UCLouvain

École polytechnique de Louvain

Low-cost high-speed sensor fusion with GRiSP and Hera

Author: **Lucas NÉLIS**
Supervisor: **Peter VAN ROY**
Readers: **Peer STRITZINGER, Ramin SADRE**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

Internet of Things applications using sensor fusion are increasingly numerous. Unfortunately, performing sensor fusion can require computational capabilities which can be a limiting factor for low-cost IoT devices. On top of that, some of these applications try to react in real-time to their environment making low latency a key element for them. These real-time applications need to perform sensor fusion at the edge in order to reduce latency and as fast as possible to react in time. Optimising the computations and the latency of these applications becomes their main challenge. Hera, a fault-tolerant low-cost sensor fusion framework, had been developed to provide users with a high-level ready-to-use sensor fusion tool. It has been developed and tested on GRiSP-Base boards, an Erlang based low-cost IoT device. However, Hera was slow and could not be used for real-time applications.

In this master thesis, I have improved Hera's speed by porting it to GRiSP2, the new version of GRiSP boards, and by improving its matrix library. Initially, the matrix library of Hera was written in Erlang which has poor performances computations. Fortunately, Erlang offers the possibility of introducing C functions through the NIF mechanism. Using Numerl, a NIF matrix library, the matrix computations of the Kalman filter, the algorithm used for sensor fusion, were sped up by a factor of ten. The combined hardware and software improvements led to a new version of Hera, Hera v2.0, which is almost 25 times faster than originally.

Acknowledgements

This master thesis is the result of multiple months of work during which I received help from various people. I would like to thank them, without whom this work would have been considerably harder.

First, I thank my family and Aline for their encouragement and support throughout the year.

Next, I thank Tanguy Losseau, the author of the Numerl library, for correcting the bugs found during its integration into Hera and for his work on the NIF wrapper for BLAS.

Then, I would like to thank Peer Stritzinger and the whole GRiSP team for their support, patience and reactivity through the project. They helped me greatly with the installation of Hera on GRiSP2 and quickly solved multiple bugs discovered during this project.

Last but not least, I would like to thank my supervisor, Peter Van Roy, for his ideas, advice and knowledge that guided me and allowed me to work more efficiently week after week. I'm thankful to have been given the opportunity to contribute to this enriching project.

Contents

Chapter 1

Introduction

1.1 Internet of Things and sensor fusion

Internet of Things or IoT was first described by Kevin Ashton in 1999 as "a system in which objects in the physical world could be connected to the Internet by sensors" [1]. Nowadays, we still use this term for devices, items or objects with Internet connectivity and computing capabilities. Most of these Things collect data on their environment through sensors which then needs to be processed. Cloud computing is a good solution for (large) data processing but it has one major drawback: sending data to a cloud center and then retrieving it when needed introduces latency. IoT applications, such as real-time applications, may need to adapt to their environment or inform another device of a change. This kind of behaviour must be as instantaneous as possible making latency its worst enemy. Thankfully, all IoT applications don't need the computational power of cloud computing, making edge computing an interesting choice [2]. Edge computing aims at processing the data as close as possible to the final user. In some cases, the processing can even be done by the IoT device itself. On top of the latency reduction, computing at the edge allows the system to be cheaper and more independent.

Sensor fusion is one way of processing the sensors' data, it combines data from multiple sources to build a coherent view of the real world, understandable by humans [3]. Being able to use this view as soon as possible is the main challenge of sensor fusion. IoT's sensors and computational ability make a good match for sensor fusion. Moreover, integrating sensor fusion into an IoT application enables it to utilise its sensor data even further. However, sensor fusion involves matrix computations which can be challenging to perform quickly. Optimising these operations is critical as too much computing time would make the computed view outdated and thus useless.

1.2 Hera: a sensor fusion framework

As more and more IoT applications emerged, allowing them to easily integrate sensor fusion was key to push the domain. Hera, a near real-time fault-tolerant and scalable Erlang sensor fusion framework capable of interacting with an IoT network [4], was created to that extent. It was tested on GRiSP-Base boards and relied only on the boards for its computations. With GRiSP's Erlang sensor drivers and Hera's high level interface, building IoT applications became less complex. Then, the work of [5] demonstrated that it was possible to use Hera for simple position and orientation tracking by adding a Kalman filter to Hera. They achieved to update the physical model of an Attitude and Heading Reference System (AHRS) at a rate of 3.75 [Hz]. At that point, Hera wasn't ready for fast real world application being limited by its slow speed.

1.3 Contributions

In this master thesis, I present Hera v2.0, an Erlang sensor fusion framework designed for IoT. It aims at performing low-cost and high speed sensor fusion at the edge of the network by improving Hera's speed.

The following improvements have been made to Hera:

- Porting Hera from GRiSP-Base boards to GRiSP2 boards achieving an update rate of 69.5 [Hz] of the AHRS physical model.
- Replacing Hera's Erlang matrix library by a C matrix library speeding up the Kalman filter loop computation time by a factor of 10.
- Integrating this new matrix library into the AHRS experiment and achieving an update rate of its physical model of 89 [Hz].

The repository of Hera v2.0 can be found at `https://github.com/lunelis/hera` and the repository of the GRiSP application used to test Hera v2.0 can be found at `https://github.com/lunelis/sensor_fusion`.

1.4 Related work

1.4.1 Low-cost Internet of Things sensor fusion

Internet of Things devices tend to be relatively cheap. These devices are usually designed to perform specific tasks such as measurements and communications. In

order to realise their tasks while maintaining a low-cost, the devices are simplified at most and compromises may be made on the performances or precision of the device. Finding the right combination of accuracy and cost can be a tricky task. In [6], combination of sensors for drone position estimations are analysed. Their performances are compared in order to provide a cost-efficiency scale.

In [7], a fall detection device for elderly people has been developed. A gyroscope, magnetometer and accelerometer are used to build an AHRS. Then, the authors use a sensor fusion filter designed for AHRS to provide an estimation of the orientation of the person. Another filter is used to fuse this orientation with the altitude measured by a barometer resulting in an accurate altitude estimation. The fall of the person, who wears the device at his waist, can be determined from the changes in the altitude estimate. The authors achieved better fall detection precision than previous systems but still fails to evaluate correctly a few situations. They didn't use a Kalman filter as the computational load was too heavy for their device and opted for a lighter sensor fusion filter. Their device is able to evaluate the fall of a person at a rate of 50 Hz with every computation being done on their device as latency cannot be tolerated in such application. They state that further improvements of the precision will need calibration of their device by each individual.

1.4.2 High speed sensor fusion at the edge

Real time applications, such as drones or automated vehicles, have to react in a dynamic and fast manner to their environment. Measurements made by such applications can have a direct impact on the device and must be treated as soon as possible. This means fusing quickly the measured data. On top of that, computations must be done at the edge of the network as the delay introduced by cloud computing or even data transfer to a nearby server may be too much [8].

A real-time sensor fusion framework for perception in autonomous vehicles has been developed in [9]. Perception is the ability to detect what is around the vehicle. Various data can be used for that purpose such as lidars, cameras, radars and sonars. The authors used a convolutional network and a Kalman filter to fuse the data from the car's sensors directly on the edge with a device inside the car. They declare that their system provides to the car a detailed map of its environment but do not give the update rate of that environment. The computational load of their algorithms isn't negligible and needs adequate hardware. They used an NVIDIA Xavier embedded computer to perform the computation which range from 500$ to 1500$ depending on the model. With the prices of the camera, lidars and radars the total price of their system exceeds 2000$.

1.4.3 Kalman filter and sensor fusion

The Kalman filter is a well-establish method for sensor fusion. It has been used in various domains thanks to its general nature. Variants of the filter have been created to adapt it to new challenges such as the extended Kalman filter for nonlinear problems.

In [10], an augmented reality application has been developed to interact with robots remotely. A motion sensor placed on the hands recognises their gestures and position. A Kinect camera sensor is used to collect motion velocity data. The data from the two sources are then fused through a traditional Kalman filter and the results are sent in real-time to the controlled robot. Participants tested the application which proved to have good accuracy at recognizing simple gestures. Their results also show that their application was more accurate when using the Kalman filter than without it.

A real-time road-object detection and tracking method for autonomous vehicles is proposed in [11]. The data from radars and lidars is fused with a customized unscented Kalman filter to obtain the precise position and velocity of the objects around the car. The unscented Kalman filter is an alternative to the extended Kalman filter that adds additional steps in order to reduce the accumulated error. The authors compared the results obtained with both filters and showed that the unscented Kalman filter got better results by a large margin. Their final application using the unscented Kalman filter was able to recognize with high accuracy bicycles, pedestrians and cars.

Chapter 2

Resources

2.1 The GRiSP project

The GRiSP project[1] aims at providing easy to use microcomputers to create Internet of Things projects without soldering or having to use the C language. It runs an open source[2] Erlang based custom operating system using RTEMS (Real-Time Executive for Multiprocessor Systems) allowing the user to interact with the device using an Erlang shell. On top of that, the board is equipped with multiple Digilent Pmod ports to connect sensors. Data can be retrieved easily from these sensors thanks to dedicated Erlang drivers. The Hera framework was tested on the first version of GRiSP boards: GRiSP-Base and has been ported for this master thesis to the second version: GRiSP2.

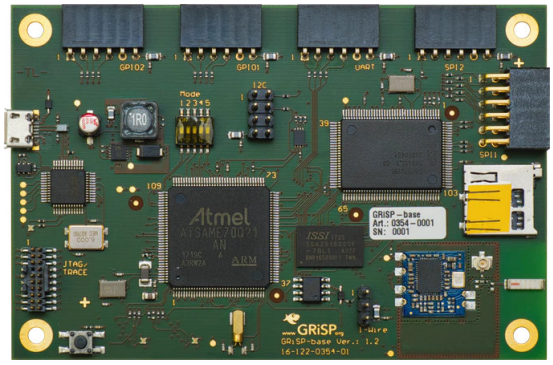

Figure 2.1: GRiSP-Base

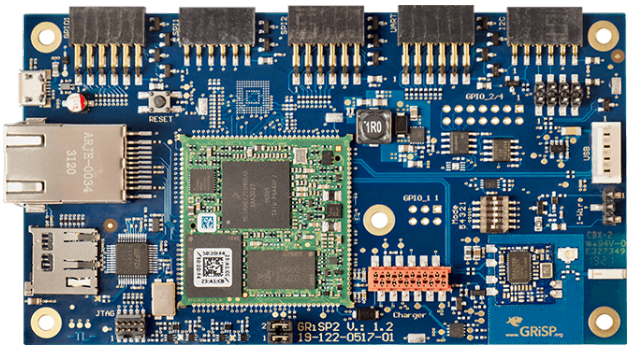

Figure 2.2: GRiSP2

[1] https://www.GRiSP.org/
[2] https://github.com/GRiSP

2.1.1 GRiSP-Base and GRiSP2 boards

The first and second version of the GRiSP boards differ mainly in hardware. The table below shows the main differences between them. GRiSP2 is a significant improvement from the GRiSP-Base thanks to its new components. This represent an improvement in computation power which will be useful in this master thesis. They are equipped with the same ports for sensors. Debugging is done via USB on both devices.

Device	**GRiSP-Base**	**GRiSP2**
CPU	Atmel SAM V71	NXP iMX6UL
Core	ARM Cortex M7	ARM Cortex-A7 with 128 KB L2 cache
CPU clock speed	Up to 300MHz	696 MHz
Memory	64 MB SDRAM	128 MB of DDR3 DRAM
Network	Wi-Fi 802.11b/g/n WLAN	Wi-Fi 802.11b/g/n WLAN 100 Mbit/s Ethernet

Table 2.1: GRiSP boards hardware comparison

2.1.2 Digilent Pmod sensors

The boards support Digilent Pmod sensors with adequate ports and drivers. These sensors can then be used to retrieve specific data. In this project, we will only use one sensor : Pmod NAV[3].

The Pmod NAV contains multiple sensors: a 3-axis accelerometer, a 3-axis gyroscope, a 3-axis magnetometer and a barometer.

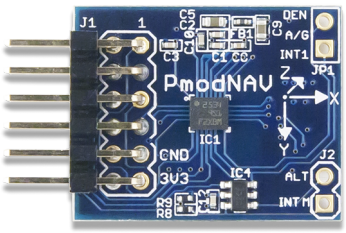

Figure 2.3: Pmod NAV

[3] https://digilent.com/reference/pmod/pmodnav/start

2.2 Erlang

The choice of GRiSP comes with Erlang which was not intended for Internet of Things applications at first but proves to be suitable and eases the development by avoiding to drop down in C. Indeed, Erlang is a functional high-level programming language that is frequently used to build networked software systems. It is designed to support distributed, concurrent, fault-tolerant and real-time applications thanks to OTP (Open Telecom Platform), a collection of libraries and tools ready to use. An application built with Erlang is composed of multiple processes that can communicate through message passing.

2.2.1 Native Implemented Functions

Erlang's Native Implemented Functions or NIFs[4] are used to integrate C functions into Erlang. They are the fastest way to call C functions from Erlang and require no switch of contexts. NIFs are linked to an Erlang module, for each C function, there will be an Erlang function. At compilation, the C function will be linked to the respective Erlang function and will become usable through it.

Using C functions instead of Erlang's can be useful to take advantage of C fast computations, especially matrix operations. However, due to NIFs implementation, NIFs do not benefit from Erlang's safety. This means that any error in a NIF could cause the crash of the Erlang virtual machine. On top of that, Erlang's processes can be paused and resumed by the schedulers, which is not the case of C functions executing outside of the virtual machine. For this reason, the C function should not run for more than a millisecond to avoid load balancing issues between the schedulers.

2.3 Kalman filter

The Kalman filter was first introduced in [12] as a recursive solution to the discrete data linear filtering problem. With the evolution of computer science, it has been largely studied and used for data fusion. Its goal is to estimate the state of a process $\hat{x}_k$ at time k by going through two independent phases. First, it predicts the state of the process by projecting in time the current state. Then, it corrects these predictions with a measurement. The estimates obtained after the prediction are called the *a priori* estimates and are recognisable by the $-$ superscript. The ones obtained after the correction are called *a posteriori* estimates [13].

[4] `https://www.erlang.org/doc/tutorial/nif.html`

In this document, you will find the following notations concerning the Kalman filter:

- $\hat{x}_k^-$: the a priori state estimate
- P_k^- : the a priori estimate covariance matrix
- F_k : the state-transition model
- $\hat{x}_k$: the a posteriori state estimate
- P_k : the a posteriori estimate covariance matrix
- B : the control-input model
- u_k : the control vector
- Q_k : the covariance of the process noise
- K_k : the optimal Kalman gain
- H_k : the observation model
- R_k : the covariance of the observation noise
- z_k : the measurement vector
- I : the identity matrix

2.3.1 Prediction phase

In this master thesis, we will focus on the discrete Kalman filter without control input. Thus, for the rest of this document, we will assume that $Bu_k = 0$. The prediction or propagation phase uses the last $\hat{x}_k$ and F_k to compute $\hat{x}_k^-$ (2.1). Then, P_k^- is computed based on P_k, F_k and Q_k (2.2). F_k is related to the process and describes how its state is expected to evolve over time. For example, a free fall body is expected to fall according to gravity and air resistance. It is up to the user to describe F_k as precisely as possible. Q_k or process noise can be measured prior to the launch of the filter and used as a constant.

$$\hat{x}_{k+1}^- = F_k\hat{x}_k + Bu_k \tag{2.1}$$

$$P_{k+1}^- = F_kP_kF_k^T + Q_k \tag{2.2}$$

2.3.2 Correction phase

The correction or update phase is executed after the prediction and corrects the prediction with a measurement z_k. Therefore, there must be a prediction in order to compute the *a posteriori* state. First, the Kalman gain is computed using P_k^-, H_k and R_k (2.3). R_k can be computed before the launch of the filter similarly to Q_k.

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{2.3}$$

The Kalman gain can then be used with measurement(s) z_k, H_k and the *a priori* state estimate to compute the *a posteriori* state estimate (2.4). Measurements z_k represent measured change in the process, this can be data measured by sensors or any other type of input data. Finally, P_k is computed using P_k^-, H_k and K_k (2.5)

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k\hat{x}_k^-) \tag{2.4}$$

$$P_k = (I - K_k H_k)P_k^- \tag{2.5}$$

2.4 Numerl : a matrix computation library

Numerl is an Erlang matrix computation library using NIFs [14]. It provides the user with a set of basic vector and matrix operations. The library uses custom structure to represent the matrices in C. It is composed of the dimensions of the matrix and an array of double. This array of double represents the matrix in a row major format. This custom representation combined with the speed of the C language allows the computations to be from 6 to 17 times faster than their native counterparts depending on the operation.

2.4.1 Performances comparison

The author of Numerl compared the speed of its library with Hera's Erlang matrix library. For each matrix operation, the computation time was compared for multiple matrix sizes on a general purpose computer. The author concluded that Numerl was faster than Hera's matrix library by a factor of:

- 12 for addition of matrices
- 6 for multiplication of a matrix by a number
- 12 for multiplication of matrices
- 8 for transposition of matrices

- 17 for inversion of matrices

These factors vary depending on the size of the matrix. Indeed, the bigger the matrix was the bigger the factor between the two computation time. In this master thesis, most of the matrices used are of size 4x4. On top of that, Numerl will be running on GRiSP boards which differs a little from a general purpose computer. For these reasons, comparisons between Numerl and Hera's matrix library will be made again on GRiSP boards in 5.3.

Chapter 3

Hera: structure

Hera is an Erlang/OTP application designed for sensor fusion developed by the two successive master thesis [4] and [5]. It supports distributed networks and concurrency in order to build networks of Internet of Things devices. These devices can collect various data about their environment which will then be fused through Hera according to the specifications of the user. Hera is designed to be easy to use and experiment with in order to be used in academic or industrial contexts. Even though Hera has been developed and tested with GRiSP board, it is not limited to them and could be used on any IoT device supporting Erlang. The most recent version of Hera can be found at `https://github.com/sebkm/hera` and will be the starting point of this master thesis.

3.1 A fault tolerant sensor fusion

One of Hera's main strength is its fault tolerance. Hera's strategy is to collect sensor data on each node of the network, broadcast it through UDP to the other nodes and receive their data, fuse the gathered data at each node and finally exchange the final result. If there is a node crash or data loss, the data fusion can still be performed without the sensor data missing from this node. The result will be less precise but still usable.

This approach achieves fault tolerance by redundancy and dynamism. Redundancy is ensured because every node performs the sensor fusion computation. Crashing nodes receive the state of the system right after restart allowing them to restart faster. Dynamism, on the other hand, is achieved by adapting the sensor fusion computation to the data received. Indeed, every node may have different sensor data as some could have been lost through UDP broadcast or because a node has crashed while broadcasting. Each node could then end up with slightly

different result as they may not have the same data. But these results would still be close from the reality and usable.

3.2 Supervision tree

Erlang applications are built using different independent processes that interact with each other through message passing. Some of these processes are more prone to crashing and need to be restarted as soon as possible. These processes can be supervised by a supervisor which will watch over them and react to their crashes. The supervised processes will be called children. A process can be a supervisor and a child at the same time. Processes that are not supervisors are called workers. The supervision tree is the structure used to represent this organisation.

In Hera, the supervision tree is relatively simple, `hera_sup` is the highest supervisor and watches over the `hera_com`, `hera_data` and `hera_measure_sup` processes. `hera_measure_sup` is also a supervisor and watches over the `hera_measure` processes defined by the user. Figure 3.1 shows the relations between the processes with rectangles representing supervisors, circles representing workers and arrows going from a supervisor to their children. Two `hera_measure` processes are represented on the figure but this number can vary, without any the node would act as a database and with too many the node would be too slow to be useful.

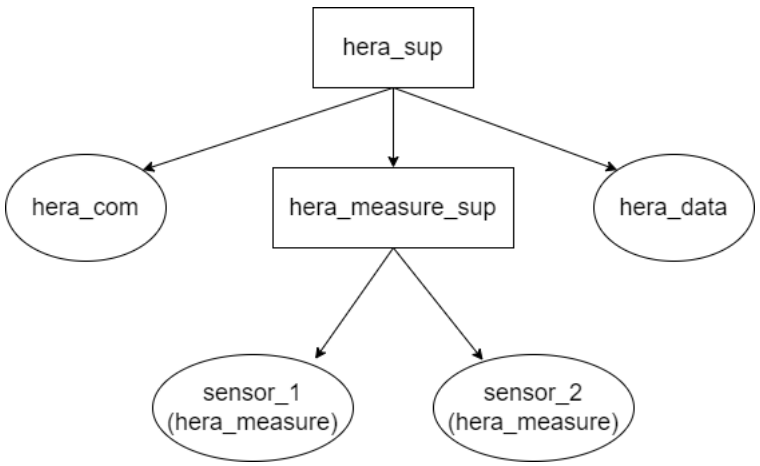

Figure 3.1: Hera supervision tree

3.3 Processes

3.3.1 `hera_measure`

The `hera_measure` module defines an Erlang behaviour for the measurement processes of the user. For each module following the `hera_measure` behaviour, the user must provide a `measure/1` function containing the instructions for an iteration and a `init/1` function returning the initial state of the system and containing the following informations:

- a name
- the number of iteration threshold after which the process must stop
- a timeout value in millisecond representing the time the process will wait after each iteration
- a boolean indicating if the process must be synchronized with others

When launched by `hera_measure_sup`, the user's processes will initialise themselves thanks to the `init/1` function. Then, they will call their `measure/1` function, send the results through the `hera_com` process and wait their timeout. The process will then repeat these three operations until the number of iteration threshold is reached. If a `hera_measure` process crashes, it is restarted by its supervisor and initialises itself again before starting the measurements.

3.3.2 `hera_com`

Because Hera supports distributed networks, a node needs to be able to communicate with other nodes in the network. `hera_com` is responsible of receiving and sending messages to the other nodes of the network using multi-cast UDP. Since UDP is not reliable, messages can be lost, resulting in a difference in information between the nodes.

Other processes of the node can use the `send/3` function to broadcast data to other nodes but also to themselves. Because multiple `hera_measure` processes can be launched on the same node, communicating the results to itself is useful in order to not miss any sensor data. When sensor data is received by `hera_com` processes, it is stored through `hera_data`.

3.3.3 `hera_data`

Using data from other nodes is a central concept in sensor fusion, making storing and accessing data easily crucial. The `hera_data` process is responsible for these

tasks. It has been implemented as a `gen_server` to ease the request handling. The data for each `hera_measure` and its node are stored in a map with the `store/4` call. Retrieving the data from a specific node and measurement process can be done using the `get/2` call. Additionally, sensor data can be stored under `.csv` format to be conserved by setting the `log_data` environment variable to true.

3.4 Interactions between nodes

When building an application with Hera, multiple scenarios can occur. There may be one or more nodes to the network and each of these nodes may have from one to a few `hera_measure` processes. Each node will also have its own `hera_data` and `hera_com` processes. Figure 3.2 describes how a measure is communicated from one node to another [5].

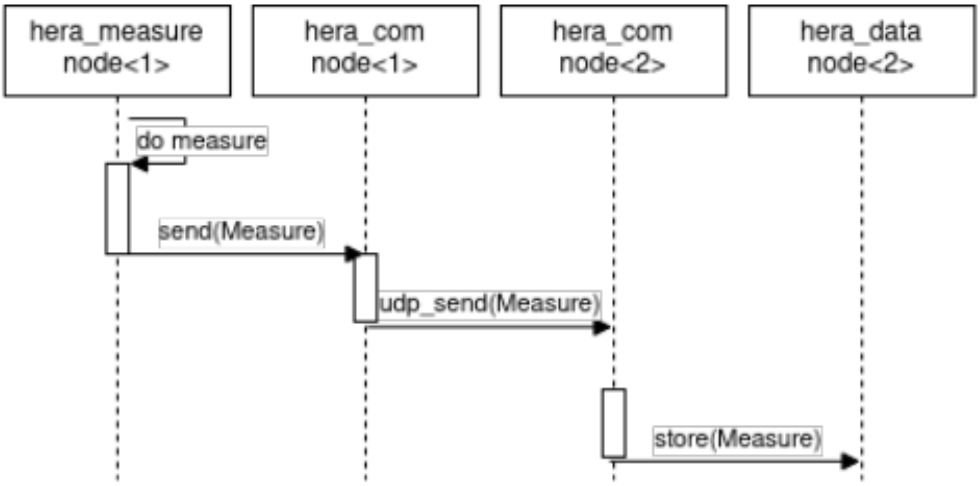

Figure 3.2: Interactions between two nodes

First, a measure is taken by `node<1>`. Its `hera_measure` process sends it to its `hera_com` process. Then, the measure gets broadcasted to every node in the network. In this example, `node<1>` broadcasts the measure to `node<2>` and `node<1>`. The figure 3.2 describes how the message is then received by the `hera_com` of the other nodes and transferred to `hera_data` to be stored.

When performing sensor fusion, node often need to perform computations on other nodes' data. Figure 3.3 shows how `node<2>` can use the data previously received. It first asks for the data of `node<1>` from its `hera_data` process. Performs

the computations and then sends the results to `hera_com` which broadcasts it to the other nodes.

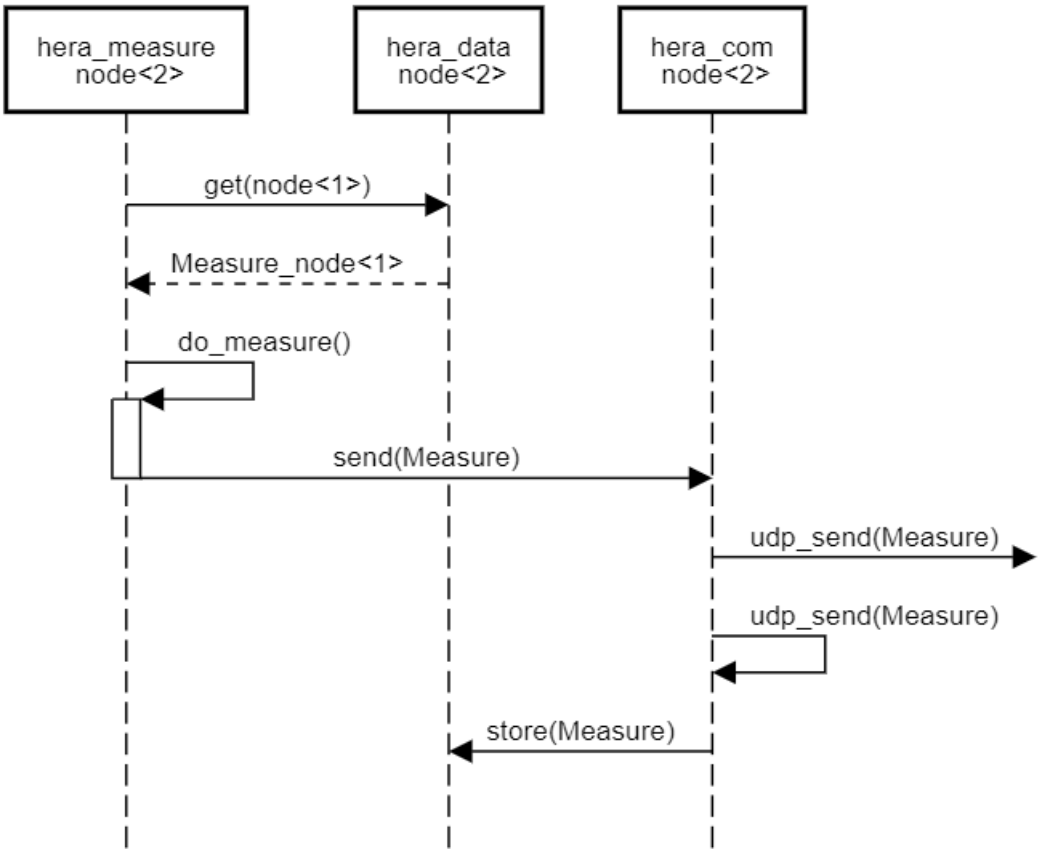

Figure 3.3: Using data from another node

Chapter 4

From GRiSP-Base to GRiSP2

Hera had not been updated for over a year. In the meantime, the GRiSP project continued to grow both on hardware and software level. In order to continue the development of Hera, it was necessary to update its dependencies and make it run on the latest hardware available. Indeed, since Hera's last update, the GRiSP2 board came out as an upgrade to the GRiSP-Base board. Fortunately, this upgrade was mainly on the hardware levels, meaning that no change had to be made to Hera. The following changes concern the sensor fusion GRiSP application[1] used to show Hera's capabilities. The GRiSP2 version of the sensor fusion application without the changes of chapter 5 can be found in the `grisp2_basic` branch of the repository.

4.1 Dependencies

In order to compile and deploy applications, GRiSP relies on rebar3, a build tool and package management tool for Erlang. Hera's first version recommended the use of `rebar3` 3.13 and Erlang/OTP 22.0 due to GRiSP's support at the time. As GRiSP supports more recent versions, `rebar3` 3.18 and Erlang/OTP 25.0 are recommended. GRiSP supports `rebar3` up to its latest version 3.21 but due to a problem with the `relx` library at the time, `rebar3` 3.18 was the latest stable version for GRiSP. Using newer versions of `rebar3` should not be a problem for the application as long as it's supported by GRiSP.

4.1.1 `rebar3` packages

Two rebar packages are needed to compile the application correctly, `rebar3_hex` and `rebar3_grisp`. The required versions for these packages were 6.9.6 and

[1] `https://github.com/lunelis/sensor_fusion`

1.3.0 respectively. The updates for `rebar3_grisp` are mandatory for the proper functioning of the GRiSP2 board and Hera. Indeed, during the testing of Hera on GRiSP2, it was discovered that sensor emulation was broken. Its latest version (2.5.0) fixes sensor emulation without which Hera cannot function properly. The version used for `rebar3_hex` is less important, 7.0 or above can be used.

4.1.2 Custom GRiSP library

The previous version of the sensor fusion application[2] used a custom version of the GRiSP Erlang Runtime Library[3] because the library's driver for the Pmod MAXSONAR didn't support the single mode. Since the Pmod MAXSONAR is not used in this master thesis and for simplicity and maintenance reasons, the official version of the library was used instead. This ensures that the application will get the latest updates without having to manually update the custom library.

The mode of the Pmod MAXSONAR can still be useful and a pull request is open to include the mode in the official version[4]. The pull request only concerns the driver for the sensor and should therefore not be abandoned as the sensor's driver didn't had to be updated for GRiSP2.

4.2 Network settings

When building the application, GRiSP pulls configuration files such as network files depending on the GRiSP board version. Previously the configuration files for the node network and internet (`erl_inetrc`, `grisp.ini.mustache`, `wpa_supplicant.conf`) were located at `grisp/grisp_base/files`. These were moved to `grisp/grisp2/common/deploy/files`. The content of the files was adapted to fit the new IP addresses of each node but not other parameter was modified.

4.3 Port changes

The GRiSP2 board ports are not exactly the same than its predecessor. On top of the added Ethernet and USB ports, the ports for the sensor have slightly changed. The UART, GPI01 and SPI2 ports went from 6 pin to 12. The SPI1 port went from 12 pin to 6. The GPI02 port got removed. These changes affect the GRiSP applications as sensors have to be explicitly declared with the `grisp:add_device/2`

[2] `https://github.com/sebkm/sensor_fusion`
[3] `https://github.com/grisp/grisp`
[4] `https://github.com/grisp/grisp/pull/78/files`

call to indicate at which port they are connected.

In this master thesis, the Pmod NAV is used and has 12 pins. It was changed from the SPI1 port to the SPI2 port. The following line was the only one that needed to be modified to use the sensor fusion application on GRiSP2.

```
grisp:add_device(spi2, pmod_nav)
```

Chapter 5

Speeding up matrix computations

As an Erlang framework, Hera is equipped with a matrix computation module called `mat` in order to realise the computations related to the Kalman filter but also to provide the user with common matrix operations. As demonstrated in [14], matrix operations in Erlang are much slower than in C, especially when dealing with larger matrices. In order to reduce the computation time of Hera, replacing the Erlang matrix computations by C matrix computations is crucial. Numerl[1] is an Erlang NIF library implementing all the matrix operations needed in C and providing an interface in Erlang. It can be integrated in GRiSP2 with some adaptation due to GRiSP environment limitations with NIFs.

5.1 Numerl integration in GRiSP2

GRiSP applications limits the use of NIFs to statically linked NIFs because Erlang on RTEMS cannot load dynamic libraries. Thus, NIFs must be compiled with GRiSP's Erlang Runtime System to be used. This makes the deployment of a new version of the application time consuming. In order to integrate Numerl into GRiSP, the following requirements had to be followed:

1. Have a main C file named numerl_nif.c located at grisp/grisp2/common/build/nifs/numerl_nif.c
2. Have the C macro STATIC_ERLANG_NIF defined to 1 at the top of numerl_nif.c
3. Set the name of the driver and module in the call to ERL_NIF_INIT to numerl

[1] `https://github.com/tanguyl/numerl`

4. Have an Erlang module named numerl.erl

Because Numerl was built as a NIF library, the C and Erlang code of Numerl were both already contained into one file each, step 1 and 4 consisted in copying the files at the right places. Step 2 is straightforward but important as the NIF must be recognised as static in order to be compiled into GRiSP. Step 3 allows to link the C functions with the Erlang functions.

Numerl's last version used the openblas and lapacke C libraries for the inversion and matrix multiplication operations, adding these libraries to the Runtime Library is less trivial than it seems. At the time of development, the GRiSP team was working on adding openblas to GRiSP. In the meantime, the inversion and matrix multiplication operation implementations were replaced by native C implementations present in earlier versions of Numerl. The code used for the Numerl NIF can be found in the appendix A.

5.2 Modifications to Hera

The `mat` module of Hera is still useful as it acts as an interface for the rest of the framework. The specifications and definitions of its functions were untouched. The Erlang implementations were mostly replaced by Numerl calls (see appendix B). Because Numerl uses a different matrix representation (see 2.4), there needed to be a function that translated an Erlang matrix into a Numerl one and inversely. Two functions were added in order to use Numerl correctly. `matrix/1` takes an Erlang matrix, which is defined as a list of list of numbers, as input and returns a Numerl matrix. This is a mandatory operation as Numerl matrix operations can then be performed on Numerl matrices. The function can be used like this:

```
1> M = mat:matrix([[1,2,3,4],
                   [4,1,2,3],
                   [3,4,1,2],
                   [2,3,4,1]]).
```

`to_array/1` takes a Numerl matrix as input and returns an Erlang array. This operation is useful for accessing the whole matrix's content at once or to use a list comprehension. The Numerl matrix M can be tranformed back to Erlang like this:

```
2> N=mat:to_array(M).
[1.0,2.0,3.0,4.0,4.0,1.0,2.0,3.0,3.0,4.0,1.0,2.0,2.0,3.0,
 4.0,1.0]
```

The drawback of this operation is that it flatten the matrix, this can be a problem if the dimensions of the matrix are unknown to the user. An additional function

could be added to solve that issue but no use was found for such function in this project. Indeed, accessing specific row, column or element in the matrix should be done using the `row/2`, `column/2`, `get/3` and `at/2` functions.

All the other modules of Hera didn't receive a change after the Numerl integration. Indeed, the only other module relying on the `mat` module is the `kalman` module. Because the interface of the `mat` module stayed the same and because no matrix are created from scratch in the `kalman` module, no changes were needed. This is the main advantage of using the `mat` module in Hera and should be kept that way for maintenance purposes if a faster alternative to Numerl was found.

5.3 Numerl's performances

Numerl was created to provide Erlang with a fast computation library. It was tested on a general purpose computer and obtained good results as seen in 2.4.1. The GRiSP environment being different from a general purpose computer, testing Numerl on GRiSP could end up in slighly different results. It has been demonstrated by [14] that the inversion operation is the slowest and it is used once per Kalman filter loop, therefore performing it quickly is important. In order to compare the Erlang implementation with the Numerl one, the inversion operation will be performed with both implementation on 500000 random matrices on a GRiSP2 board. Figure 5.1 and 5.2 show the respective performances of the Erlang matrix library and Numerl.

Two observations can be made from these tests. Firstly, the gaps in computation time between the two implementation is very similar to the one observed in [14]. Secondly, when increasing in matrix size, the factor between the two implementations increases. For the 3x3 matrices, the Numerl implementation is about 10 times faster, this factor goes up to 23 for the 5x5 matrices. This increase of factor difference is observed for every operation at various scales. This leads to the conclusion that the differences observed in [14] are similar to the ones in a GRiSP environment.

5.4 Limitations of NIFs

In order to use NIFs with GRiSP, they have to be compiled in the GRiSP Erlang Runtime System. Thus, the NIFs files must be included in the GRiSP application. In our case, Numerl's files are included in the sensor fusion GRiSP application not in the Hera Erlang application. Despite that, Hera is calling the `numerl` module of

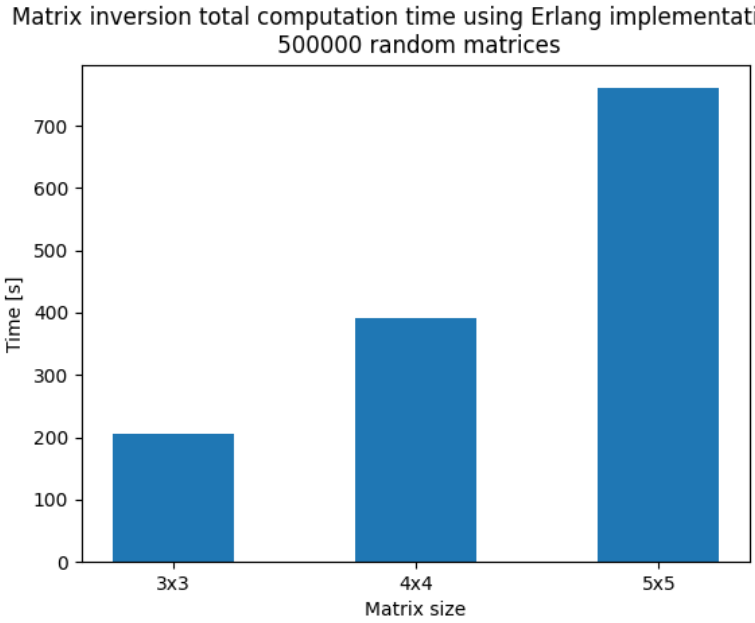

Figure 5.1: Pure Erlang matrix inversion performances on GRiSP2

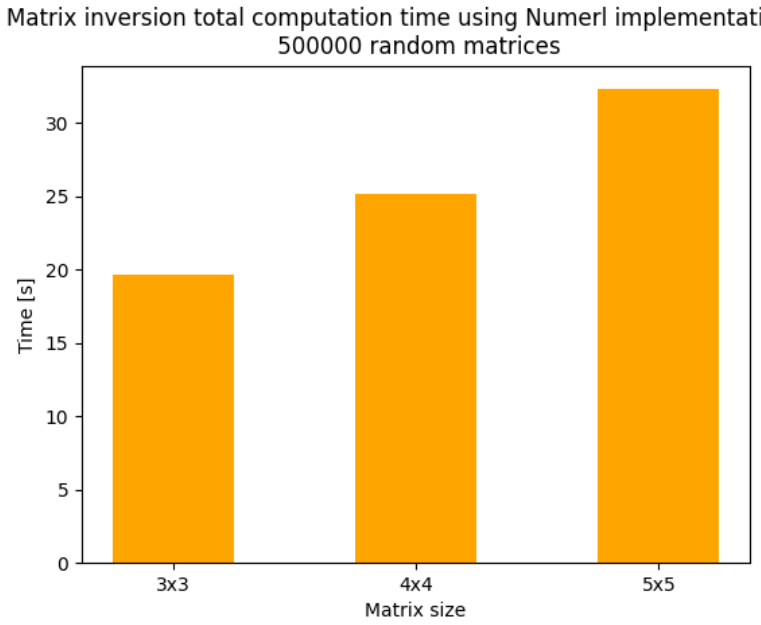

Figure 5.2: Numerl matrix inversion performances on GRiSP2

the sensor fusion application. This means that any GRiSP application wanting to use Hera must copy the files of Numerl correctly to have Numerl compiled as NIF. In order to still leave some flexibility to the user, the older matrix module written in Erlang is still available under the name `oldmat` (see appendix B.2). Ideally, Numerl should be included into Hera, but this is not possible as only static NIFs can be used inside the GRiSP environment.

Because the NIFs have to be compiled into GRiSP Erlang Runtime System, each time a modification is made to the C files of the NIFs, the whole Runtime System must be recompiled. This process can take up to one hour depending on the method used. For this project, the compilation was done using a docker image which speeds up the process.

5.5 Correcting Numerl

After the integration of Numerl in the `mat` module of Hera, unit tests were also updated to verify the correctness of the updated `mat` module. These tests were carried on a general purpose computer before testing on GRiSP. At the time, the version tested was still using the openblas library for the dot product with the function `cblas_dgemm`. The tests related to the dot product operation showed that the results were incorrect for the following `numerl` calls:

```
C = numerl:matrix([[1,2], [3,4]]),
D = numerl:matrix([[5], [6]]),
R = numerl:dot(C,D).
```

R was equal to $[[5], [15]]$ when it should have been equal to $[[17], [39]]$. To compute the dot product, the `cblas_dgemm` function of the openblas library was used. The LDB parameter value used was the number of rows of D when it should have been the number of columns of D.

Correcting this error lead to finding another one in the same call. For the following `numerl` calls:

```
C = numerl:matrix([[1,2], [3,4]]),
D = numerl:matrix([[5,6], [7,8]]),
R = numerl:mtfli(numerl:dot(C,D)).
```

The value of R was not always the same. Sometimes being correct ($[[19, 22], [43, 50]]$), sometimes being incorrect ($[[19, 22], [43, 56]]$ or $[[19, 22], [43, -2147483648]]$). The dot product of `numerl` takes two matrices as input, but `cblas_dgemm` takes three matrices and other parameters as input to compute $C := alpha*op(A)*op(B)+beta*C$. The input matrices of the numerl call correspond to the matrices A and B. The

matrix C is not initialized but memory is allocated for it, this memory could have non zero values. The *beta* argument was wrongly assigned to one, giving the matrix C an impact on the computation if it was not a zero matrix. This error was corrected by setting $beta = 0$. The Github issue concerning these two errors can be found at: `https://github.com/tanguyl/numerl/issues/16`.

The correction of these errors did not matter as the version of Numerl used for the project couldn't make use of the openblas library. However, with the integration of the blas library[2] into GRiSP, replacing the dot product and inversion C implementation by openblas calls would speed up the matrix computations even further.

[2] `https://github.com/erlef/blas`

Chapter 6

Experimental speed comparison

After porting Hera to GRiSP2 and upgrading its `mat` module to perform matrix operations in C. It was time to measure how much these changes affected Hera's performances. In order to measure this change of performance, the sensor fusion GRiSP application using Hera developed in [5] was used to compare the performances of the different versions of Hera.

6.1 Attitude and Reference Heading System

In the sensor fusion application, multiple experiments were made. One of the latest was an Attitude and Reference Heading System or AHRS. An AHRS allows to track the orientation of an item by combining the information obtained from a 3-axis accelerometer, a 3- axis magnetometer and a 3-axis gyroscope. These sensors are all contained into the Pmod NAV. Orientation tracking using an AHRS is not a simple computational task and needs to be performed at high frequency to be as precise as possible.

6.1.1 Finding the orientation

Using a magnetometer to retrieve the magnetic north and a accelerometer to retrieve the direction of gravity when there is no linear acceleration, the orientation can be computed [5].

$$\begin{aligned} East &= \hat{Down} \times \hat{m} \\ North &= \hat{East} \times \hat{Down} \\ DCM &= (\hat{North}^T \times \hat{East}^T \times \hat{Down}^T) \end{aligned} \tag{6.1}$$

With m representing the magnetic north and $Down$ representing the gravity. The Direct Cosine Matrix or rotation matrix is found. It contains the information about

the orientation and can be used to build a Kalman filter. But first, it must be converted into a quaternion representation (6.2) which is a non ambiguous way of representing orientation and rotation of objects in three dimensional space. In the following equations, D represent the rotation matrix.

$$q_1^2 = \frac{1}{4}(1 + D_{11} + D_{22} + D_{33}) \tag{6.2}$$

$$q_{am} = \frac{1}{4q_1} \begin{bmatrix} 4q_1^2 \\ D_{32} - D_{23} \\ D_{13} - D_{31} \\ D_{21} - D_{12} \end{bmatrix} \tag{6.3}$$

6.1.2 Building the Kalman filter

After converting the rotation matrix into a quaternion representation, the parameters of the Kalman filter (6.5) can be defined as such:

$$\Omega(\omega) = \begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \\ -\omega_x & 0 & -\omega_z & \omega_y \\ -\omega_y & \omega_z & 0 & -\omega_x \\ -\omega_z & -\omega_y & \omega_x & 0 \end{bmatrix} \tag{6.4}$$

$$\begin{aligned} F &= I_{4\times4} + \frac{\Delta t}{2}\Omega(\omega) \\ Q &= \sigma_Q^2 I_{4\times4} \\ H &= I_{4\times4} \\ z &= q_{am} \\ R &= \sigma_R^2 I_{4\times4} \end{aligned} \tag{6.5}$$

Where:

ω = the measurement vector from the gyroscope
σ_Q^2 = the process noise variance
σ_R^2 = the observation noise variance

6.1.3 Parameter fine-tuning

In the experiment, the values for the variances σ_Q^2 and σ_R^2 were found by trial and error. Fine-tuning these variables even further did not improve the precision of the Kalman filter significantly. Thus, the values $\sigma_Q^2 = 10^{-3}$ and $\sigma_R^2 = 10^{-2}$ were kept.

Usually, F and Q the parameters of the prediction phase are independent of sensor measurements allowing the prediction to be performed multiple times while there is no measurement. Here, the authors decided to introduce the gyroscope measurement in the prediction parameter F for two reasons. Firstly, they were limited by the computational need of the Kalman filter and were never able to make predictions while waiting for a measurement. Secondly, introducing the gyroscope in the prediction allows the system to be much more precise. The experiment will be kept as it is to have a functional experiment on all devices and versions of Hera. In addition, the speed obtained by measurement will represent a full Kalman loop which will be more representative than an unknown number of predictions before each correction phase.

6.2 Experiment setup

One GRiSP board was used for the experiment along a Pmod NAV sensor, a computer was used to receive and store the data from the `hera_measure` processes. Regarding the GRiSP board's movement, multiple routines were executed such as 360° rotations around each of the 3 axis, waves and stationary position. Since, the routines did not have an impact on the computational performances measured, the measurements found in this sections are all made on a motionless GRiSP board.

The experiment is built around two `hera_measure` processes. The first one is responsible of retrieving the measures from the Pmod NAV sensor. The sensor data is stored and retrieved by the second process which acts as the AHRS described in 6.1. It retrieves new sensor data, transforms it into a quaternion representation and then computes an estimate of the state with a Kalman filter. It then repeats this process every time new sensor data is fetched. Because the Kalman filter prediction and correction both need a measurement to be computed, the AHRS process cannot perform Kalman predictions while waiting for a new measure. This means that the Kalman filter speed is limited by the sensor measurement speed in this case. Trying to synchronise these the two processes can thus become beneficial (see 6.5).

6.3 Computational time comparison

In order to obtain an orientation estimation that is as precise as possible, speeding up the computation of the AHRS process is the key. Indeed, performing more iterations of the Kalman filter per second means getting more estimates of the orientation resulting in a more precise estimation. To keep track of the evolution

of the performances of Hera, the computation time of the AHRS process has been measured in three different contexts:

- GRiSP-Base: using the old version of Hera running on a GRiSP-Base board
- GRiSP2: using the version of Hera adapted for GRiSP2 and running on a GRiSP2 board.
- GRiSP2-Numerl: using the latest version of Hera including the Numerl library and running on a GRiSP2 board.

The computations of one iteration of the AHRS process can be divided into three parts: the rotation matrix conversion into a quaternion representation, the prediction phase and the correction phase of the Kalman filter. In order to observe as precisely as possible the improvements in computation speed in the AHRS process, the speed of each of these parts have been measured. The computation time of each part has been measured for 10000 iterations and the mean was taken to reduce variance. The exact values for each part can be found in table 6.1.

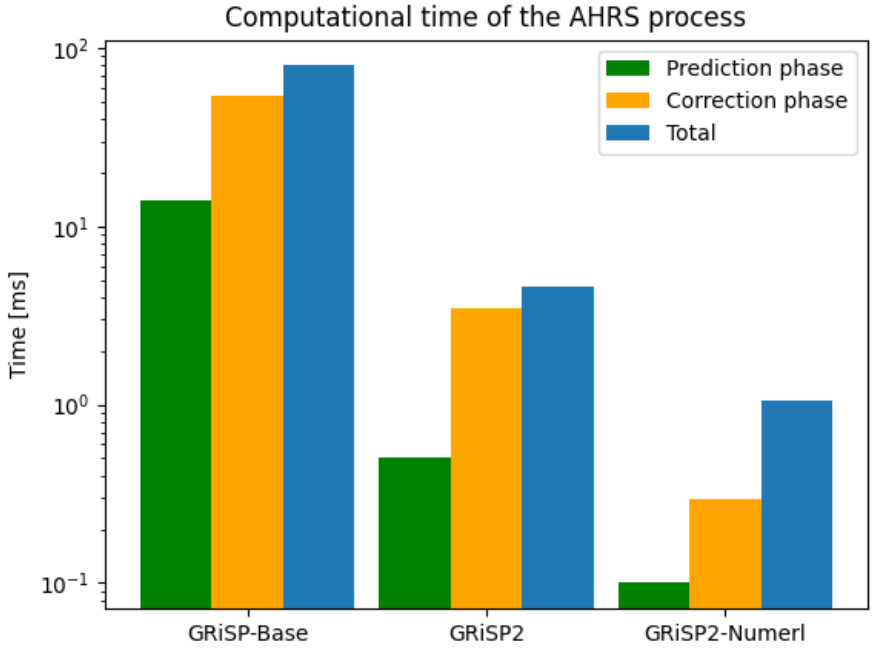

Figure 6.1: Mean computational time of the different parts of the AHRS process

	GRiSP-Base	GRiSP2	GRiSP2-Numerl
Prediction phase	13.9	0.5	0.1
Correction phase	53.7	3.5	0.295
Quaternion representation	12.3	0.6	0.66
Total	79.9	4.6	1.06

Table 6.1: Mean computational time [ms] of the different parts of the AHRS process

Globally, the GRiSP2 version computes 17 times faster than the GRiSP-Base version with no change in the Hera software. This improvement is massive and makes Hera usable for faster applications on it's own. An improvement of factor 4 is then observed between the GRiSP2-Numerl version and the GRiSP2 version. The computational time of the GRiSP2-Numerl version is about 75 faster than the starting version, with the major improvement being observed on the correction phase of the Kalman filter.

Initially, the correction phase took the most time to be computed. This is due to the large number of complex operations that is performed in that phase compared to the other. Eight matrix multiplications, one matrix inversion and a few matrix additions. The hardware upgrade sped up the phase by 15 times, a little less than the global improvement for this transition. On the other hand, the Numerl integration brought an improvement of almost factor 12 to the correction phase computational time. Thus, in line with the observation made in 2.4.1 and 5.3, using Numerl for the matrix computations greatly improved the computation time of complex operations such as matrix inversion and matrix multiplication.

But replacing the Erlang matrix computations by C matrix computations didn't help for every part of the computations. For the quaternion representation conversion, no time is gained between the GRiSP2 and GRiSP2-Numerl versions. The operations performed in this part are matrices multiplied by constants and matrices declarations which are simple operations. Some of these operations are still performed in Erlang which is why no improvement is observed.

6.4 Iteration frequency comparison

The improvements observed for the computational part of the AHRS process are great but do not reflect the improvements of the process as a whole. The AHRS process is composed of multiple parts, one of them being the computational part, which also take time during one iteration of the process. The number of complete iterations per second of the AHRS process represents the number of orientation

estimation that will be available to use for other processes. This metric will be called iteration frequency. Ultimately, iteration frequency is the most important metric regarding this experiment as it measures how precise applications using the AHRS process will able to be.

6.4.1 Decomposing the AHRS process

The AHRS process has to read the sensor data from a map data structure, compute the estimate of the orientation and then communicate the results of its computations like every `hera_measure` process. The figure 6.2 shows how the time for one iteration of the AHRS process is divided between these parts. The **Other** bar represent the time communicating the results and waiting for the CPU to be free.

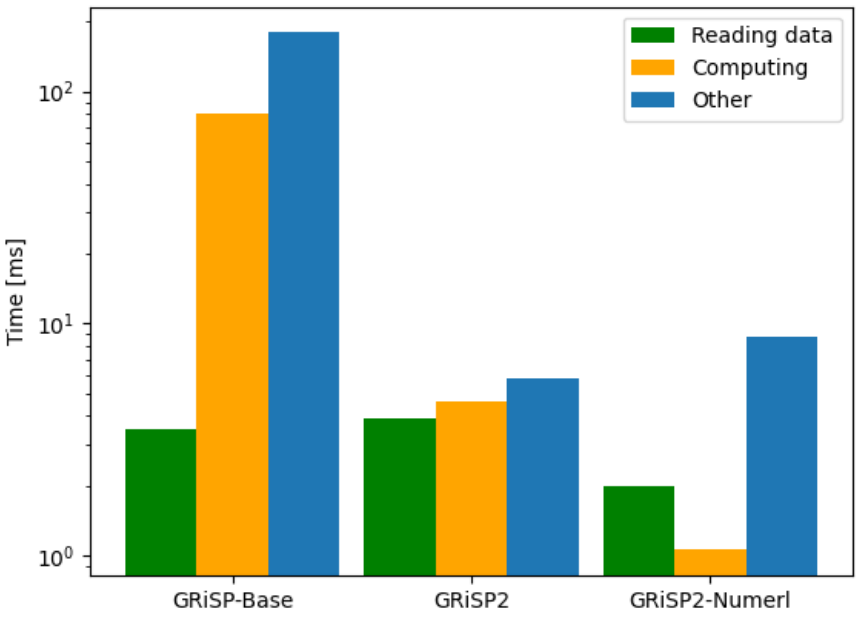

Figure 6.2: Mean execution time of the different parts of the AHRS process

As seen in the previous section, the **Computing** bar has diminished by a factor of 75 between the GRiSP-Base and the GRiSP2-Numerl versions. On top of that, the hardware changes allowed to spend less time communicating and waiting but a slight increase is observed when adding Numerl. A similar increase can be observed for the data read time between the GRiSP-Base and GRiSP2 versions. Even though

	GRiSP-Base	GRiSP2	GRiSP2-Numerl
Reading data	3.5	3.9	2
Computing	79.9	4.6	1.06
Other	179.2	5.8	8.8

Table 6.2: Mean execution time of the different parts of the AHRS process

these bars represent the mean value of 10000 iterations, the variance is quite high as it depends entirely on the schedulers decisions. Because there are other processes such as `hera_data` or `hera_com` that also require the CPU, the wait time can vary. Small variations in the CPU waiting time are more noticeable when dealing with microseconds. The figure 6.3 shows that the time spent reading the data from the Erlang map varies a lot, even GRiSP2-Numerl which has the fastest time for reading data has a lot of slower outliers. On top of that, these results were not reproducible as reading time varies so much. The reading time mean is usually around 3.5 seconds for both GRiSP2 and GRiSP2-Numerl processes.

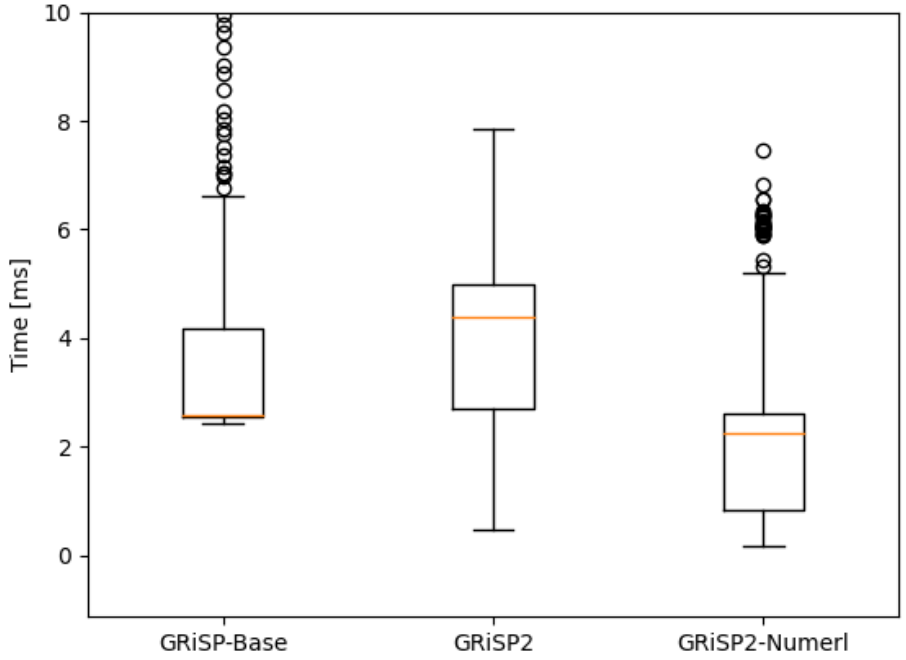

Figure 6.3: Box plot of the time needed to read data at each iteration

6.4.2 AHRS and NAV iteration frequencies

In [5], the authors achieved an iteration frequency of approximately 3.75 Hz which was not enough for fast paced applications. The hardware and software improvements made to Hera multiplied that number by over 22 to achieve an iteration frequency of 84.6 Hz. The figure 6.4 shows how the iteration frequency of the AHRS and NAV processes has evolved through the different versions of Hera. In this experiment, the AHRS process needs new sensor data to compute a new estimate of the orientation. Because of that, observing the iteration frequency of NAV, the provider of the sensor data, is interesting. Despite not being changed between the GRiSP2 and GRiSP2-Numerl version, the NAV process sees its iteration frequency increase. Because the orientation is computed faster, CPU resources are freed allowing the NAV process to retrieve information from the sensors faster. This behaviour can be observed by the difference in iteration frequency between the two processes being roughly equal for the GRiSP2 and GRiSP2-Numerl versions. Improving any part of Hera may improve the speed of every part of Hera as it is very likely to be running on a single core processor.

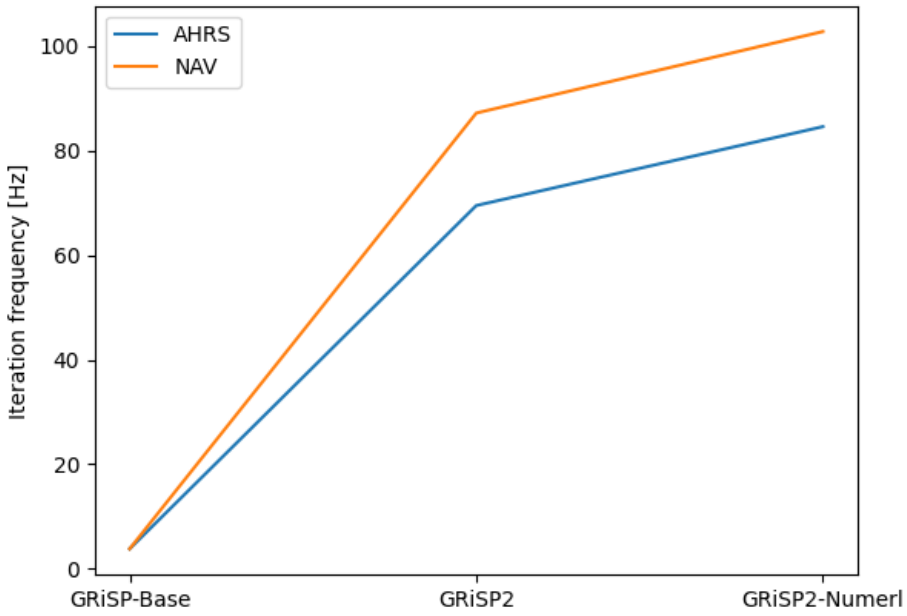

Figure 6.4: Comparison of the iteration frequency of the AHRS and NAV processes

	GRiSP-Base	GRiSP2	GRiSP2-Numerl
AHRS process	3.8	69.5	84.6
NAV process	3.8	87.2	102.8

Table 6.3: Iteration frequency [Hz] of the AHRS and NAV processes

6.5 Timeout fine-tuning

The `hera_measure` processes have an integrated timeout parameter making the process wait the duration of that timeout after having completed an iteration. This timeout can be used in order to reduce the amount of CPU resources required by a process. In the measurements of the previous sections, a timeout of 0 was given for both AHRS and NAV processes. Because the iteration frequency of the NAV process is higher than its counterpart, the AHRS process sometimes receives multiples measures. These measures can be used at once without a problem to compute an estimate. However, by adding a timeout to the NAV process, it is possible to reduce the number of measures taken to free some CPU resources. These resources can then be used to compute more orientation estimates.

By trial and error, the timeout value of 1 millisecond for the NAV process gave the best iteration frequency for the AHRS process while maintaining the NAV iteration frequency above it. The figure 6.5 shows the results obtain with the newly found timeout.

	GRiSP-Base	GRiSP2	GRiSP2-Numerl
AHRS process	3.8	72	89
NAV process	3.8	83.2	93.1

Table 6.4:

Fine-tuning the timeout allows to distribute CPU resources differently. The GRiSP2-Numerl version is slighly faster with the added timeout on NAV, going from 84.6 Hz to 89 Hz. In terms of duration of an iteration this represent an improvement of 0.6 millisecond. This kind of fine-tuning is specific and depends on the application. In our case, getting more iterations of the NAV process wasn't particularly valuable in comparison to getting more iterations of its counterpart.

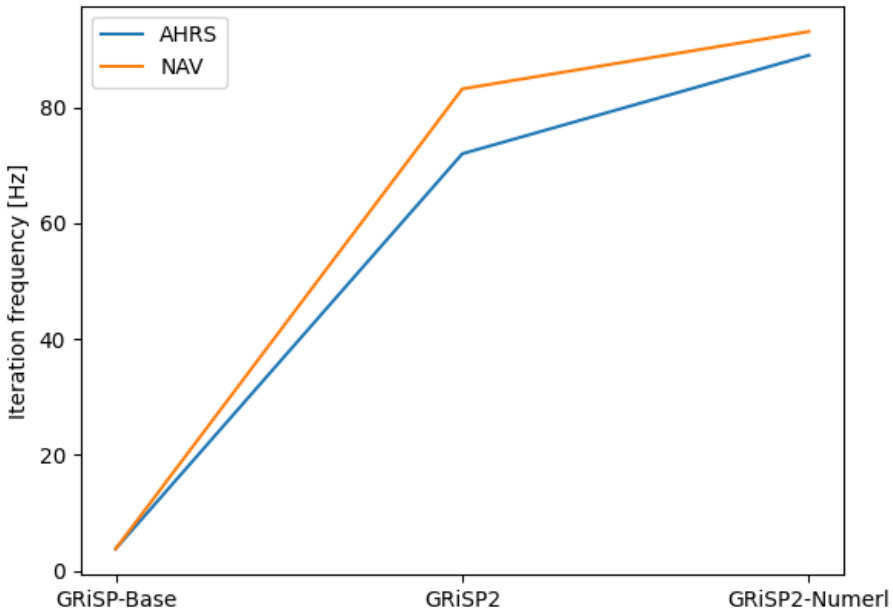

Figure 6.5: Comparison of the iteration frequency of the AHRS and NAV processes after fine-tuning

Chapter 7

Conclusion

At the start of this master thesis, Hera had already proven to be capable of performing sensor fusion entirely at the edge in a low-cost environment. It provided the user with a fault-tolerant sensor fusion framework that supports distributed networks. However, the number of applications that could be created using Hera was limited due to its lack of speed.

Throughout my master thesis, I was able to demonstrate that it is possible to overcome that weakness to perform low-cost sensor fusion at relatively high speed. In order to achieve this, Hera was ported to GRiSP2 and its matrix library was replaced by Numerl. This new version of Hera is called Hera v2.0.

7.1 Results

The performances measurements of the different versions of Hera were made on the Attitude and Heading Reference System experiment. The iteration frequency of the process estimating the orientation of an object and the computation time were used as a comparison point at each step.

Firstly, Hera was ported from GRiSP-Base boards to GRiSP2 boards. GRiSP2 boards being a significant hardware upgrade over their predecessor, the iteration frequency of the AHRS process went from 3.8 Hz to 69.5 Hz. Porting Hera to GRiSP2 reduced the execution time of every process bringing an overall upgrade to Hera's speed.

Secondly, the Erlang matrix library of Hera was replaced by Numerl, a matrix library using NIFs to take advantage of the fast computations of C. The use of Numerl reduced the time spent computing an estimate of the orientation from

4.6 ms to 1.06 ms with the improvements being focused on the Kalman filter computations. The prediction phase and correction phase of the filter got faster by a factor of 5 and 11.8 respectively. This significant improvement in computation time has led to an increase of the iteration frequency, reaching a value of 84.6 Hz.

Finally, by refining the timeout parameter of the AHRS and NAV processes, an iteration frequency of 89 Hz was achieved. This result is representative of the performances achieved by Hera v2.0 making it roughly 23 times quicker than Hera v1.0 thanks to the hardware and software upgrades. These improvements allow Hera to be used more reliably and increase its number of possible applications.

7.2 Future work

7.2.1 Performance improvements

The actual state of Hera allows it to be used in applications that require high speed sensor fusion at the edge with low-cost equipment. However, Hera is not perfect, `hera_measure` processes have too much input/output overhead, additional processes could be added to push even further the Kalman filter capabilities and matrix computations could be optimised even more with the introduction of BLAS. Such improvements could lead Hera to be even more precise and increase the range of potential applications.

The `hera_data` module isn't fast enough to keep up with the speed of the computations. Indeed, The Erlang data structure `maps` used for data storage has inconsistent performances. The time spent retrieving stored data varies a lot which slows down each iteration of the Kalman filter significantly. Other data structures such as `ets` are more appropriate for big tables and have constant access time to the data. On top of that, the requests made to `hera_data` are treated sequentially. This behaviour is not ideal as one process storing a large amount of data could make other processes wanting to retrieve data wait. Storing and retrieving data could be separated in two processes with a data structure supporting atomic operations. Due to a lack of time, these problems could not be fully explored but optimising the data storage would globally smooth the input/output operations of Hera and improve its speed as data is often stored and queried.

On top of that, the Kalman filter could have a dedicated process, allowing it to perform predictions while waiting for new sensor data. When the data is received, the correction phase would be executed and the result sent back to another process. This would allow the Kalman filter to perform at an even higher speed but it would

come with higher CPU needs slowing down the other processes. This approach wasn't used in this project because it would not have been possible to test it given that the prediction needs sensor data.

During this year, a NIF based wrapper for BLAS has been developed by the Erlang Ecosystem Foundation and the author of Numerl. It allows to use BLAS calls in Erlang and has been integrated to the GRiSP toolchain right before the end of this project. Due to a lack of time, it could not be integrated into this project. However, replacing the Numerl library or some of the matrix operations by BLAS calls could improve even further the speed of Hera's Kalman filter. The `SYMM`, `GEMM` and `GEMV` BLAS routines can be used for the Kalman filter. BLAS routines can perform multiple operations at once in an optimised manner and are particularly effective when dealing with large matrices. The NIF library can be found at `https://github.com/erlef/blas`.

7.2.2 Applications

The performances improvements made to Hera open the door to new applications. Real-time fault-tolerant sensor fusion applications can be built easily on top of Hera without having to dive into Internet of Things networking. This year, a gesture recognition system using Hera was built [15] as a master thesis. It uses the GRiSP2 version of Hera without Numerl and achieves to learn multiple gestures and then recognise them. This system can then be used to build multiple applications such as controlling lights or doors in a room with sensors equipped on the wrist.

Person or object position tracking using sonars, as seen in [5], could be built with better precision using Hera v2.0. It can be used for multiple applications such as a fall detector for elderly people [7] or movement detection of a cook to assist him through an IoT network of kitchen devices.

At UCLouvain, multiple master thesis using Hera are planned for the next years. They are great examples of the range of applications that can be built with Hera. One of them is a human body movement tracker. The task of the application will be to collect data from sensors attached to a human body performing a moving activity such as sports. The data will then be analysed through machine learning algorithms to have a better understanding of the movements of a person. Another applications consists in self-balancing an IoT device with motors. The device's orientation and position will be computed similarly as seen in this master thesis and then used to balance the item. This project requires the system to be reactive as perturbations could be introduced making the high speed Kalman filter of Hera v2.0 very valuable.

Bibliography

[1] Karen Rose, Scott Eldridge, and Lyman Chapin. "The internet of things: An overview". In: *The internet society (ISOC)* 80 (2015), pp. 1–50.

[2] Wei Yu et al. "A survey on the edge computing for the Internet of Things". In: *IEEE access* 6 (2017), pp. 6900–6919.

[3] Man Lok Fung, Michael ZQ Chen, and Yong Hua Chen. "Sensor fusion: A review of methods and applications". In: *2017 29th Chinese Control And Decision Conference (CCDC)*. IEEE. 2017, pp. 3853–3860.

[4] Guillaume Neirinckx, Julien Bastin, and Peter Van Roy. "Sensor fusion at the extreme edge of an internet of things network". MA thesis. Université catholique de Louvain, 2020.

[5] Sébastien Kalbusch, Vincent Verpoten, and Peter Van Roy. "The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking". MA thesis. Université catholique de Louvain, 2021.

[6] Jono Vanhie-Van Gerwen et al. "Indoor drone positioning: Accuracy and cost trade-off for sensor fusion". In: *IEEE Transactions on Vehicular Technology* 71.1 (2021), pp. 961–974.

[7] Paola Pierleoni et al. "A wearable fall detector for elderly people based on AHRS and barometric sensor". In: *IEEE sensors journal* 16.17 (2016), pp. 6733–6744.

[8] Takashi Kusaka and Takayuki Tanaka. "Stateful Rotor for Continuity of Quaternion and Fast Sensor Fusion Algorithm Using 9-Axis Sensors". In: *Sensors* 22.20 (2022), p. 7989.

[9] Babak Shahian Jahromi, Theja Tulabandhula, and Sabri Cetin. "Real-time hybrid multi-sensor fusion framework for perception in autonomous vehicles". In: *Sensors* 19.20 (2019), p. 4357.

[10] Chunxu Li, Ashraf Fahmy, and Johann Sienz. "An augmented reality based human-robot interaction interface using Kalman filter sensor fusion". In: *Sensors* 19.20 (2019), p. 4586.

[11] Wael Farag. “Kalman-filter-based sensor fusion applied to road-objects detection and tracking for autonomous vehicles”. In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 235.7 (2021), pp. 1125–1138.

[12] Rudolph Emil Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

[13] Greg Welch, Gary Bishop, et al. “An introduction to the Kalman filter”. In: (1995).

[14] Tanguy Losseau and Peter Van Roy. “Numerl: Efficient Vector and Matrix Computation for Erlang”. MA thesis. Université catholique de Louvain, 2022.

[15] Sébastien Gios and Peter Van Roy. “Sensor fusion for gesture recognition on an Internet of Things network”. MA thesis. Université catholique de Louvain, 2023 (to appear).

Appendix A

Numerl files

A.1 numerl.c

```
#define STATIC_ERLANG_NIF 1

#include <erl_nif.h>
#include <stdio.h>
#include <string.h>
#include <math.h>


/*
-------------------------------------------------------------------------------/

                    ------------------------------------
                                                       /
                    /                INIT FC             /
                                                       /
                    ------------------------------------
                                                       /
-------------------------------------------------------------------------------/

*/


ERL_NIF_TERM atom_nok;
ERL_NIF_TERM numerl_atom_true;
ERL_NIF_TERM numerl_atom_false;
```

```
ERL_NIF_TERM atom_matrix;
const int CSTE_TIMESLICE = 5;

ErlNifResourceType *MULT_YIELDING_ARG = NULL;

int numerl_load(ErlNifEnv* env, void** priv_data,
   ERL_NIF_TERM load_info){
     atom_nok = enif_make_atom(env, "nok\0");
     numerl_atom_true = enif_make_atom(env, "true\0");
     numerl_atom_false = enif_make_atom(env, "false\0");
     atom_matrix = enif_make_atom(env, "matrix\0");
     return 0;
}

int upgrade(ErlNifEnv* caller_env, void** priv_data, void**
     old_priv_data, ERL_NIF_TERM load_info){
     return 0;
}

//Gives easier access to an ErlangBinary containing a
   matrix.
typedef struct{
     int n_rows;
     int n_cols;

     //Content of the matrix, in row major format.
     double* content;

     //Erlang binary containing the matrix.
     ErlNifBinary bin;
} Matrix;


//Access asked coordinates of matrix
double* matrix_at(int col, int row, Matrix m){
     return &m.content[row*m.n_cols + col];
}

//Allocates memory space of for matrix of dimensions n_rows
   , n_cols.
```

```
//The matrix_content can be modified, until a call to
   array_to_erl.
//Matrix content is stored in row major format.
Matrix matrix_alloc(int n_rows, int n_cols){
    ErlNifBinary bin;

    enif_alloc_binary(sizeof(double)*n_rows*n_cols, &bin);

    Matrix matrix;
    matrix.n_cols = n_cols;
    matrix.n_rows = n_rows;
    matrix.bin = bin;
    matrix.content = (double*) bin.data;

    return matrix;
}

//Creates a duplicate of a matrix.
//This duplicate can be modified until uploaded.
Matrix matrix_dup(Matrix m){
    Matrix d = matrix_alloc(m.n_rows, m.n_cols);
    memcpy(d.content, m.content, d.bin.size);
    return d;
}

//Free an allocated matrix that was not sent back to Erlang
   .
void matrix_free(Matrix m){
    enif_release_binary(&m.bin);
}

//Constructs a matrix erlang term.
//No modifications can be made afterwards to the matrix.
ERL_NIF_TERM matrix_to_erl(ErlNifEnv* env, Matrix m){
    ERL_NIF_TERM term = enif_make_binary(env, &m.bin);
    return enif_make_tuple4(env, atom_matrix, enif_make_int
       (env,m.n_rows), enif_make_int(env,m.n_cols), term);
}

int enif_is_matrix(ErlNifEnv* env, ERL_NIF_TERM term){
```

```
    int arity;
    const ERL_NIF_TERM* content;

    if(!enif_is_tuple(env, term))
        return 0;

    enif_get_tuple(env, term, &arity, &content);
    if(arity != 4)
        return 0;

    if(content[0] != atom_matrix
        || !enif_is_number(env, content[1])
        || !enif_is_number(env, content[2])
        || !enif_is_binary(env, content[3]))

            return 0;
    return 1;
}

//Reads an erlang term as a matrix.
//As such, no modifications can be made to the red matrix.
//Returns true if it was possible to read a matrix, false
   otherwise
int enif_get_matrix(ErlNifEnv* env, ERL_NIF_TERM term,
   Matrix *dest){

    int arity;
    const ERL_NIF_TERM* content;

    if(!enif_is_tuple(env, term))
        return 0;

    enif_get_tuple(env, term, &arity, &content);
    if(arity != 4)
        return 0;

    if(content[0] != atom_matrix
        || !enif_get_int(env, content[1], &dest->n_rows)
        || !enif_get_int(env, content[2], &dest->n_cols)
        || !enif_inspect_binary(env, content[3], &dest->bin
```

```
        ))
    {
        return 0;
    }

    dest->content = (double*) (dest->bin.data);
    return 1;
}

//Used to translate at once a number of ERL_NIF_TERM.
//Data types are inferred via content of format string:
//  n: number (int or double) translated to double.
//  m: matrix
//  i: int
int enif_get(ErlNifEnv* env, const ERL_NIF_TERM* erl_terms,
    const char* format, ...){
    va_list valist;
    va_start(valist, format);
    int valid = 1;

    while(valid && *format != '\0'){
        switch(*format++){
            case 'n':
                //Read a number as a double.
                ;
                double *d = va_arg(valist, double*);
                int i;
                if(!enif_get_double(env, *erl_terms, d))
                    if(enif_get_int(env, *erl_terms, &i))
                        *d = (double) i;
                    else valid = 0;
                break;

            case 'm':
                //Reads a matrix.
                valid = enif_get_matrix(env, *erl_terms,
                   va_arg(valist, Matrix*));
                break;

            case 'i':
```

```
                //Reads an int.
                valid = enif_get_int(env, *erl_terms,
                    va_arg(valist, int*));
                break;

            default:
                //Unknown type... give an error.
                valid = 0;
                break;
        }
        erl_terms ++;
    }

    va_end(valist);
    return valid;
}


// ------------------------------------------------------------------------

// ------------------------------------------
                                          /
//                          /                    NIFS
                  /                                    /
// ------------------------------------------
                                          /
// ------------------------------------------------------------------------



//@arg 0: List of Lists of numbers.
//@return: Matrix of dimension
ERL_NIF_TERM nif_matrix(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    unsigned n_rows, line_length, dest = 0, n_cols = -1;
    ERL_NIF_TERM list = argv[0], row, elem;
```

```
    Matrix m;

    //Reading incoming matrix.
    if(!enif_get_list_length(env, list, &n_rows) && n_rows
       > 0)
      return enif_make_badarg(env);

    for(int i = 0; enif_get_list_cell(env, list, &row, &
       list); i++){
      if(!enif_get_list_length(env, row, &line_length))
          return enif_make_badarg(env);

      if(n_cols == -1){
          //Allocate binary, make matrix accessor.
          n_cols = line_length;
          m = matrix_alloc(n_rows,n_cols);
      }

      if(n_cols != line_length)
          return enif_make_badarg(env);

      for(int j = 0; enif_get_list_cell(env, row, &elem,
         &row); j++){
          if(!enif_get_double(env, elem, &m.content[dest
             ])){
              int i;
              if(enif_get_int(env, elem, &i)){
                  m.content[dest] = (double) i;
              }
              else{
                  return enif_make_badarg(env);
              }
          }
          dest++;
      }
    }
    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return matrix_to_erl(env, m);
}
```

```
#define PRECISION 10

//Used for debug purpose.
void debug_write(char info[]){
    FILE* fp = fopen("debug.txt", "a");
    fprintf(fp, info);
    fprintf(fp, "\n");
    fclose(fp);
}

void debug_write_matrix(Matrix m){
    char *content = enif_alloc(sizeof(char)*((2*m.n_cols-1)
       *m.n_rows*PRECISION + m.n_rows*2 + 3));
    content[0] = '[';
    content[1] = '\0';
    char converted[PRECISION];

    for(int i=0; i<m.n_rows; i++){
        strcat(content, "[");
        for(int j = 0; j<m.n_cols; j++){
            snprintf(converted, PRECISION-1, "%.5lf", m.
               content[i*m.n_cols+j]);
            strcat(content, converted);
            if(j != m.n_cols-1)
                strcat(content, " ");
        }
        strcat(content, "]");
    }
    strcat(content, "]");
    debug_write(content);
    enif_free(content);

}


//@arg 0: matrix.
//@arg 1: int, coord m: row
//@arg 2: int, coord n: col
//@return: double, at cord Matrix(m,n).
ERL_NIF_TERM nif_get(ErlNifEnv * env, int argc, const
```

```
    ERL_NIF_TERM argv[]){
      int m,n;
      Matrix matrix;

      if(!enif_get(env, argv, "iim", &m, &n, &matrix))
          return enif_make_badarg(env);
      m--, n--;

      if(m < 0 || m >= matrix.n_rows || n < 0 || n >= matrix.
         n_cols)
          return enif_make_badarg(env);

      int index = m*matrix.n_cols+n;
      return enif_make_double(env, matrix.content[index]);
}

ERL_NIF_TERM nif_at(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
      int n;
      Matrix matrix;

      if(!enif_get(env, argv, "mi", &matrix, &n))
          return enif_make_badarg(env);
      n--;

      if( n < 0 || n >= matrix.n_cols * matrix.n_rows)
          return enif_make_badarg(env);

      return enif_make_double(env, matrix.content[n]);
}

//Matrix to flattened list of ints
ERL_NIF_TERM nif_mtfli(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
      Matrix M;
      if(!enif_get(env, argv, "m", &M)){
          return enif_make_badarg(env);
      }

      int n_elems = M.n_cols * M.n_rows;
```

```
    ERL_NIF_TERM *arr = enif_alloc(sizeof(ERL_NIF_TERM)*
        n_elems);
    for(int i = 0; i<n_elems; i++){
        arr[i] = enif_make_int(env, (int)M.content[i]);
    }

    ERL_NIF_TERM result = enif_make_list_from_array(env,
        arr, n_elems);
    enif_consume_timeslice(env, CSTE_TIMESLICE);
    enif_free(arr);
    return result;
}

//Matrix to flattened list of ints
ERL_NIF_TERM nif_mtfl(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
    Matrix M;
    if(!enif_get(env, argv, "m", &M)){
        return enif_make_badarg(env);
    }

    int n_elems = M.n_cols * M.n_rows;
    ERL_NIF_TERM *arr = enif_alloc(sizeof(ERL_NIF_TERM)*
        n_elems);
    for(int i = 0; i<n_elems; i++){
        arr[i] = enif_make_double(env, M.content[i]);
    }

    ERL_NIF_TERM result = enif_make_list_from_array(env,
        arr, n_elems);
    enif_free(arr);
    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return result;
}

//Equal all doubles
//Compares whether all doubles are approximately the same.
int equal_ad(double* a, double* b, int size){
    for(int i = 0; i<size; i++){
        if(fabs(a[i] - b[i])> 1e-6)
```

```
            return 0;
    }
    return 1;
}

//@arg 0: Array.
//@arg 1: Array.
//@return: true if arrays share content, false if they have
    different content // size..
ERL_NIF_TERM nif_equals(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){
    Matrix a,b;

    if(!enif_get(env, argv, "mm", &a, &b))
        return numerl_atom_false;

    //Compare number of columns and rows
    if((a.n_cols != b.n_cols || a.n_rows != b.n_rows))
        return numerl_atom_false;

    //Compare content of arrays
    if(!equal_ad(a.content, b.content, a.n_cols*a.n_rows))
        return numerl_atom_false;

    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return numerl_atom_true;
}


//@arg 0: int.
//@arg 1: Array.
//@return: returns an array, containing requested row.
ERL_NIF_TERM nif_row(ErlNifEnv * env, int argc, const
   ERL_NIF_TERM argv[]){
    int row_req;
    Matrix matrix;
    if(!enif_get(env, argv, "im", &row_req, &matrix))
        return enif_make_badarg(env);

    row_req--;
```

```
    if(row_req<0 || row_req >= matrix.n_rows)
        return enif_make_badarg(env);

    Matrix row = matrix_alloc(1, matrix.n_cols);
    memcpy(row.content, matrix.content + (row_req * matrix.
       n_cols), matrix.n_cols*sizeof(double));

    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return matrix_to_erl(env, row);
}


//@arg 0: int.
//@arg 1: Array.
//@return: returns an array, containing requested col.
ERL_NIF_TERM nif_col(ErlNifEnv * env, int argc, const
   ERL_NIF_TERM argv[]){
    int col_req;
    Matrix matrix;
    if(!enif_get(env, argv, "im", &col_req, &matrix))
        return enif_make_badarg(env);

    col_req--;
    if(col_req<0 || col_req >= matrix.n_cols)
        return enif_make_badarg(env);


    Matrix col = matrix_alloc(matrix.n_rows, 1);

    for(int i = 0; i < matrix.n_rows; i++){
        col.content[i] = matrix.content[i * matrix.n_rows +
            col_req];
    }

    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return matrix_to_erl(env, col);
}


//@arg 0: int.
```

```
//@arg 1: int.
//@return: empty Matrix of requested dimension..
ERL_NIF_TERM nif_zeros(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){
    int m,n;
    if(!enif_get(env, argv, "ii", &m, &n))
        return enif_make_badarg(env);

    Matrix a = matrix_alloc(m,n);
    memset(a.content, 0, sizeof(double)*m*n);
    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return matrix_to_erl(env, a);
}

//@arg 0: int.
//@arg 1: int.
//@return: empty matrix of dimension [arg 0, arg 1]..
ERL_NIF_TERM nif_eye(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){
    int m;
    if(!enif_get_int(env, argv[0], &m))
        return enif_make_badarg(env);

    if(m <= 0)
        return enif_make_badarg(env);

    Matrix a = matrix_alloc(m,m);
    memset(a.content, 0, sizeof(double)*m*m);
    for(int i = 0; i<m; i++){
        a.content[i*m+i] = 1.0;
    }
    enif_consume_timeslice(env, CSTE_TIMESLICE);
    return matrix_to_erl(env, a);
}

//Either element-wise multiplication, or multiplication of
   a matrix by a number.
ERL_NIF_TERM nif_mult(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){
```

```
    Matrix a,b;
    double c;

    if(enif_get(env, argv, "mn", &a, &c)){
        Matrix d = matrix_alloc(a.n_rows, a.n_cols);

        for(int i = 0; i<a.n_cols*a.n_rows; i++){
            d.content[i] = a.content[i] * c;
        }

        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }
    else
    if(enif_get(env, argv, "mm", &a, &b)
            && a.n_rows*a.n_cols == b.n_rows*b.n_cols){

        Matrix d = matrix_alloc(a.n_rows, a.n_cols);

        for(int i = 0; i < a.n_rows*a.n_cols; i++){
            d.content[i] = a.content[i] * b.content[i];
        }

        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }

    return enif_make_badarg(env);
}


//Either element-wise addition, or addition of a matrix by
   a number.
ERL_NIF_TERM nif_add(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){

    Matrix a,b;
    double c;
```

```
    if(enif_get(env, argv, "mn", &a, &c)){
        Matrix d = matrix_alloc(a.n_rows, a.n_cols);

        for(int i = 0; i<a.n_cols*a.n_rows; i++){
            d.content[i] = a.content[i] + c;
        }

        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }
    else
    if(enif_get(env, argv, "mm", &a, &b)
            && a.n_rows*a.n_cols == b.n_rows*b.n_cols){

        Matrix d = matrix_alloc(a.n_rows, a.n_cols);

        for(int i = 0; i < a.n_rows*a.n_cols; i++){
            d.content[i] = a.content[i] + b.content[i];
        }
        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }

    return enif_make_badarg(env);
}


//Either element-wise subtraction, or subtraction of a
   matrix by a number.
ERL_NIF_TERM nif_sub(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){

    Matrix a,b;
    double c;

    if(enif_get(env, argv, "mn", &a, &c)){
        Matrix d = matrix_alloc(a.n_rows, a.n_cols);
```

```
        for(int i = 0; i<a.n_cols*a.n_rows; i++){
            d.content[i] = a.content[i] - c;
        }
        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }
    else
    if(enif_get(env, argv, "mm", &a, &b)
            && a.n_rows*a.n_cols == b.n_rows*b.n_cols){

        Matrix d = matrix_alloc(a.n_rows, a.n_cols);

        for(int i = 0; i < a.n_rows*a.n_cols; i++){
            d.content[i] = a.content[i] - b.content[i];
        }
        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }

    return enif_make_badarg(env);
}


//Either element-wise division, or division of a matrix by
   a number.
ERL_NIF_TERM nif_divide(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){

    Matrix a,b;
    double c;

    if(enif_get(env, argv, "mn", &a, &c)){

        if(c!=0.0){
            Matrix d = matrix_alloc(a.n_rows, a.n_cols);

            for(int i = 0; i<a.n_cols*a.n_rows; i++){
```

```
                d.content[i] = a.content[i] / c;
            }
            enif_consume_timeslice(env, a.n_cols*a.n_rows /
                100);
            return matrix_to_erl(env, d);
        }
    }
    else if(enif_get(env, argv, "mm", &a, &b)
            && a.n_rows*a.n_cols == b.n_rows*b.n_cols){

        Matrix d = matrix_alloc(a.n_rows, a.n_cols);

        for(int i = 0; i < a.n_rows*a.n_cols; i++){
            if(b.content[i] == 0.0){
                return enif_make_badarg(env);
            }
            else{
                d.content[i] = a.content[i] / b.content[i];
            }
        }
        enif_consume_timeslice(env, a.n_cols*a.n_rows /
           100);
        return matrix_to_erl(env, d);
    }

    return enif_make_badarg(env);
}


//Transpose of a matrix.
Matrix tr(Matrix a){
    Matrix result = matrix_alloc(a.n_cols ,a.n_rows);

    for(int j = 0; j < a.n_rows; j++){
        for(int i = 0; i < a.n_cols; i++){
            result.content[i*result.n_cols+j] = a.content[j
               *a.n_cols+i];
        }
    }
    return result;
```

```
}
ERL_NIF_TERM nif_transpose(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){

    Matrix a;
    if(!enif_get_matrix(env, argv[0], &a))
        return enif_make_badarg(env);
    enif_consume_timeslice(env, a.n_cols*a.n_rows / 100);
    return matrix_to_erl(env, tr(a));
}


//arg0: Matrix.
ERL_NIF_TERM nif_inv(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){

    Matrix a;
    if(!enif_get_matrix(env, argv[0], &a))
        return enif_make_badarg(env);

    if(a.n_cols != a.n_rows){
        return atom_nok;
    }
    int n_cols = 2*a.n_cols;

    double* gj = (double*) enif_alloc(n_cols*a.n_rows*
       sizeof(double));
    for(int i=0; i<a.n_rows; i++){
        memcpy(gj+i*n_cols, a.content+i*a.n_cols, sizeof(
           double)*a.n_cols);
        memset(gj+i*n_cols + a.n_cols, 0, sizeof(double)*a.
           n_cols);
        gj[i*n_cols + a.n_cols + i] = 1.0;
    }

    //Elimination de Gauss Jordan:
    //https://fr.wikipedia.org/wiki/%C3%89
       limination_de_Gauss-Jordan

    //Row of last found pivot
```

```
int r = -1;
//j for all indexes of column
for(int j=0; j<a.n_cols; j++){

    //Find the row of the maximum in column j
    int pivot_row = -1;
    for(int cur_row=r; cur_row<a.n_rows; cur_row++){
        if(pivot_row<0 || fabs(gj[cur_row*n_cols + j])
           > fabs(gj[pivot_row*n_cols+j])){
            pivot_row = cur_row;
        }
    }
    double pivot_value = gj[pivot_row*n_cols+j];

    if(pivot_value != 0){
        r++;
        for(int cur_col=0; cur_col<n_cols; cur_col++){
            gj[cur_col+pivot_row*n_cols] /= pivot_value
                ;
        }
        gj[pivot_row*n_cols + j] = 1.0; //make up for
            rounding errors

        //Do we need to swap?
        if(pivot_row != r){
            for(int i = 0; i<n_cols; i++){
                double cpy = gj[pivot_row*n_cols+i];
                gj[pivot_row*n_cols+i]= gj[r*n_cols+i];
                gj[r*n_cols+i] = cpy;
            }
        }

        //We can simplify all the rows
        for(int i=0; i<a.n_rows; i++){
            if(i!=r){
                double factor = gj[i*n_cols+j];
                for(int col=0; col<n_cols; col++){
                    gj[col+i*n_cols] -= gj[col+r*n_cols
                        ]*factor;
                }
```

```
                    gj[i*n_cols+j] = 0.0;     //make up for
                        rounding errors
                }
            }
        }

    }

    Matrix inv = matrix_alloc(a.n_rows, a.n_cols);
    for(int l=0; l<inv.n_rows; l++){
        int line_start = l*n_cols + a.n_cols;
        memcpy(inv.content + inv.n_cols*l, gj + line_start,
            sizeof(double)*inv.n_cols);
    }

    enif_free(gj);
    enif_consume_timeslice(env, a.n_cols*a.n_rows / 100);
    return matrix_to_erl(env, inv);
}


//Arguments: double alpha, matrix A, matrix B, double beta,
    matrix C
ERL_NIF_TERM nif_dgemm(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){
    Matrix A, B;

    if(!enif_get(env, argv, "mm", &A, &B)
            || A.n_cols != B.n_rows){

        return enif_make_badarg(env);
    }

    //debug_write_matrix(A);
    //debug_write_matrix(B);

    int n_rows = A.n_rows;
    int n_cols = B.n_cols;

    if(A.n_cols != B.n_rows)
```

```
        return atom_nok;

    Matrix result = matrix_alloc(n_rows, n_cols);
    memset(result.content, 0.0, n_rows*n_cols * sizeof(
       double));

    //Will this create a mem leak?
    Matrix b_tr = tr(B);

    for(int i = 0; i < n_rows; i++){
        for(int j = 0; j < n_cols; j++){
            for(int k = 0; k<A.n_cols; k++){
                result.content[j+i*result.n_cols] += A.
                   content[k+i*A.n_cols] * b_tr.content[k+j*
                   b_tr.n_cols];
            }
        }
    }

    enif_release_binary(&b_tr.bin); // might need to
       replace by enif_release_binary(&m.bin);

    enif_consume_timeslice(env, A.n_cols*A.n_rows*B.n_rows
       / 100);
    return matrix_to_erl(env, result);
}

ErlNifFunc nif_funcs[] = {
    {"matrix", 1, nif_matrix},
    {"get", 3, nif_get},
    {"at", 2, nif_at},
    {"mtfli", 1, nif_mtfli},
    {"mtfl", 1, nif_mtfl},
    {"equals", 2, nif_equals},
    {"row", 2, nif_row},
    {"col", 2, nif_col},
    {"zeros", 2, nif_zeros},
    {"eye", 1, nif_eye},
    {"mult", 2, nif_mult},
    {"add",  2, nif_add},
```

```
    {"sub",  2, nif_sub},
    {"divide", 2, nif_divide},
    {"transpose", 1, nif_transpose},
    {"inv", 1, nif_inv},

    {"dot", 2, nif_dgemm}
};

ERL_NIF_INIT(numerl, nif_funcs, numerl_load, NULL, upgrade,
    NULL)
```

A.2 numerl.erl

```
-module(numerl).
-on_load(init/0).
-export([ eval/1, eye/1, zeros/2, equals/2, add/2, sub/2,
    mult/2, divide/2, matrix/1, rnd_matrix/1, get/3, at/2,
    mtfli/1, mtfl/1, row/2, col/2, transpose/1, inv/1, nrm2
    /1, vec_dot/2, dot/2]).

%Matrices are represented as such:
%-record(matrix, {n_rows, n_cols, bin}).

init()->
    ok = erlang:load_nif(atom_to_list(?MODULE), 0).

%Creates a random matrix.
rnd_matrix(N)->
    L = [[rand:uniform(20) || _ <- lists:seq(1,N) ] || _ <-
        lists:seq(1,N)],
    matrix(L).

%Combine multiple functions.
eval([L,O,R|T])->
    F = fun numerl:O/2,
    eval([F(L,R) |T]);
eval([Res])->
    Res.

%%Creates a matrix.
%List: List of doubles, of length N.
%Return: a matrix of dimension MxN, containing the data.
matrix(_) ->
    nif_not_loaded.

%%Returns the Nth value contained within Matrix.
at(_Matrix,_Nth)->
    nif_not_loaded.

%%Returns the matrix as a flattened list of ints.
mtfli(_mtrix)->
```

```
    nif_not_loaded.

%%Returns the matrix as a flattened list of doubles.
mtfl(_mtrix)->
    nif_not_loaded.

%%Returns a value from a matrix.
get(_,_,_) ->
    nif_not_loaded.

%%Returns requested row.
row(_,_) ->
    nif_not_loaded.


%%Returns requested col.
col(_,_) ->
    nif_not_loaded.


%%Equality test between matrixes.
equals(_, _) ->
    nif_not_loaded.


%%Addition of matrix.
add(_, _) ->
    nif_not_loaded.


%%Subtraction of matrix.
sub(_, _) ->
    nif_not_loaded.


%% Matrix multiplication.
mult(A,B) when is_number(B) -> '*_num'(A,B);
mult(A,B) -> '*_matrix'(A,B).

'*_num'(_,_)->
```

```
    nif_not_loaded.

'*_matrix'(_, _)->
    nif_not_loaded.

%Matrix division by a number
divide(_,_)->
    nif_not_loaded.


%% build a null matrix of size NxM
zeros(_, _) ->
    nif_not_loaded.

%%Returns an Identity matrix NxN.
eye(_)->
    nif_not_loaded.

%Returns the transpose of the given square matrix.
transpose(_)->
    nif_not_loaded.

%Returns the inverse of asked square matrix.
inv(_)->
    nif_not_loaded.


%------CBLAS---------

%nrm2
%Calculates the squared root of the sum of the squared
   contents.
nrm2(_)->
    nif_not_loaded.

% : dot product of two vectors
% Arguments: vector x, vector y.
%   x and y are matrices
% Returns the dot product of all the coordinates of X,Y.
vec_dot(_, _)->
```

```
    nif_not_loaded.

% dgemm: A dot B
% Arguments: Matrix A, Matrix B.
%   alpha, beta: numbers (float or ints) used as doubles.
%   A,B,C: matrices.
% Returns the matrice resulting of the operations alpha * A
    * B + beta * C.
dot(_,_)->
    nif_not_loaded.
```

Appendix B

Hera matrix module

B.1 mat.erl

```
-module(mat).

-export([tr/1, inv/1]).
-export(['+'/2, '-'/2, '=='/2, '*'/2, '*  '/2]).
-export([row/2, col/2, get/3]).
-export([zeros/2, eye/1, diag/1]).
-export([eval/1]).
-export([matrix/1,to_array/1]).

-export_type([matrix/0]).

-type matrix() :: numerl:matrix().

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% API
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% create matrix
matrix(M) ->
    numerl:matrix(M).
```

```
%% returns an array
to_array(M) ->
    numerl:mtfl(M).

%% transpose matrix
-spec tr(M) -> Transposed when
    M :: matrix(),
    Transposed :: matrix().

tr(M) ->
    numerl:transpose(M).


%% matrix addition (M3 = M1 + M2)
-spec '+'(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'+'(M1, M2) ->
    numerl:add(M1,M2).


%% matrix subtraction (M3 = M1 - M2)
-spec '-'(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'-'(M1, M2) ->
    numerl:sub(M1,M2).


%% matrix multiplication (M3 = Op1 * M2)
-spec '*'(Op1, M2) -> M3 when
    Op1 :: number() | matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'*'(N, M) when is_number(N) ->
```

```
    numerl:mult(M,N);
'*'(M1, M2) ->
    numerl:dot(M1,M2).


%% transposed matrix multiplication (M3 = M1 * tr(M2))
-spec '*  '(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'*  '(M1, M2) ->
    numerl:dot(M1, tr(M2)).


%% return true if M1 equals M2
-spec '=='(M1, M2) -> boolean() when
    M1 :: matrix(),
    M2 :: matrix().

'=='(M1, M2) ->
    numerl:equals(M1,M2).


%% return the row I of M
-spec row(I, M) -> Row when
    I :: pos_integer(),
    M :: matrix(),
    Row :: matrix().

row(I, M) ->
    numerl:row(I,M).


%% return the column J of M
-spec col(J, M) -> Col when
    J :: pos_integer(),
    M :: matrix(),
    Col :: matrix().
```

```
col(J, M) ->
    numerl:col(J,M).


%% return the element at index (I,J) in M
-spec get(I, J, M) -> Elem when
    I :: pos_integer(),
    J :: pos_integer(),
    M :: matrix(),
    Elem :: number().

get(I, J, M) ->
    numerl:get(I,J,M).


%% return a null matrix of size NxM
-spec zeros(N, M) -> Zeros when
    N :: pos_integer(),
    M :: pos_integer(),
    Zeros :: matrix().

zeros(N, M) ->
    numerl:zeros(N,M).


%% return an identity matrix of size NxN
-spec eye(N) -> Identity when
    N :: pos_integer(),
    Identity :: matrix().

eye(N) ->
    numerl:eye(N).


%% return a square diagonal matrix with the elements of L
   on the main diagonal
-spec diag(L) -> Diag when
    L :: [number(), ...],
    Diag :: matrix().
```

```
diag(L) ->
    N = length(L),
    D = diag(L, [[0 || _ <- lists:seq(1, N)] || _ <- lists:
        seq(1, N)] , 0, []),
    numerl:matrix(D).


%% compute the inverse of a square matrix
-spec inv(M) -> Invert when
    M :: matrix(),
    Invert :: matrix().

inv(M) ->
    %N = numerl:matrix(M),
    numerl:inv(M).


%% evaluate a list of matrix operations
-spec eval(Expr) -> Result when
    Expr :: [T],
    T :: matrix() | '+' | '-' | '*' | '*   ',
    Result :: matrix().

eval([L|[O|[R|T]]]) ->
    F = fun mat:O/2,
    eval([F(L, R)|T]);
eval([Res]) ->
    Res.

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Internal functions
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% build a diagonal matrix from a zero matrix
diag([], [], _, Acc) ->
    lists:reverse(Acc);
```

```
diag([X|Xs], [Row|Rows], I, Acc) ->
    {L1, [_|T1]} = lists:split(I, Row),
    NewRow = lists:append([L1, [X], T1]),
    diag(Xs, Rows, I+1, [NewRow|Acc]).
```

B.2 oldmat.erl

```
-module(oldmat).

-export([tr/1, inv/1]).
-export(['+'/2, '-'/2, '=='/2, '*'/2, '*    '/2]).
-export([row/2, col/2, get/3]).
-export([zeros/2, eye/1, diag/1]).
-export([eval/1]).

-export_type([matrix/0]).

-type matrix() :: [[number(), ...], ...].

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% API
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% transpose matrix
-spec tr(M) -> Transposed when
      M :: matrix(),
      Transposed :: matrix().

tr(M) ->
      tr(M, []).


%% matrix addition (M3 = M1 + M2)
-spec '+'(M1, M2) -> M3 when
      M1 :: matrix(),
      M2 :: matrix(),
      M3:: matrix().

'+'(M1, M2) ->
      element_wise_op(fun erlang:'+'/2, M1, M2).
```

```
%% matrix subtraction (M3 = M1 - M2)
-spec '-'(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'-'(M1, M2) ->
    element_wise_op(fun erlang:'-'/2, M1, M2).


%% matrix multiplication (M3 = Op1 * M2)
-spec '*'(Op1, M2) -> M3 when
    Op1 :: number() | matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'*'(N, M) when is_number(N) ->
    [[N*X|| X <- Row] || Row <- M];
'*'(M1, M2) ->
    '*  '(M1, tr(M2)).


%% transposed matrix multiplication (M3 = M1 * tr(M2))
-spec '*  '(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'*  '(M1, M2) ->
    [[lists:sum(lists:zipwith(fun erlang:'*'/2, Li, Cj))
        || Cj <- M2]
        || Li <- M1].


%% return true if M1 equals M2 using 1e-6 precision
-spec '=='(M1, M2) -> boolean() when
    M1 :: matrix(),
    M2 :: matrix().
```

```
'=='(M1, M2) ->
    case length(M1) == length(M2) of
        true when length(hd(M1)) == length(hd(M2)) ->
            RoundFloat = fun(F) -> round(F*1000000)/1000000
                end,
            CmpFloat = fun(F1, F2) -> RoundFloat(F1) ==
                RoundFloat(F2) end,
            Eq = element_wise_op(CmpFloat, M1, M2),
            lists:all(fun(Row) -> lists:all(fun(B) -> B end
                , Row) end, Eq);
        false -> false
    end.


%% return the row I of M
-spec row(I, M) -> Row when
    I :: pos_integer(),
    M :: matrix(),
    Row :: matrix().

row(I, M) ->
    [lists:nth(I, M)].


%% return the column J of M
-spec col(J, M) -> Col when
    J :: pos_integer(),
    M :: matrix(),
    Col :: matrix().

col(J, M) ->
    [[lists:nth(J, Row)] || Row <- M].


%% return the element at index (I,J) in M
-spec get(I, J, M) -> Elem when
    I :: pos_integer(),
    J :: pos_integer(),
    M :: matrix(),
    Elem :: number().
```

```
get(I, J, M) ->
    lists:nth(J, lists:nth(I, M)).


%% return a null matrix of size NxM
-spec zeros(N, M) -> Zeros when
    N :: pos_integer(),
    M :: pos_integer(),
    Zeros :: matrix().

zeros(N, M) ->
    [[0 || _ <- lists:seq(1, M)] || _ <- lists:seq(1, N)].


%% return an identity matrix of size NxN
-spec eye(N) -> Identity when
    N :: pos_integer(),
    Identity :: matrix().

eye(N) ->
    [[ if I =:= J -> 1; true -> 0 end
        || J <- lists:seq(1, N)]
        || I <- lists:seq(1, N)].


%% return a square diagonal matrix with the elements of L
   on the main diagonal
-spec diag(L) -> Diag when
    L :: [number(), ...],
    Diag :: matrix().

diag(L) ->
    N = length(L),
    diag(L, zeros(N, N), 0, []).


%% compute the inverse of a square matrix
-spec inv(M) -> Invert when
    M :: matrix(),
```

```
    Invert :: matrix().

inv(M) ->
    N = length(M),
    A = lists:zipwith(fun lists:append/2, M, eye(N)),
    Gj = gauss_jordan(A, N, 0, 1),
    [lists:nthtail(N, Row) || Row <- Gj].


%% evaluate a list of matrix operations
-spec eval(Expr) -> Result when
    Expr :: [T],
    T :: matrix() | '+' | '-' | '*' | '*  ',
    Result :: matrix().

eval([L|[O|[R|T]]]) ->
    F = fun mat:O/2,
    eval([F(L, R)|T]);
eval([Res]) ->
    Res.

%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Internal functions
%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% transpose matrix with accumulator
tr([[]|_], Rows) ->
    lists:reverse(Rows);
tr(M, Rows) ->
    {Row, Cols} = tr(M, [], []),
    tr(Cols, [Row|Rows]).


%% transpose the first row of a matrix with accumulators
tr([], Col, Cols) ->
    {lists:reverse(Col), lists:reverse(Cols)};
```

```
tr([[H|T]|Rows], Col, Cols) ->
    tr(Rows, [H|Col], [T|Cols]).


%% apply Op element wise on matrices M1 and M2
element_wise_op(Op, M1, M2) ->
    lists:zipwith(fun(L1, L2) -> lists:zipwith(Op, L1, L2)
       end, M1, M2).


%% Gauss-Jordan method from
%% https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-
   Jordan#Pseudocode
gauss_jordan(A, N, _, J) when J > N ->
    A;
gauss_jordan(A, N, R, J) ->
    case pivot(col(J, lists:nthtail(R, A)), R+1, {0, 0}) of
        {_, 0} ->
            gauss_jordan(A, N, R, J+1);
        {K, Pivot} ->
            A2 = swap(K, R+1, A),
            [Row] = row(R+1, A2),
            Norm = lists:map(fun(E) -> E/Pivot end, Row),
            A3 = gauss_jordan_aux(A2, {R+1, J}, Norm, 1, []
               ),
            gauss_jordan(A3, N, R+1, J+1)
    end.


%% Matrix(i, :) -= Matrix(i, j)/Pivot * Matrix(R, :) forall
    i\{R}
%% Matrix(R, :) *= 1/Pivot
%% with Pivot = Matrix(R, j)
gauss_jordan_aux([], _, _, _, Acc) ->
    lists:reverse(Acc);
gauss_jordan_aux([_|Rows], {I, J}, L, I, Acc)->
    gauss_jordan_aux(Rows, {I, J}, L, I+1, [L|Acc]);
gauss_jordan_aux([Row|Rows], {R, J}, L, I, Acc) ->
    F = lists:nth(J, Row),
    NewRow = lists:zipwith(fun(A, B) -> A-F*B end, Row, L),
```

```
    gauss_jordan_aux(Rows, {R, J}, L, I+1, [NewRow|Acc]).


%% find the gauss jordan pivot of a column
pivot([], _, Pivot) ->
    Pivot;
pivot([[H]|T], I, {_, V}) when abs(H) >= abs(V) ->
    pivot(T, I+1, {I, H});
pivot([_|T], I, Pivot) ->
    pivot(T, I+1, Pivot).


%% swap two indexes of a list
%% taken from https://stackoverflow.com/a/64024907
swap(A, A, List) ->
    List;
swap(A, B, List) ->
    {P1, P2} = {min(A,B), max(A,B)},
    {L1, [Elem1 | T1]} = lists:split(P1-1, List),
    {L2, [Elem2 | L3]} = lists:split(P2-P1-1, T1),
    lists:append([L1, [Elem2], L2, [Elem1], L3]).


%% build a diagonal matrix from a zero matrix
diag([], [], _, Acc) ->
    lists:reverse(Acc);
diag([X|Xs], [Row|Rows], I, Acc) ->
    {L1, [_|T1]} = lists:split(I, Row),
    NewRow = lists:append([L1, [X], T1]),
    diag(Xs, Rows, I+1, [NewRow|Acc]).
```

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