from the gyroscope data because the angular velocity is constant.

The much cleaner gyroscope signal allows to reach a stable estimation in only 2 s instead of 15 s for the previous model ("+gyro" curve on Fig.16). From this result, we learn the importance of the gyroscope as inertial sensor.

5.5 Radius Estimation

Since the gyroscope provides information regarding the angular velocity, we can use the accelerometer to estimate the radius with the observation model $h = \hat{r}\hat{\omega}^2$. Of course, we keep the same data sources as the previous configuration, which are an accelerometer, a gyroscope, and two sonars. Measuring r, 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 measured 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.

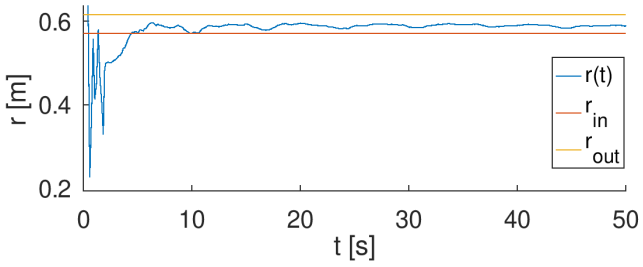

Figure 15. Estimation of r

As you can see on Fig.15, it takes $\approx$ 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.16) shows that estimating both ω and r is a much harder problem, since the convergence takes significantly longer.

5.6 Adding a Magnetometer

A magnetometer provides information about the heading θ_m which can be computed from (3) where m_x and m_y are the measured magnetic field corrected for hard-iron bias. In the observation model h, we use $\hat{\theta}'$ from (4) instead of $\hat{\theta}$ because of the wrapping effect of the SO(2) group [14]. This allows to find the shortest path between $\hat{\theta}$ and θ_m. Because the heading is not constant, $\sigma^2_{\theta_m} = 0.015$ cannot be computed directly from the magnetometer data. However, we can compare it with θ, the true heading obtained from ω and the known initial position θ_0 using $\theta(t) = \theta_0 + \omega t$. The complete model, encoded for Hera, is visible in Listing 2.

$$\theta_m = \arctan2(m_y, m_x) \tag{3}$$

$$\hat{\theta}' = (\hat{\theta} \bmod 2\pi) - 2k\pi \quad \text{where}$$
$$k = \underset{k\in\{0,1\}}{\operatorname{argmin}} \left(\left| \theta_m - (\hat{\theta} \bmod 2\pi) + 2k\pi \right| \right) \tag{4}$$

The magnetometer gives very useful information an so, we are able to restore the precision of $\hat{\omega}$ ("+mag" curve on Fig.16) to what we achieved before adding the radius as state variable.

5.7 Model Encoding in Hera

Hera offers a high-level approach because the user only needs to provide the sensor fusion model with the sensor interfaces and nothing else! It allows the user to focus on the important task: the elaboration of a sensor fusion model. Listing 1 shows the interface we used for the sonar. As you can see, it suffices to call the `get/0` function of the driver and

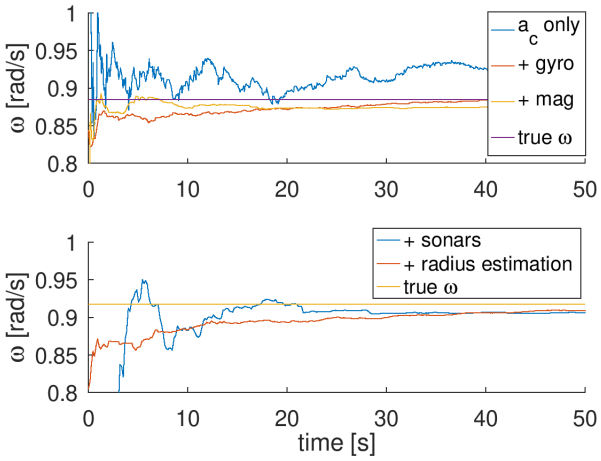

Figure 16. Estimation of ω

```
1  measure(State={MaxRange, X, Y}) ->
2    case pmod_maxsonar:get() of
3      Range when Range*0.0254 =< MaxRange ->
4        {ok, [Range*0.0254, X, Y], State};
5      _ -> {undefined, State}
6    end.
```

Listing 1. Example of sensor interface (Erlang code)

eventually to alter the result or even piggyback additional information, like the position of the sonar, stored in the state.

Listing 2 gives the complete model of section 5.6 which consists of estimating the position of a toy train in real time as well as its angular velocity and the radius of the circular track with an accelerometer, a gyroscope, a magnetometer, and two sonars. The first step is to fetch the data from the local data store (lines 2-4). The second step consists in writing the EKF parameters in matrix form (lines 10-41). In the third and final step, we call the library function to perform the EKF computation and output the result (lines 42-43). There are two important particularities: we filter already used data with a local timestamp (lines 2-5) and we use list comprehensions to declare the observation model (lines 23-41). This "variadic" notation allows to define the complete model once and let the length of the matrices change depending on the available data.

5.8 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 also matches soft real-time expectations. The early results show that some time is 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

```
measure({T0, X0, P0}) ->
N=[D || {_,_,Ts,D}<-hera_data:get(nav),T0<Ts],
M=[D || {_,_,Ts,D}<-hera_data:get(mag),T0<Ts],
S=[D || {_,_,Ts,D}<-hera_data:get(sonar),T0<Ts],
T1 = hera:timestamp(),
if length(N) + length(M) + length(S) == 0 ->
  {undefined, {T0, X0, P0}};
true ->
  Dt = (T1 - T0)/1000,
  F = fun([_, _, [O], [W], [Radius]]) -> [
    [Radius*math:cos(O)],
    [Radius*math:sin(O)],
    [O+W*Dt],
    [W],
    [Radius]] end,
  Jf = fun([_, _, [O], _, [Radius]]) -> [
    [0,0,-Radius*math:sin(O),0,math:cos(O)],
    [0,0,Radius*math:cos(O),0,math:sin(O)],
    [0,0,1,Dt,0],
    [0,0,0,1,0],
    [0,0,0,0,1]] end,
  Q = mat:zeros(5,5),
  H = fun([[X], [Y], [O], [W], [Radius]]) ->
    [[Radius*W*W] || _ <- N] ++
    [[W] || _ <- N] ++
    [[shortest_path(-OZ, O)] || [OZ] <- M] ++
    [[dist({X,Y},{Px,Py})] || [_,Px,Py] <-S] end,
  Jh = fun([[X], [Y], _, [W], [Radius]]) ->
    [[0,0,0,2*Radius*W,W*W] || _ <- N] ++
    [[0,0,0,1,0] || _ <- N] ++
    [[0,0,1,0,0] || _ <- M] ++
    [[dhdx({X,Y},{Px,Py}),dhdx({Y,X},{Py,Px})
     ,0,0,0] || [_,Px,Py] <- S] end,
  Z = [[-Ay] || [Ay,_] <- N] ++
      [[-Gz] || [_,Gz] <- N] ++
      [[-O] || [O] <- M] ++
      [[Range] || [Range,_,_] <- S],
  R = mat:diag(
    [?VAR_A || _ <- N] ++
    [?VAR_G || _ <- N] ++
    [?VAR_M || _ <- M] ++
    [?VAR_S || _ <- S]),
  {X,P}=kalman:ekf({X0,P0},{F,Jf},{H,Jh},Q,R,Z),
  {ok, lists:append(X), {T1, X, P}}
end.

dist({X,Y}, {Px,Py}) ->
  math:sqrt(math:pow(X-Px,2)+math:pow(Y-Py,2)).

dhdx({X,Y}, {Px,Py}) ->
  D = dist({X,Y}, {Px,Py}),
  (X-Px) * math:sqrt(D) / D.

shortest_path(Z, O) ->
  NewO = math:fmod(O, 2*math:pi()),
  case abs(Z-NewO+2*math:pi()) < abs(Z-NewO) of
    true -> NewO - 2*math:pi();
    false -> NewO
  end.
```

Listing 2. Complete sensor fusion model of the first phase (Erlang code)

under hardware failure because this would be similar to removing sensors.

6 Second Phase: A MEMS AHRS

In the first phase, we analyzed the feasibility of sensor fusion with Hera and GRiSP. In the second phase, we develop 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 ≈ 3.75 Hz.

6.1 Orientation Estimation

We can determine the orientation from the direction of gravity and the north [3]. 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 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 (5).

$$\begin{aligned} \text{East} &= \hat{\text{Down}} \times \hat{m} \\ \text{North} &= \hat{\text{East}} \times \hat{\text{Down}} \\ \text{DCM} &= \begin{pmatrix} \hat{\text{North}}^T & \hat{\text{East}}^T & \hat{\text{Down}}^T \end{pmatrix} \end{aligned} \tag{5}$$

As shown in [19], it is possible to build a Kalman filter with the DCM, but we use the quaternion representation because this approach requires less operations. Furthermore, we cannot use Euler angles because they are ambiguous (i.e. the same orientation can be represented with different combinations). The DCM can be converted into a quaternion with (6) [6].

$$\begin{aligned} q_1^2 &= \frac{1}{4}(1 + R_{11} + R_{22} + R_{33}) \\ q_{am} &= \frac{1}{4q_1} \begin{pmatrix} 4q_1^2 \\ R_{32} - R_{23} \\ R_{13} - R_{31} \\ R_{21} - R_{12} \end{pmatrix} \end{aligned} \tag{6}$$

At this point, we add the gyroscope data in the quaternion-based Kalman filter (8) inspired from [7]. 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

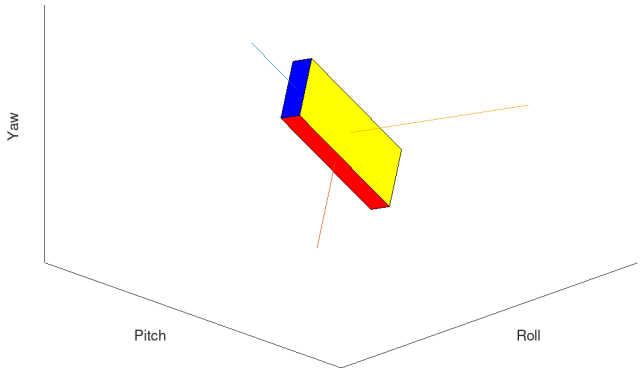

Figure 17. Orientation in real time on the visualization tool

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 found by trial-and-error and we did not optimize these parameters as much as possible because we only focus on demonstrating feasibility.

$$\Omega(\omega) = \begin{pmatrix} 0 & \omega_x & \omega_y & \omega_z \\ -\omega_x & 0 & -\omega_z & \omega_y \\ -\omega_y & \omega_z & 0 & -\omega_x \\ -\omega_z & -\omega_y & \omega_x & 0 \end{pmatrix} \tag{7}$$

$$\begin{aligned} F &= I_{4x4} + \frac{\Delta t}{2}\Omega(\omega) \\ H &= I_{4x4} & Z &= q_{am} \\ Q &= \sigma_Q^2 I_{4x4} & R &= \sigma_R^2 I_{4x4} \end{aligned} \tag{8}$$

However, the sign of the predicted quaternion $\hat{q}^-$ may be changed such that (9) is valid, to ensure that the difference computed by the Kalman filter is the smallest [5]. Moreover, the estimated quaternion requires a normalization to avoid distortions due to small computing errors.

$$q_{am} \cdot \hat{q}^- > 0 \tag{9}$$

6.2 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 compare the observation model provided by the DCM from (5) with the inertial prediction model F from (8) and the complete quaternion-based Kalman filter.

Quantitative demonstration whether the estimation is smooth and correct is difficult. During the experiments, we visualized the rotation in real time with our modular visualization tool (Fig.17) and a video demonstration is available at [10]. 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.

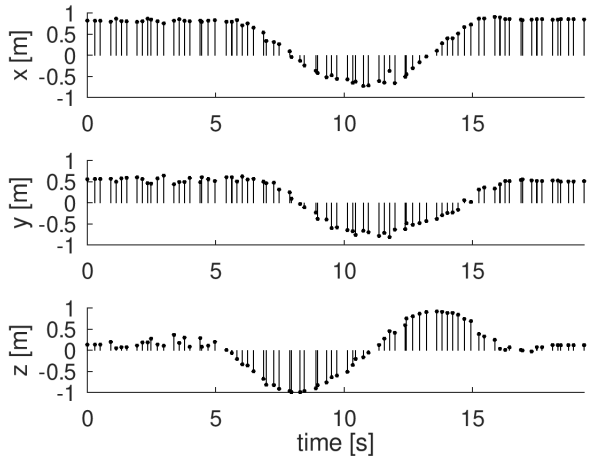

Figure 18. Orientation from the accelerometer and the magnetometer

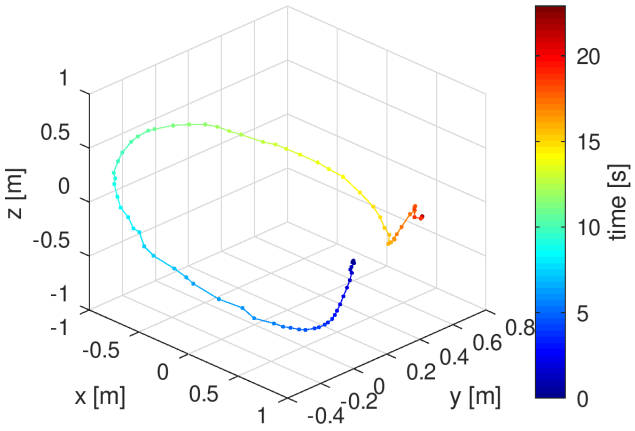

Figure 19. Orientation with a gyroscope only

On Fig.18, you can see the orientation provided by the DCM from (5) with the accelerometer and the magnetometer. It is shaky and unsmooth.

Fig.19 gives the orientation from the inertial prediction model of the quaternion-based Kalman filter (without the accelerometer nor the magnetometer). The gyroscope really helps to predict the next orientation with a lot of smoothness, but after a complete 360° rotation, we see that it is subject to drift.

From Fig.20, it is clear that the Kalman filter allows to combine the absolute information of the magnetometer and accelerometer with the smooth inertial prediction of the gyroscope to get the best of both without the disadvantages.

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

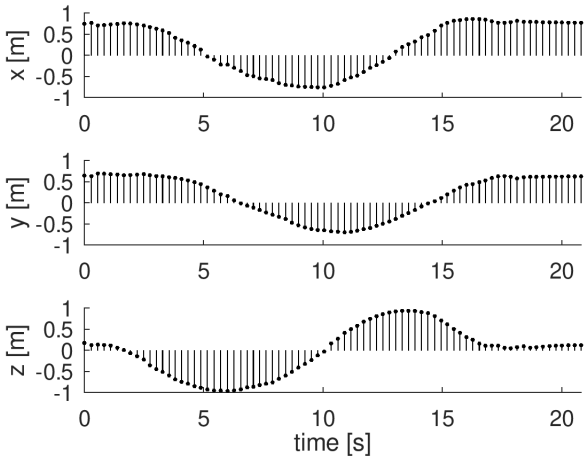

Figure 20. Orientation from the fusion of accelerometer, magnetometer, and gyroscope via Kalman filter

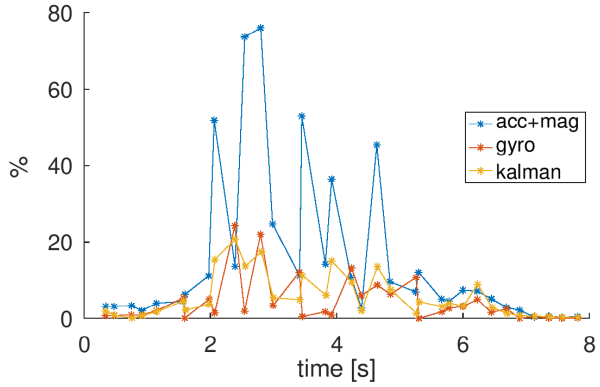

Figure 21. Deviation from previous position while shaking

motion, the DCM from (5) 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.21 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, but also reveals that the inertial model 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.

7 Conclusion and Future Work

The Hera framework offers a high-level approach for low-cost and fault-tolerant sensor fusion 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 will be directly usable to achieve a higher update frequency and increased accuracy. We now give some of the possibilities for future work based on Hera.

Performance Improvements: 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 operations. Despite these limitations, it provides good accuracy and is fast enough for soft real-time applications. The GRiSP 2, a second-generation GRiSP board, should be available in July 2021 and will provide a 20× improvement in computation speed. Furthermore, we are working on a native matrix library for Erlang [13] that will provide an additional 10× to 100× improvement in speed for the matrix operations of the Kalman filter. In addition, we expect improved Pmod sensor drivers to become available.

Combination with Machine Learning: Libraries for machine learning (ML) and data mining give new abilities, such as motion recognition. Asynchronous algorithm for ML like [17] could enrich the Hera library. It is clear that these algorithms require increased computation speed, but this should become possible with the planned improvements.

Targeting Rugged Terrains: With 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.

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 exist Pmod actuators for controlling the external world, like proportional motor control.

Simple and easy to use, the Hera framework is perfect for IoT prototyping. With its interesting properties, it offers a good basis for sensor fusion at the extreme edge. The application used for the examples of this paper and Hera are also freely available as open-source software on Github[15,16,17] and can be used with GRiSP-Base boards from Stritzinger GmbH. The interested reader may find additional background information regarding the subjects of this paper in [11].

[15] https://github.com/sebkm/sensor_fusion
[16] https://github.com/sebkm/hera
[17] https://github.com/sebkm/hera_synchronization

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.

Acknowledgments

We thank Peer Stritzinger for his help and the anonymous reviewers for their comments.

References

[1] Puput Dani Prasetyo Adi and Akio Kitagawa. 2019. 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 (2019), 10.

[2] Digilent. 2020. *Pmod™ Interface Specification.* National Instruments.

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

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

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

[6] Jay A Farrell. 2015. *Computation of the Quaternion from a Rotation Matrix.* Technical Report. University of California. 2 pages.

[7] Kaiqiang Feng, Jie Li, Xiaoming Zhang, Chong Shen, Yu Bi, Tao Zheng, and Jun Liu. 2017. A new quaternion-based Kalman filter for real-time attitude estimation using the two-step geometrically-intuitive correction algorithm. *Sensors* 17, 9 (2017), 6. https://doi.org/10.3390/s17092146

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

[9] Junjun Guo, Xianghui Yuan, and Chongzhao Han. 2017. Sensor selection based on maximum entropy fuzzy clustering for target tracking in large-scale sensor networks. *IET Signal Processing* 11, 5 (2017), 613–621. https://doi.org/10.1049/iet-spr.2016.0306

[10] Sébastien Kalbusch and Vincent Verpoten. 2021. Demonstration of AHRS. https://www.info.ucl.ac.be/~pvr/Hera-demo.mp4.

[11] Sébastien Kalbusch and Vincent Verpoten. 2021. *The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking.* Master's thesis. UCLouvain. http://hdl.handle.net/2078.1/thesis:30740

[12] LightKone. 2020. *Lightweight Computations on the Edge.* European Union's Horizon 2020 research and innovation programme under grant agreement No 732505. https://www.lightkone.eu/

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

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

[15] Muhammad Muzammal, Romana Talat, Ali Hassan Sodhro, and Sandeep Pirbhulal. 2020. A multi-sensor data fusion enabled ensemble approach for medical data from body sensor networks. *Information Fusion* 53 (2020), 155–164. https://doi.org/10.1016/j.inffus.2019.06.021

[16] Guillaume Neirinckx and Julien Bastin. 2020. *Sensor fusion at the extreme edge of an internet of things network.* Master's thesis. UCLouvain. http://hdl.handle.net/2078.1/thesis:26491

[17] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I Jordan, Kannan Ramchandran, Chris Re, and Benjamin Recht. 2016. Cyclades: Conflict-free asynchronous machine learning. https://arxiv.org/abs/1605.09721.

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

[19] William Premerlani and Paul Bizard. 2009. *Direction Cosine Matrix IMU: Theory.* Technical Report. DIY DRONE: USA. 1–30 pages.

[20] Jessica Velasco, Leandro Alberto, Henrick Dave Ambatali, Marlon Canilang, Vincent Daria, Jerome Bryan Liwanag, and Gilfred Allen Madrigal. 2020. Internet of things-based (IoT) inventory monitoring refrigerator using arduino sensor network. *Indonesian Journal of Electrical Engineering and Computer Science* 18, 1 (04 2020), 508–515. https://doi.org/10.11591/ijeecs.v18.i1.pp508-515

[21] Greg Welch and Gary Bishop. 1995. *An Introduction to the Kalman Filter.* Technical Report. University of North Carolina at Chapel Hill.

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

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