UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

COMPUTATIONALLY EFFICIENT VISION-BASED ROBOT CONTROL

Matheus Terrivel
Master Thesis
December 2017

Examination Committee:
Prof. dr. ir. M.J.G. Bekooij
V. E. Hakim, MSc.
Ir. J. (Hans) Scholten

Computer Architecture for
Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
7522 NH Enschede
The Netherlands

Abstract

Video target tracking systems is a trending research topic, with a plethora of applications emerging from recent studies, with both visual object detection and tracking disciplines being the most notable, principally on embedded platforms. Not only are they employed in various fields, but also remarkably combines several branches of studies, such as control engineering, video processing, and more recently, machine learning and sensor fusion. Autonomous vehicles are a notable example, being equipped with a variety of sensors, including cameras, and widely apply image processing and sensor fusion techniques, thus providing more concise and high-level information, which increases robustness and more importantly, certainty, on decision making. However, such techniques must respect real-time constraints, especially in terms of timing, due the fact a delay might have a high cost under certain circumstances.

Currently, there is a broad interest in processing images with neural networks, which are superior in terms of performance and robustness in comparison to traditional image processing algorithms. Although the rapid development of image sensors in combination with neural network technology, computational power of the underlying platform is still a bottleneck, especially for embedded applications. Moreover, the platform is commonly responsible for multiple tasks, which might include a user interface, data processing, (digital) filtering and high-level control, thus cannot be fully dedicated to the neural network itself. Finally, modern system-on-chips comprise hardware accelerators and multiple processing cores, which enable embedded systems to accomplish the desired throughputs, and achieve better results and efficiency in comparison to pure software implementations. Most of the time, these SoCs are completely customizable and interaction between software and hardware is facilitated.

In this thesis, the focus is on both implementation and evaluation of a computational efficient robot control, based on neural networks to detect and localize a specific target (another robot), on an embedded platform. Sensor fusion and Kalman filtering are addressed, with the latter being used to post-process the output of the neural network, meanwhile the former combines encoder, accelerometer and gyroscope data which is used to derive the ego-motion information of the robot. Moreover, a specific approach for estimating the ego-motion impact in terms of pixels is proposed and discussed. The neural network, however, is not the focus, thus is only briefly discussed. During development, computational complexity, project extensibility and parallel tasks were the main concern, with the former being reduced whenever possible.

A complete working setup was the result of this endeavor, with the application being a toy example: its goal is to track a specific target, and follow it by means of controlling another robotic platform. Sensor fusion is accomplished with a complementary filter, which boosts the ego-motion speed accuracy and refresh rate. The estimation of the ego-motion impact, in terms of pixel movement, was implemented through approximation functions that are only dependent on the derived speed and gyroscope data, and proved to be fairly accurate without the need of complex procedures. The standard Kalman filters implemented proved to handle false-negatives and target occlusions in a reliable manner, and can be instantiated and run in parallel. Although the final implementation executes in a PC running Linux, the whole application can be easily ported to an embedded Linux.

Acknowledgements

Initially, I would like to laud my supervisor, Professor Marco Bekooij, for providing me with his orientation, involvement and valuable feedback and monitoring throughout the development of my master thesis. Furthermore, the proposed project constantly provided new challenges, in addition to comprising different fields which regularly encouraged me to amass relevant knowledge and pushed me outside the comfort zone. Not only did I improve academically, but also personally, hence I am extremely grateful for this opportunity.

Next, I would like to thank all colleagues in both Robotics and Mechatronics (RaM) and Computer Architecture for Embedded Systems (CAES) groups, more specifically Viktorio El Hakim, Oğuz Meteer, Zhiyuan Wang, Konstantinos Fatseas, and Kiavash Mortezavi Matin, for every moment spent together, either by keeping me company, listening to silly problems, sharing a coffee, discussing scientific and personal topics, and providing relevant feedback during the development of this thesis.

Subsequently, not only am I grateful to each person I came across during my studies abroad, but also to every friend back in Brazil. Thanks to your cheering and encouragement to pursue and fulfill my aspirations, I was able to smoothly surpass countless adversities and finally achieve my goals.

Ultimately, I dedicate all work to my family, more specifically my father Geraldo José Domingues Terrível and my mother Rosária de Campos Teixeira, which supported me unconditionally during my studies. I own this achievement to you both, and hope you recognize that. Moreover, I wish my effort will be re-used in the future to impact peoples' lives in a favorable manner.

Matheus Terrivel,
Enschede, November 22nd, 2017

Glossary

Term	Definition
0b__	Represents a binary value (usually an 8-bit value)
0x__	Represents a hexadecimal value (usually a 8 or 16-bit value)
ACK	Acknowledge – In some communications when a message is sent, the receiver sends back an Acknowledge package to confirm that it received successfully the previous information
bps	Bits per second – Refers to speed (i.e. baud rate) in a communication protocol
C, C++, C#	C, C plus plus and C sharp – Refers to programming languages
CRC	Cyclic Redundancy Check – Algorithm used to perform error check when transmitting data
DC	Direct Current
DoF	Degree(s) of Freedom – Refers to how many degrees of freedom a device has
EKF	Extended Kalman Filter – Non-linear version of the Kalman Filter
fps	Frames per second – Refers to camera frame rates
FPU	Floating-Point Unit – Unit that is dedicated to floating-point arithmetic operations, usually a dedicated piece of hardware
GND	Ground – Refers to ground or reference of a circuit
GPIO	General-Purpose Input/Output, usually related to a microcontroller pin
GPU	Graphics Processing Unit – Refers to a graphic card
I/O	Input/Output, normally refers to direction of a pin
I2C	Inter-integrated Circuit – Intra-board communication protocol
IC	Integrated Circuit – Refers to a small chip
IMU	Inertial Measurement Unit – Combination of motion sensors, usually a accelerometer, a gyroscope and a magnetometer
KF	Kalman Filter – Optimal filter for linear systems, mainly used for data fusion
LED	Light-Emitting Diode – Electronic component which emits a light
LSB	Least Significant Bit/Byte – Refers to ordination of bits/bytes
MCU	Microcontroller Unit
MSB	Most Significant Bit/Byte – Refers to ordination of bits/bytes
PC	Personal Computer
PCB	Printed Circuit Board
POSIX	Portable Operating System Interface
PWM	Pulse-width modulation – Modulation technique that sets an average voltage by switching on/off the voltage. Used for different kind of applications like RGB LEDs dimming
RMSE	Root-Mean-Square Error – Measure of differences between values
RPM	Rotations Per Minute
RT	Real-Time
SoC	System-on-Chip – Refers to an integrated circuit which comprises components of a computer and other electronic systems
UART	Universal Asynchronous Receiver/Transmitter – Serial protocol
UI	User interface
UKF	Unscented Kalman Filter – Non-linear version of the Kalman Filter, parallelism is a possibility

Table of Contents

Abstract 2
Acknowledgements 3
Glossary 4
Table of Contents 5
1. Introduction 7
1.1 Problem definition 8
1.2 Contributions 9
1.3 Thesis outline 10
2. Robotic system 11
2.1 Structure 12
2.2 Embedded hardware 13
2.2.1 MegaPi board 13
2.2.2 Sensors & Actuators 14
2.3 Library improvements 16
2.3.1 Speed Controller 16
2.3.2 IMU 18
2.4 Complementary Filter 19
2.4.1 Filter structures 20
2.4.2 Speed Estimation 22
2.4.3 Results 27
2.5 Communication protocol 34
2.6 Final Implementation 36
2.6.1 Hardware 36
2.6.2 Software 37
3. Tracking system 40
3.1 Camera 41
3.2 Neural Network 42
3.3 Pre-Kalman Filter 44
3.3.1 Translation impact 46
3.3.2 Rotation impact 52
3.3.3 Combined pixel speeds 57
3.4 Kalman Filter 59
3.4.1 Filter Design 61

3.4.2 Design Space Exploration 65
3.4.4 Real Data Analysis 75
3.4.5 Final Design 80
3.5 Pixel Control 81
3.6 Final Implementation 83
3.6.1 Hardware 84
3.6.2 Software 85
3.6.3 Drivers 90
4. System analysis 94
4.1 Delay impact 94
4.2 Real-time analysis 99
5. Conclusions and future work 102
6. Bibliography 104
7. Appendices 107
Appendix A: PID Controller 107
Appendix B: Pre-Kalman Filter Equations 108
B.1 Translation 108
B.2 Rotation 108
Appendix C: Embedded Software 109
Snippet 1. Encoder motor 109
Snippet 2. IMU module 110
Snippet 3. Pre-Kalman Filter 110
Snippet 4. Kalman Filter's prediction & update steps 112
Snippet 5. GoPro Stream Handler 114
Snippet 6. Neural Network with TensorFlow 115
Snippet 7. ffmpeg invocation and parameters 116
Snippet 8. MAPP compilation options 117
Snippet 9. POSIX: Message queue creation example 117
Snippet 10. POSIX: Thread creation example 118
Snippet 11. Float over serial example 119

1. Introduction

Video target tracking systems have become a trend research topic, especially for embedded systems, covering several fields such as image, signal and video processing, machine learning, pattern recognition, and control engineering. Moreover, multiple sensors and thus sensor fusion techniques have been applied in order to improve the tracking performance [1]. Not only do vision tracking systems enable machines to perform complex tasks, but also require less hardware requirements and are easier to be implemented in comparison to other traditional systems, such as radar- or LiDAR-based systems. Thus, video target tracking is widely applied in commercial applications, ranging from medical to autonomous vehicles. Whenever multiple sensors are used, sensor fusion techniques are used in order to derive relevant information and improve the overall performance of the system, with the Kalman filter and its variations being the most commonly used.

In terms of target detection and localization, the state-of-the-art techniques are based on neural networks, which have been replacing traditional image processing techniques due to their superior performance and robustness. However, neural networks are computationally expensive and other technologies are still utilized for object detection in embedded systems, such as radar, due the fact data processing is faster, thus real-time processing is possible. In terms of sensors, most video target tracking systems comprise cheap and accurate inertial measurement units (IMUs), in addition to (mono- or stereo-)cameras, mainly focusing on improving the tracking algorithm and performing Simultaneous Localization and Mapping (SLAM). However, multiple sensors increase overall complexity of the system, which must include sensor fusion techniques, hence requires even more resources from the underlying platform.

The rapid development of image sensors enables stable streams with high framerate and image quality, easily achieving 60 fps at 1080p or higher resolution, which contribute to increasing the target recognition, especially for neural network-based implementations. Moreover, advances on sensor technologies provide lightweight and more accurate devices. Over the last decade, however, most video tracking systems had been implemented on a remote computer or cluster, equipped with high performance CPUs and GPUs, specifically addressing the computational complexity issue of both image processing and sensor fusion techniques. The remote device then transmits back commands to the embedded platform, which introduces latency for (mobile) applications, decreasing their performance because real-time requirements cannot be guaranteed. Moreover, standard computers are not power-efficient, expensive and relatively large, hence not feasible for mobile applications.

More recently, several system-on-chips significantly improved their hardware capabilities, enabling data processing to be performed in the embedded platform itself, as the hardware conditions do not represent a bottleneck to implementing complex video processing and sensor fusion algorithms. Considering such SoCs comprise hardware accelerators and multiple processing cores, being most of the time completely customizable and simplifying interaction between software and hardware, embedded systems are capable of achieving the desired throughputs, achieving better results and efficiency in comparison to pure software implementations.

Initially, the main goal of this project was to utilize the ZYBO [2] board, which comprises a Zynq-7010 SoC [3] with a dual ARM Cortex-A9 and FPGA fabric. Moreover, it is equipped with many peripherals

such as a dual-role HDMI port, and supports bare-metal code, alongside embedded Linux and hardware accelerators. Hence, the focus of this thesis is to implement and evaluate a video tracking system with object recognition and tracking, which exploits both the FPGA fabric for executing the neural network – the most computationally expensive part of the application –, and the ARM Cortex-A9 for data fusion and high-level control. The application is a toy example, and its goal is to not only track a specific target, but also follow it by means of controlling another robotic platform equipped with sensors and actuators. This thesis summarizes and describes the whole design process and decisions during the development of such system, except the neural network hardware implementation. More specifically, due to the fact the neural network in the end of this project was still not ported to the FPGA, a PC was used instead. However, decisions during the project were made taking into account the underlying platform would be an embedded system with limited resources, thus the final implementation can be easily ported to the ZYBO board, for instance.

This chapter introduces the research problem: initially, the problem is explained in more detail and the major research objectives are defined; subsequently, the contributions of this thesis are presented and briefly discussed, followed by the outline of this report in the last section.

1.1 Problem definition

Porting a visual tracking system that comprises both tracking with a neural network and sensor fusion is not an easy task. A pure software solution demands vast computational capabilities, with the neural network being extremely computationally intensive due the required operations performed on the frames. A pure hardware solution, on the other hand, is limited by the amount of resources available, and is commonly limited in terms of flexibility and configurations. Recent solutions utilize multiple sensors and complex sensor fusion techniques in order to navigate, instead of perform tracking, such as SLAM [4], Extended Kalman Filters [5, 6, 7, 8], Unscented Kalman Filters [8, 9, 10], among others, in addition to some combining different techniques [5, 6, 9] which demands even more resources, thus computational power. Camera-IMU setups are widely applied for positioning [10, 11], which requires online camera external parameters calibration, performed by Kalman filters. Neural networks are rarely used in mobile applications due its complexity and execution time which restrains real-time requirements, with either traditional image processing techniques being applied instead, or third-party libraries used, such as skeleton tracking [4], for instance. Moreover, whenever a neural network is indeed utilized for object detection and tracking, it is executed in a remote PC [12], and filtering its output is rarely implemented, which greatly degrades the performance in occlusion scenarios [4] or false-negatives. Instead, Kalman filters usually are applied to pre-process data which is inputted to the neural network [13, 14].

Considering the aforementioned facts, this thesis addresses an alternative solution in order to detect and localize an object with a neural network, taking into account the ego-motion impact and filtering the output of the neural network with a robust filtering technique, in order to properly steer a robot and follow a target, all implemented in a resource-constrained embedded system. Moreover, this thesis stresses the following objectives:

1. Implement a low-level system which is responsible for sensor fusing data and steering a robot:
    a. Using a resource-constrained embedded system;
    b. Considering both encoder and IMU data;

    c. Improving the refresh rate of the ego-motion data;
    d. Properly controlling the speed;
    e. Providing ego-motion data to a higher-level system.
2. Realize a robust visual object tracking system:
    a. Using software which interacts with hardware in a resource-constrained embedded system;
    b. Using a neural network-based detection and localization implementation, ideally on hardware – implementation of the neural network is NOT addressed, however;
    c. Coupling a Kalman filter to improve stability and addressing both tracking with occlusion and possible false negatives outputted by the neural network;
    d. Retrieving ego-motion data from the lower-level system;
    e. Considering the ego-motion impact on the target being tracked, in pixel speeds;
    f. Interacting with the low-level system in order to follow the target.
3. Analyze and evaluate the complete system:
    a. Considering delays;
    b. Identifying bottlenecks;
4. Test the complete system in practice.

In summary, the research questions are as follows: *Is it possible to implement application which comprises tracking and sensor fusion techniques on a practical embedded system? Furthermore, what are the complications for such system?*

1.2 Contributions

In summary, this thesis contributes to five major points. Initially, it (1) addresses the usage of a complementary filter to fuse encoder and IMU data, more specifically accelerometer and gyroscope information. Not only does such filter reduces computational complexity, but also boosts the ego-motion speed accuracy and refresh rate, being particularly useful when low-cost and low-accuracy sensors are deployed, although this technique is currently not often applied.

Secondly, the ego-motion, more specifically speed and angular velocity with respect to heading, is considered and its impact on the object being tracked is estimated by a simpler (but specific) method, which does not require computationally expensive algorithms (2). The output of the neural network applied for detection and localization is additionally filtered by standard Kalman filters, which are based on a linear model and their implementation do not depend on matrix operations, reducing even further the require computational power of the underlying platform (3). Moreover, the derived ego-motion impact is used by the Kalman filters, making it possible to keep tracking the target when occlusions occur, in addition to common false-negatives outputted by the neural network (4).

Finally, distance control is abstracted by investigating pixel control instead, hence complex translations between camera and world coordinates are avoided (5). Furthermore, two PID controllers are combined in order to simplify steering the robot.
With respect to internal contributions, this thesis describes an implementation that was tested in practice and validates the correct behavior of the integrated system, combining multiple techniques

from different areas. Hence, the implementation is a complete base system in terms of structure, hardware and software, which can be applied in a variety of applications.

1.3 Thesis outline

Heretofore, the topic of this thesis and related research has been introduced. In the upcoming chapters, the whole system is extensively discussed, with its complete overview being depicted in Figure 1 below:

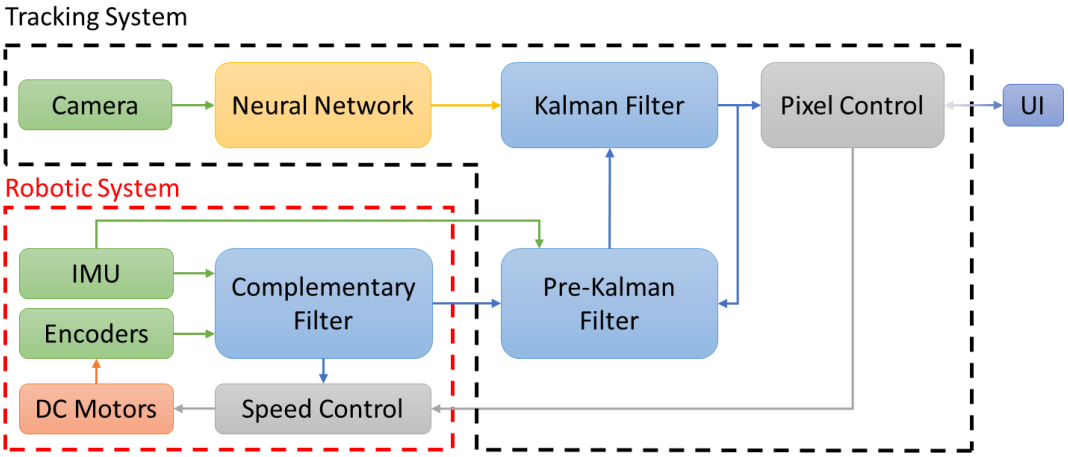


Figure 1. Complete system overview

Chapter 2 describes the robotic system (highlighted in red), by introducing the overall structure, the related embedded hardware, improvements done to the standard libraries, followed by the complementary filter used for fusing encoder and IMU data and the respective results. Finally, the protocol used for communicating with this system is discussed and the final implementation detailed.

Chapter 3 is the unit of this thesis, and addresses the tracking system (highlighted in black), briefly introducing the camera used and the neural network scheme. Subsequently, the Pre-Kalman filter module is detailed, which is responsible for translating the ego-motion impact to pixel speeds which are forwarded to the Kalman filter. The Kalman filter used for filtering the output of the neural network is then assessed, regarding its design and additional algorithms applied specifically for tracking, alongside the obtained results. Subsequently, the Pixel Control module is presented and explained, which is responsible for computing the speed and direction that must be forwarded to the robotic system in order to it properly follow the target. Finally, the final implementation in terms of both hardware and software related to the tracking system is explored, including the drivers developed for the Kalman filters, PID controllers and communication with the robotic system.

Chapter 4 analyzes the complete system behavior, exploring the impact of the delay on the performance, alongside a basic dataflow graph discussion to explore RT analysis techniques.

Finally, Chapter 5 concludes the thesis and presents future work and final thoughts.

2. Robotic system

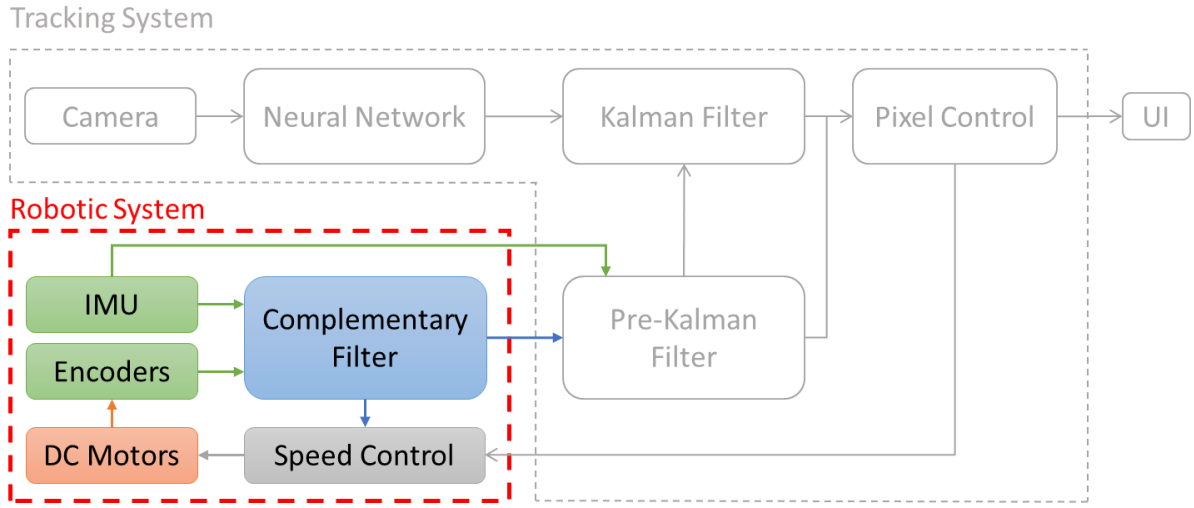


Figure 2. Robotic system outline

In order to introduce the complementary filter, which is the first main part of this project, the underlying robotic system used – the Makeblock Ultimate Robot Kit V2.0 10-in-1 [15] -, with its mechanical structure, key hardware modules likewise extra ones and their capabilities, are briefly discussed. Moreover, improvements with respect to the default software libraries are assessed, considering the desired functionalities of the complementary filter itself in addition to overall improvements, such as the low-level speed controller. Finally, the filter is explained and analyzed, alongside the micro-protocol used for communicating with the robotic system, as well as the final architecture in terms of both software and hardware. In summary, the highlighted blocks shown in Figure 2 above are detailed in the upcoming sections.

The Makeblock Ultimate Robot Kit has many structural parts, a mainboard – the MegaPi -, Bluetooth connectivity and several (proprietary) modules. This kit was readily available in the beginning of the project and comprised, among others, an Inertial Measuring Unit (IMU) module, DC motors coupled with encoders, and their respective drivers, which are required for the final application: follow an object. Moreover, it is simple to use and tailor to this application, without much effort. All development tools are provided in the seller's website, and the system can be fully customized by the user, in terms of software and structure. Hardware components, on the other hand, are complete modules and is up to the user to integrate them or not. For a detailed description, one should refer to [15].

Although the complementary filter will be extensively discussed later in this chapter, it is important to define its inputs and outputs briefly due the fact most design decisions were based on what it requires for the desired behavior. In summary, the complementary filter is used to improve and complement the speed estimation procedure, and fuses data from both encoder and accelerometer, which is relevant for the tracking system. Moreover, the tracking system requires data from a gyroscope, but this will be further explored in Chapter 3. Taking this into consideration, the complementary filter's inputs are: linear speed which is derived from linear accelerations retrieved from an accelerometer,

and the linear speed derived with the encoder. The output, on the other hand, is the fused linear speed, with the gyroscope data being directly forwarded to the tracking system.
Furthermore, a set of basic requirements and functionalities was defined for the robotic system, which should follow another robot, and thus must be able to:

- Move forwards, backwards, and turn;
- Support a camera on top: the frames will be further processed by the tracking system;
- Process data from an IMU: necessary for the complementary filter;
- Control the speed of the motors: low-level speed control;
- Receive speed references from the tracking system;
- Send relevant data to the tracking system: current speed and angular velocities.

2.1 Structure

The kit used in this project has several mechanical parts, such as plastic gears, acrylic supports, screws, and metal part, which can be assembled in different ways. Although there are default configurations provided by the seller alongside the respective piece of software, such as “Robotic Arm Tank”, “Self-Balancing Robot” and “Robotic Bartender” [15], for this project a custom setup was built which contemplates a base structure with two tracks driven by two separate DC motors, and two elevated platforms: a shorter one for supporting the battery, and a taller one to accommodate the camera. Finally, a support for the main board is required and should be located as close as possible to the modules used. The structure used is shown in Figure 3 below:

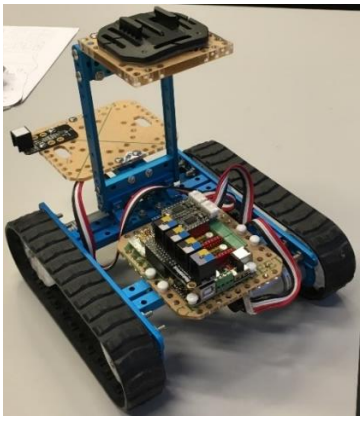


Figure 3. Robotic system structure

The tracks have been utilized due the fact driving the robot in a different manner would overcomplicate its odometry model and drive procedure, especially when changing direction. Not only do the tracks simplifies the operation, but also improves robustness due the fact they overcome obstacles easier. Furthermore, this structure complies with the previous requirements:

- Move forwards, backwards, and turn: with tracks and two separate motors, the robot can be steered properly;
- Support a camera on top.

The battery platform is necessary for powering the system, but in this case also provides a stable support for the IMU module. Moreover, adding modules and (most likely) structures were taken into consideration, thus this platform can still be augmented in the future.

2.2 Embedded hardware

Besides the mechanical parts, the kit includes a microcontroller board based on ATmega2560 [16] – the MegaPi, which is both Raspberry Pi and Arduino compatible. Programming is done through USB, and for this application only the Arduino IDE was necessary during development, although it is possible to use different development tools. Moreover, the kit comprises several electronic modules which are plug-and-play, with the most relevant ones being: Bluetooth, DC motor driver, ultrasonic sensor, RJ25 shield for I2C/UART modules, 3-axis accelerometer and gyroscope sensor, and a compass. In total, three (3) encoder motors are also included, apart from the aforementioned modules, which are DC motors coupled with encoders. Cables are provided and connections are simple to use, thus will not be explained in this section. The complete parts list can be found in [15].

With respect to hardware requirements, the system must support, at least:

- Driving 2 DC motors simultaneously;
- IMU connection for the complementary filter;
- Serial communication or (optimally) wireless module for exchanging data with a master and debugging;
- Floating-point operations for control loops, filtering and data processing.

2.2.1 MegaPi board

The MegaPi, as explained before, is based on the ATmega2560 microcontroller and can drive up to 10 servo motors, 8 DC or 4 stepper motors simultaneously. Additionally, most I/O pins are exposed and can be used for custom hardware and software implementations, as depicted in Figure 4 below. Note that the motors' circuitry is highlighted in purple, power input and switch in gray, Bluetooth in orange and the microcontroller itself in yellow.

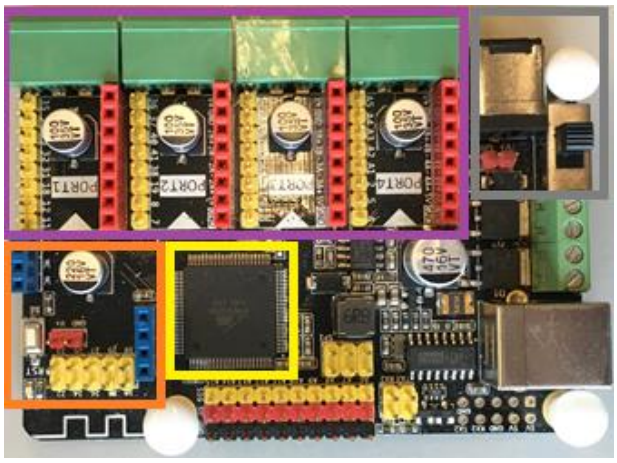


Figure 4. MegaPi board, with highlighted relevant parts

For this project, the MegaPi board and modules are sufficient and meets the requirements:

- DC motors: it is capable of driving up to 4 simultaneously;
- IMU: a 3-axis accelerometer and gyroscope sensor are included in the kit;
- Serial communication or wireless module: in addition to the Bluetooth and USB (Virtual COM), it supports an extra UART connection;

Although the ATmega2560 is an 8-bit microcontroller, it has software support for floating-point operations which are typically fast (< 1 milliseconds for division), and commonly used for control loops and filtering techniques. Taking into consideration that both the complementary filter and data processing (for the IMU) are simple and do not require many calculations, this platform is capable of performing the necessary operations on time.

Finally, it is important to note that this board operates at 5V, and its power-in range must be between 6-12V, in case a battery is used. The complete specifications of the board and ATmega2560 can be found in [15] and [16], respectively. In terms of software, the MegaPi has a complete open-source library, available at [17]

2.2.2 Sensors & Actuators

Before defining and integrating all components, it is important to consider both the available sensors and actuators. Despite that several electronic modules are included in the kit, only few are required for the application:

- IMU unit, more specifically an accelerometer and a gyroscope, for the complementary filter;
- DC motors for movement;
- DC motor drivers;
- Encoders for speed control;
- Wireless module for communication: optional, as a wired connection is more reliable.

This section describes the motors used, alongside the feedback sensors necessary for speed control and finally the IMU required by the complementary filter.

Encoder motors

In total the kit includes three (3) 25mm encoder motors, which are DC motors coupled with an incremental encoder: all three operate at 9V, but only two can achieve a higher speed (185 RPM ±10%) with the other being slower (86 RPM ±10%). The latter, however, has a higher torque in comparison to the former, and is typically used for lifting a robotic arm, which is not the case for this application. Thus, both identical encoder motors are used for driving the robot, one on each side. The encoder motor is depicted in Figure 5a below, with the encoder being in the back, close to the proprietary connection. Other specifications can be found in [18].

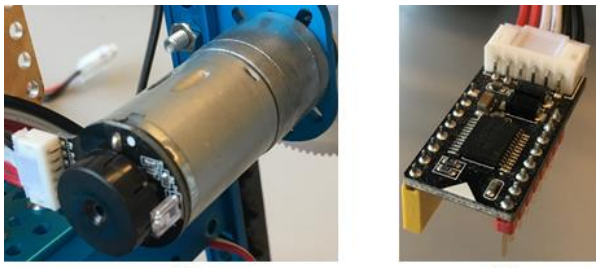


Figure 5. Encoder motor (a) and its driver (b)

DC motors are usually controlled with an H-bridge, which translates a low-voltage PWM input to the required proportional voltage (nominal voltage of the motor) output. Moreover, it can be stopped in

different ways – coasting (free rotation) or braking – and its rotation direction can also be determined. Fortunately, the kit also provides the driver module for the encoder motors, which is shown in Figure 5b above, and must be place in either slot 1 or 2 of MegaPi (check Figure 4). Moreover, the encoder connections are properly routed to the microcontroller pins in order to derive the speed.

There are different ways of keeping track of a motor's RPM, with tachometers and hall-effect sensors being an example. However, when considering all aspects of each technology, quadrature incremental or absolute encoders stand apart due their price, size, availability, and reliability. Not only do encoders require less mechanical components, but also are easy to mount and require simple data processing. A quadrature relative or incremental encoder measures a change in position, and usually outputs two channels (A and B – the quadrature signals) and one index pulse (X). The former signals are toggled in a certain order depending on the rotation direction (clockwise or counter-clockwise), and the latter is pulsed once every complete revolution (Figure 6). The amount of pulses per revolution (PPR) outputted by a one channel is stated in the datasheet, thus rotations per second can easily be derived and converted (e.g. to Hz, RPM) either from the two output channels, or from the index pulse itself by measuring the time between one or several pulses, which is less accurate (i.e. integer number of revolutions). Hence, such encoder provides rotational velocity, direction, and relative position feedback, and one per motor is required.

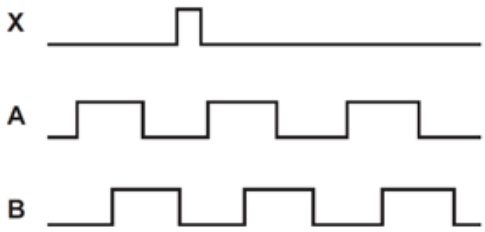


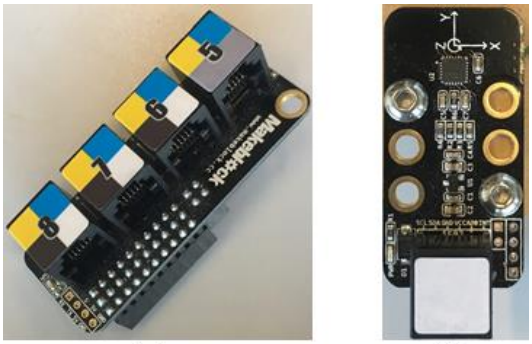
Figure 6. Quadrature signals and index for counter-clockwise rotation (a), outputted by the incremental encoder (b)

Although the encoder motor already includes an incremental encoder, no datasheet is provided by the supplier. This is not a problem, as by visually inspection it is possible to determine it is a low-quality device with 8 PPR, thus low resolution, which negatively impacts the speed control. However, the supplier does provide the necessary drivers [3] to derive the RPM and no extra hardware nor mechanical parts are needed. With respect to software, the "MeEncoderOnBoard" driver is used for initializing, configuring, and utilizing the encoder motors. Check Appendix C: Embedded Software for further details.

IMU

An Inertial Measurement Unit (IMU) usually embeds three sensors: accelerometer, gyroscope and magnetometer. These devices are often cheaper than individual sensors, and provide complete inertial information, which is relevant for many applications, such as dead reckoning and simultaneous localization and mapping (SLAM). The accelerometer is a compact electromechanical device designed to measure linear acceleration forces (i.e. change in speed). A gyroscope, on the other hand, is small and inexpensive sensor that measure angular velocity, either in degrees per second (°/s) or revolutions per second (RPS). It can be used to determine orientation through data fusion, and is found in most autonomous navigation systems, usually combined with accelerometers for improved performance with a Kalman filter or a complementary filter.

One of the modules included in the kit is a 3-axis accelerometer and gyroscope (Figure 7b), which has been and will be referred to an IMU throughout this document, even though the magnetometer is not involved. Moreover, the application only requires data from an accelerometer and gyroscope, with the former being used to determine the linear speed, meanwhile data from the latter is directly forwarded to the tracking system. Hence, the magnetometer will not be discussed in this section.



(a) (b)

Figure 7. RJ25 shield (a), and 3-axis accelerometer and gyroscope module (b)

The module shown in Figure 7b above is an I2C device and both accelerometer and gyroscope full-scale ranges are programmable, with the core IC being the MPU6050 – full specifications can be found in [19]. Moreover, the module uses a RJ25 connector and must be correctly connected to the MegaPi board through the RJ25 shield (Figure 7a). Note that the module has a white label on top of its connector, which must match one of the colors on top of the RJ25 shield. In this case, it should be connected to connector number 6, 7 or 8. Usage is straight forward with respect to software, which depends only on the "MeGyro" driver for initializing, configuring, and utilizing the encoder motors – Check Appendix C: Embedded Software for further details.

2.3 Library improvements

The processing board and its compatible modules work out-of-the box with the open-source library provided [3]. More specifically for this application, however, two drivers are used: "MeEncoderOnBoard" and "MeGyro", which handle the encoder motors and the IMU module, respectively. By exploring the source code of both libraries, one may find points of improvement, or customize it in order to add specific functionalities, for instance. The former relates to a problem found in the speed control, and the latter to instability issues and a lack of flexibility with the data retrieved from the IMU module. The main goal of modifying such drivers is to improve the overall behavior, robustness and stability of the robotic system, and in this section the major modifications are presented, further explained and justified.

2.3.1 Speed Controller

The first issue addressed is the speed controller provided by the "MeEncoderOnBoard" driver, which handles the encoder motors. Although the MegaPi documentation indicates a PID controller is used, its implementation is a simple proportional one. Initially, the performance of the proportional controller was assessed in order to decide whether a more complex would be implemented. The proportional gain was set to 0.18, and the sampling frequency to 25 Hz which are the default values

according to the documentation. It is important to notice that the reference must always be provided in RPM, and in this scenario the same speed is provided for both encoder motors. Initially, however, both sides were tested and presented highly similarities, thus the upcoming results depict one side only. Two metrics are used to compare the performance: the settling time for 95%, and the root-mean-square error (RMSE), with the latter being computed with the following formula:

$$RMSE = \sqrt{\frac{1}{N} * \sum_{i=1}^{N} (reference_i - sensor_i)^2} \tag{2.1}$$

The performance is shown in Figure 8 below, with the reference depicted in red and the actual sensor (i.e. encoder) readings in green.

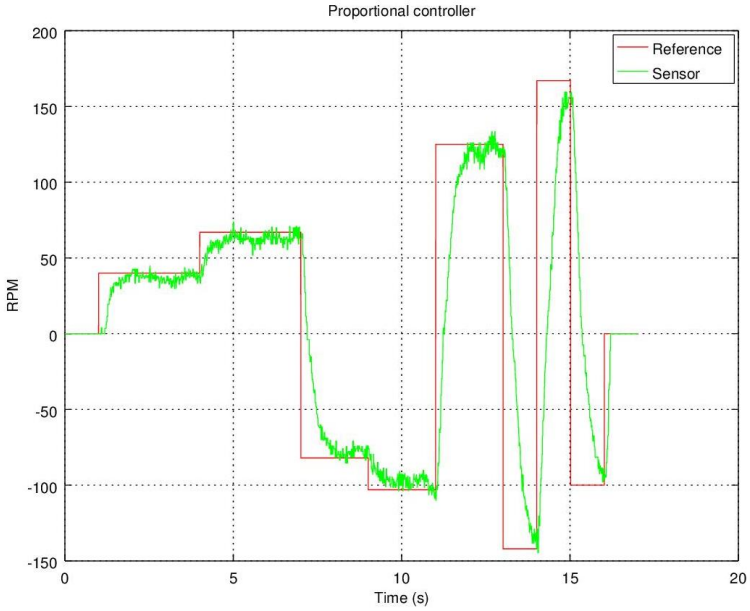


Figure 8. Proportional controller response

Even though the controller behavior is satisfactory, abrupt reference changes result in a slow response, especially when changing direction, with this implementation having a 0.8 second settling time (95%) and a RMSE of 63.08 RPM. Thus, prior to implementing the complementary filter described in the next section, the speed controller of the robotic system was modified to contemplate a PID controller, instead of a simple proportional one, which theoretically could improve overall performance, and more specifically the settling time. After implementing the PID controller, the response was evaluated in the same manner as before, and the result is shown in Figure 9 below:

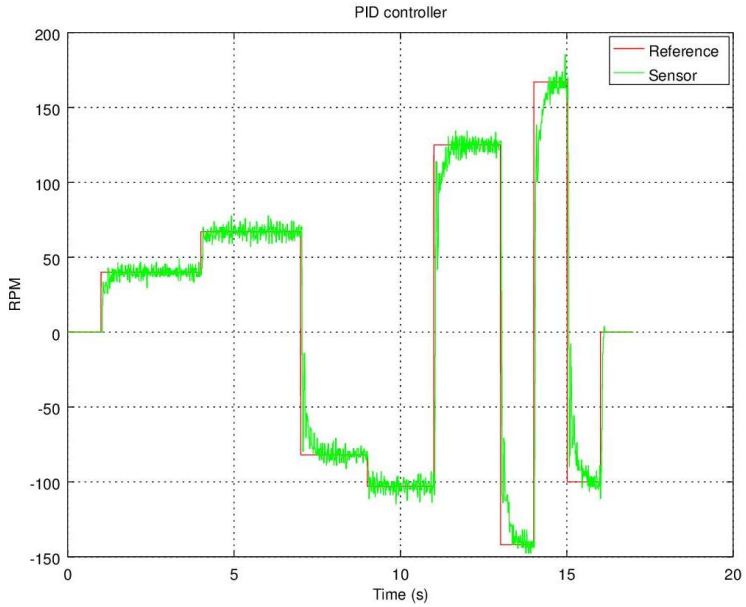


Figure 9. PID controller response

In this setup, the proportional (K_p), integral (T_i) and derivative (T_d) gains were tuned to 1.7, 0.1 and 0.0001, respectively, and the sampling frequency was increased to 50 Hz. With such configuration, the settling time was reduced to 0.3 second, corresponding to a reduction of 62.5% in comparison to the original controller. Moreover, overshoots do not surpass 5% of the reference value, and the RMSE was decreased to 27.86 RPM (55.83% reduction). The oscillation in the steady state will be further addressed by the complementary filter, which outputs a more stable speed value. Additional information about the PID controller, its implementation and tuning procedure can be found in Appendix A: PID Controller.

2.3.2 IMU

The IMU library is called “MeGyro”, although is comprises both an accelerometer and a gyroscope. During initial tests with the module, a few drawbacks were discovered:

- The calibration procedure did not take into consideration the accelerometer offsets (advised in the datasheet [19]);
- The driver only outputted fused (with a complementary filter) gyroscope data: yaw, roll and pitch. Hence the name “MeGyro”;
- The driver only outputted processed data – raw data from the sensors could not be retrieved. Note that raw data corresponds to integer values directly retrieved from the sensors, meanwhile processed considers both offset and sensitivity, resulting in a meaningful value and unit (i.e. m/s^2 and °/s);
- If the module was placed nearby metal parts, the system would often halt during initialization due magnetic interferences.

Initially, the calibration procedure was addressed in order to stabilize and reduce the bias of both gyroscope and accelerometer readings. Implementation is rather simple: the last step of the initialization is dedicated to retrieving raw data from the sensors several times (currently 500x), which are averaged in the end and internally saved as the offsets. Such values are later used when the user

requests processed data from the driver, by simply deducting the offsets from the raw readings. Note that calibration is always performed upon startup, thus the robotic system must always be in a flat surface and not be disturbed during initialization, otherwise behavior might be unpredictable due to wrong readings.

As the complementary filter input requires a speed value derived from the IMU, it is necessary to make the accelerometer data available in addition to the gyroscope data, which will further be used to calculate the speed. Additionally, it is rather advantageous to provide both raw and processed data retrieved from the sensors for debugging purposes. These modifications were both addressed simultaneously, by including specific functions to the library. Due the fact unique methods were added, usage of this driver is different from the original version. For more details, one should refer to Appendix C: Embedded Software.

Data signaling depends on how the module is placed with respect to the robot, and will be discussed in the end of this chapter. It is extremely important to keep the module away from metal parts and motors, and ideally it should be placed on top of an acrylic platform. Otherwise, the robotic system will most likely malfunction due to interferences and halt during initialization.

Finally, the accelerometer and gyroscope sensitivities were set to 2g and 500 °/s, respectively. The former corresponds to the highest resolution, and was used due the fact the linear accelerations are as precise as possible, which decreases errors for the following speed estimation step. The gyroscope sensitivity, on the other hand, can be further decreased to 250 °/s which corresponds to the best resolution, but it is not required for this application: based on tests, the robotic system is not capable of turning faster than 360 °/s. Furthermore, both sensitivities constrain the sampling frequency, and for this specific configuration up to 1kHz can be achieved. This is enough for both complementary filter and tracking system (most likely running below 200Hz). The former benefits from this high sampling frequency, as currently bottleneck the sampling frequency for the speed is restrained by the encoder readings.

2.4 Complementary Filter

Estimating the ego-motion of the robotic system is an important part of the application, considering such information will be further used by the Kalman filter and directly impacts the overall performance. Thus, being able to estimate the speed as precise and fast as possible is required. One might assume the encoder readings are enough, but in fact it only updates the readings at 50Hz, with the software improvements previously discussed. The IMU readings, on the other hand, are considerably faster for the presented sensitivities: 1kHz. Hence, by fusing both sensor data it is possible to achieve a sampling frequency of 1kHz in a low-cost platform.

The Kalman filter is widely used in sensor fusion scenarios, however it might be rather complex depending on the sensors applied and computationally expensive, especially due the amount of floating-point operations. Considering the microcontroller being used has a software floating-point support, and will be performing other tasks which require floating-point operations, another approach is needed. One simplistic alternative to the Kalman filter is the complementary filter [20], which is less complex, less computationally expensive, but yields similar results [21]. As mentioned before, the

original implementation of the IMU module ("MeGyro" driver) utilized a complementary filter to fuse accelerometer and gyroscope data and output less noisy and more accurate angular speeds (i.e. yaw, roll and pitch). Such technique is widely used for this type of sensor fusion, with many practical examples found in literature [21, 22, 23]. Furthermore, other applications use the same technique but the interest is specifically on estimating heading (i.e. yaw) and attitude [24, 25]. In summary, the complementary filter enhances accuracy, increases the sampling frequency, reduces complexity, and can be implemented in a low-cost platform as well. Hence, this section further explores the complementary filter (1st and 2nd order), addresses the necessary speed estimation techniques, and finally presents comparisons and results.

2.4.1 Filter structures

The 1st order complementary filter corresponds to two filters in parallel, with the same cut-off frequency: a high-pass and a low-pass filter. Moreover, the filter's name corresponds to the relations between both high- and low-pass filter gains, which are complementary: $G_{HPF} + G_{LPF} = 1$. The general structure of the 1st order complementary filter for the robotic system is depicted in Figure 10 below.

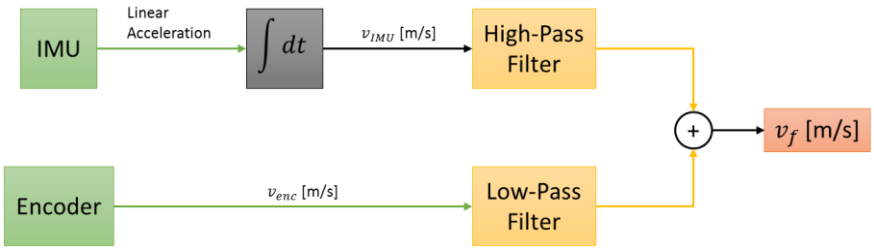


Figure 10. First-order Complementary Filter

The generic implementation of the 1st order complementary filter is straight-forward:

$$out_n = K * input_{1_n} + (1 - K) * input_{2_n} \tag{2.2}$$

$$K = \frac{f_c}{(T_s + f_c)} \tag{2.3}$$

With K being the time constant, f_c the cut-off frequency, and T_s the sampling period. The timing constant can be interpreted as a boundary between trusting one reading and the other, being usually tuned in practice, even though there are methods in literature to calculate it [23]. Note that K can be computed on-the-fly, if the T_s (or f_s) is correctly measured and the cut-off frequency is known *a priori*. However, f_c is commonly unknown thus K is defined or computed offline. The high-pass filter gain corresponds to K itself, meanwhile the low-pass filter gain is the complement: $K - 1$. Thus, both inputs must be carefully selected to match the desired filters.

The linear speed is estimated through integration based on the linear accelerations outputted by the IMU which naturally has a tendency to drift over time, hence it is reliable on the short-term and a high-pass filter is used. The encoder readings, on the other hand, are more reliable on the long-term, thus a low-pass filter is utilized. For this application the cut-off frequency is unknown, thus the time

constant is not computed dynamically, but defined as a constant, and by considering all the aforementioned information, (2) becomes:

$$v_f = K * v_{IMU} + (1 - K) * v_{enc} \tag{2.4}$$

With v_f being the fused speed, v_{IMU} the speed derived from the IMU, and v_{enc} the speed derived from the encoder.

A 2nd order complementary filter might be implemented as well, because it theoretically yields a better result at a cost of a more complex structure [23, 25], which involves two integrations as shown in Figure 11 below.

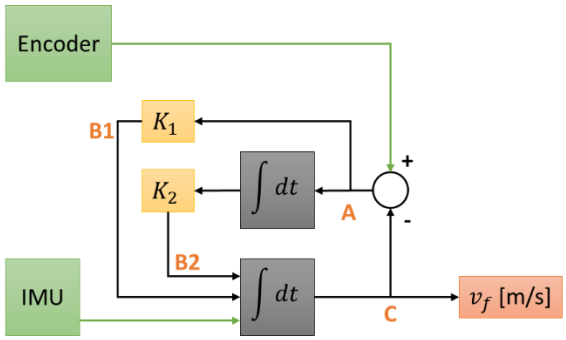


Figure 11. Second-order Complementary Filter

Note that K_1 and K_2 are positive gains, typically tuned in practice, and can be related as follows to simply work with a single gain K:

$$K_1 = 2 * K \tag{2.5}$$

$$K_2 = K^2 \tag{2.6}$$

Implementation is more complex, and can be divided into intermediate steps. Based on Figure 11, one might use the following equations to realize the 2nd order complementary filter:

$$A_n = input_{2_n} - out_{n-1} \tag{2.7}$$

$$B_1 = A_n * 2 * K \tag{2.8}$$

$$B_{2_n} = A_n * T_s * K^2 + B_{2_{n-1}} \tag{2.9}$$

$$C_n = B_1 + B_{2_n} + input_{1_n} \tag{2.10}$$

$$out_n = C_n * T_s + out_{n-1} \tag{2.11}$$

For this application, (2.7), (2.10) and (2.11) become:

$$A_n = v_{enc_n} - v_{f_{n-1}} \tag{2.12}$$

$$C_n = B_1 + B_{2_n} + v_{IMU_n} \tag{2.13}$$

$$v_{f_n} = C_n * T_s + v_{f_{n-1}} \tag{2.14}$$

It is important to notice that regardless the filter order, an integration method must be implemented to estimate the speed, which will be addressed in the next section. Obviously, both inputs must be provided in the same unit and in this case, meters/second was used.

2.4.2 Speed Estimation

Integration is strictly related to the complementary filter, due the fact a typical 1st order implementation requires so. More specifically, the accelerometer provides 3-axis linear accelerations which must be integrated to derive the speed, as follows:

$$v(t) = \int_0^t a(t)dt \tag{2.15}$$

With $v(t)$ being the linear speed, and $a(t)$ the linear acceleration, for a single axis. Notice that integration is related to continuous time, but the system is digital, thus a summation is used for the implementation. In this case, the speed at moment n is computed by summing the previous speed ($v_n = 0, n < 0$) and the value retrieved with an approximation rule, being generalized as follows:

$$v_n = v_{n-1} + \{approximation\ rule\} \tag{2.16}$$

Although there are many ways of estimating the speed in a robotic system, in this setup the procedure was purely encoder based. Such method delivers fairly accurate results, basically dependent on the encoder resolution itself and does not require an integration method for computing the speed, but may be improved by sensor fusing the IMU data that, on the other hand, does need an integration procedure. Furthermore, the interest is in estimating a 1D speed for this application, that basically corresponds to a forward or backward movement due to the robot's constraints (tracks): it can neither move sideways or up/down. Although the robot might be yawing, rolling or pitching, only the latter affects the 1D speed component. This section discusses optional approaches for speed estimation, considering the available hardware and computational capabilities of the underlying platform.

Accelerometer based

As previously stated, 3-axis linear accelerations are retrieved from the accelerometer, in m/s^2, and in order to estimate the speed an (digital) integration method must be implemented. Moreover, the accelerometer should be correctly calibrated for its static linear accelerations (offset), and go through the following procedures [26] in order to yield reasonable results:

1. Low-pass filter for noise reduction;
2. Window-filter for mechanical noise reduction;
3. Movement-end check to force the integration to 0, when the system has stopped.

Notice that these procedures must be applied to all relevant axis for the application, which in this case is the one corresponding to moving the robotic system forwards or backwards. The whole procedure

for estimating the speed is depicted in Figure 12 below. Due the fact the offsets are computed during the IMU initialization and automatically used when processed data is requested, the low-pass filter procedure for noise reduction is initially addressed. Reducing the noise is critical in order to decrease major errors during the integration procedure, which is performed by both the low-pass and window filters [26].

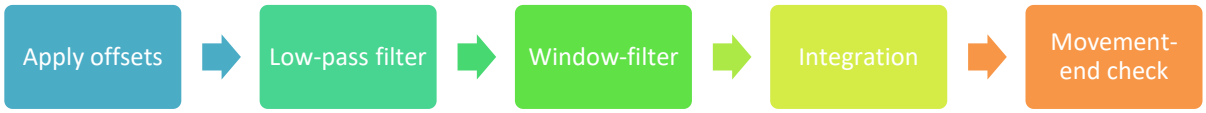


Figure 12. Accelerometer-based speed estimation procedure

One of the simplest implementations of a low-pass filter in digital systems is averaging: high-frequency disturbances are filtered out. However, simple averaging adds unnecessary delay to the system, as at least N samples must be averaged before outputting a filtered value. There are alternatives that specifically address this issue, such as the moving average: the output is computed whenever a sample is retrieved by averaging it with the previous $N-1$ samples. Notice, however, that this method still comprises a delay of N samples, but only during initialization. Another issue of the moving average (and simple average) is the division, which is computationally expensive. The exponential moving average (EMA) is then introduced, which yields surprisingly similar results with respect to the moving average, does not require division operations and is simpler to implement. Taking these factors into consideration, alongside the underlying platform, the EMA was chosen to be implemented. The EMA for a single sample is computed as follows:

$$out_n = \alpha * input_n + (1-\alpha) * out_{n-1} \tag{2.17}$$

$$\alpha = \frac{2}{(N+1)} \tag{2.18}$$

With α being defined based on the filter order N, thus can be computed *a priori*. Setting N to a high value can result in a loss of data, meanwhile setting it to a low can result in an inaccurate output value. The exponential behavior corresponds to the $(1-\alpha)$ factor which is multiplied by the previous output out_{n-1}, differently from the moving average that considers the previous input values instead. Typical EMA responses for a noisy input signal and different N values are shown in Figure 13.

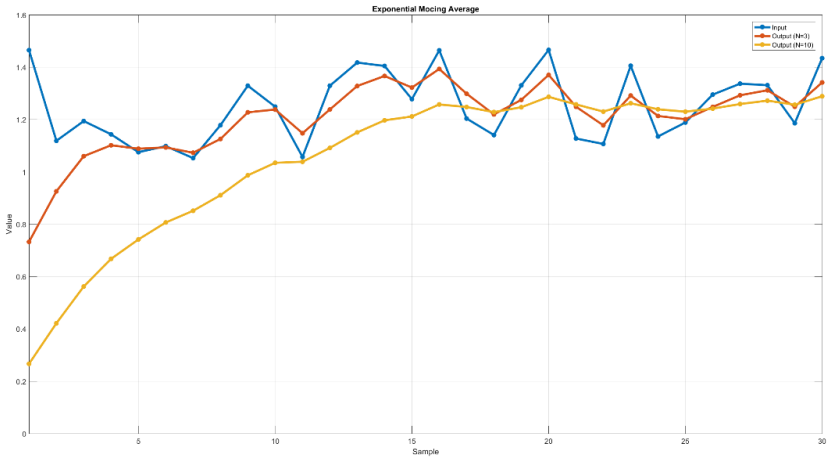


Figure 13. EMA sample responses for different N values

With respect to this application, the best results for the accelerometer readings were obtained with $N = 3$ during testing, hence (2.17) becomes:

$$filtered\ acc_n = 0.5 * (acc_n + filtered\ acc_{n-1}) \tag{2.19}$$

Next step is to implement a window-filter to reduce mechanical noise, as minor errors in acceleration could be interpreted as a constant velocity and will be summed [26]. Moreover, the low-pass filter might produce residual erroneous data, so a window of discrimination between valid and invalid data and for the no-movement condition must be implemented [26]. Implementation is straight forward: if the input is within the maximum and minimum window values, the output is 0; otherwise, the output is the input. A sample behavior of the window-filter is depicted in Figure 14 below, where the maximum and minimum window values are set to ±0.3, respectively:

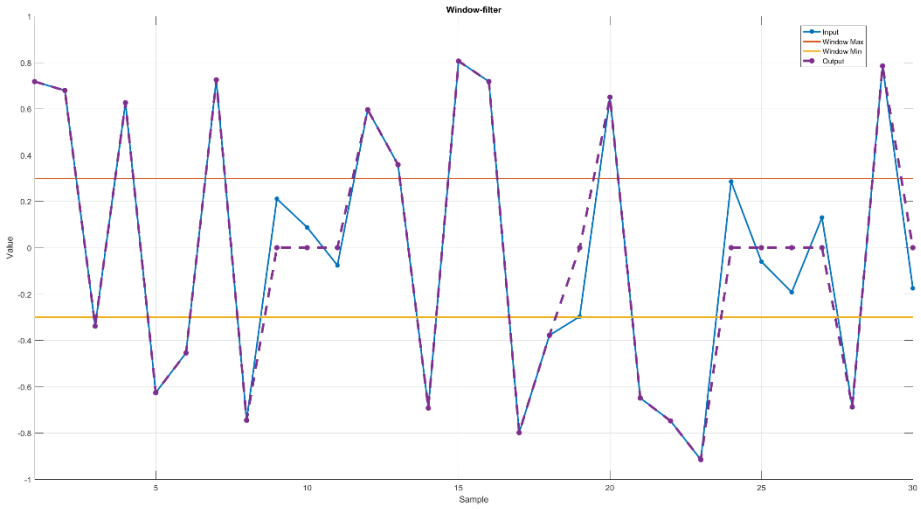


Figure 14. Window-filter sample response

The window values for the accelerometer were derived in practice, by simply leaving the robotic system in a static position without disturbances and observing the accelerometer readings, which were typically within ±0.02 m/s^2.

With the accelerometer readings properly pre-processed with both low-pass and window filters, the linear speed can finally be estimated through an integration method. A common approach is to utilize the 1st order trapezoidal rule, that approximates an integral by accumulating the area of several trapezoids, as shown in Figure 15 below.

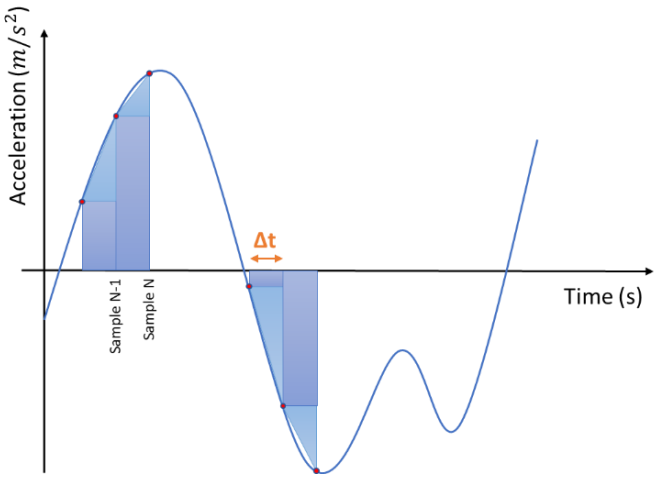


Figure 15. Illustration of the trapezoidal method

Formally, the trapezoidal rule is defined for continuous time as follows:

$$\int_a^b f(x)\,dx \approx (b-a) * \left(\frac{f(a)-f(b)}{2}\right) \tag{2.20}$$

For discrete time and considering the application, however, (2.20) reduces to the following formula which is derived from both the trapezoidal rule combined with (2.16):

$$v_n = v_{n-1} + \frac{(acc_n + acc_{n-1}) * \Delta t}{2} \tag{2.21}$$

With v_n being the speed estimation, acc_n the linear acceleration, and Δt the sampling time for sample n. Not only does the trapezoidal rule greatly reduce integration error, but it has a simple implementation and requires a single floating-point multiplication, hence this method was chosen for speed estimation.

Although the integration error is small, the speed estimation drifts over time due to error accumulation. This issue will be further suppressed by the complementary filter, but can be treated with a post-processing movement-end check, which forces the speed to zero, thus resets the integration method and consequently the error. When moving the robotic system, there is typically an initial acceleration or deceleration until a maximum velocity is reached, before the acceleration changes direction (i.e. signal) until it reaches zero once more [26]. Considering that the velocity is the result of the area below the acceleration curve, the speed will only be zero when both areas above the negative and below the positive side of the curve are the same, which is extremely unlikely to happen in a real-world scenario, consequently making the speed estimation to be unstable. Due to this fact, a movement-end check is performed after the integration process to prevent speed instability: if a certain amount of consecutive accelerometer readings is zero, the speed is set to zero.

After several tests, the maximum quantity of consecutive readings allowed before forcing the speed to zero was set to 25.

Encoder based

Deriving the speed with the encoder is a simpler matter, due the fact the sensor's output signals are captured by the microcontroller, with the library handling the conversion from PPR to RPM, as previously discussed. However, the robotic system uses two encoder motors to move around, and information from both encoders must be properly combined, which result in a 1D speed as required. Thus, encoder-based speed estimation is reduced to standard odometry of the robotic system.

Due the fact there is no interest in performing dead-reckoning, neither heading or position are estimated and using a partial-odometry is enough for the application. Considering the simplistic robotic system structure, a basic model was derived based on [27, 28, 29, 30, 31] in which slippage is not considered [32]. In this scenario, the combined encoder speed v_{enc} is the mean of the encoder readings from both sides:

$$v_{enc} = \frac{v_R + v_L}{2} \tag{2.22}$$

With v_R and v_L being the right and left encoder readings, respectively, in RPM. Alternatively, one might compute the velocity of each encoder motor, in meters per second, based on the wheel's radius R, encoder's resolution PPR, pulse difference $\Delta pulses$ and time between measurements Δt, as follows:

$$v = \frac{2 * \pi * R * \Delta pulses}{PPR * \Delta t} \tag{2.23}$$

Note that if the encoder's resolution and wheel's radius of both sides are the same, the resulting encoder speed can be computed with the following formula, derived from (2.22) and (2.23):

$$v_{enc} = \frac{\pi * R}{PPR * \Delta t} * (\Delta pulses_R + \Delta pulses_L) \tag{2.24}$$

Finally, the combined speed is computed with (22) and converted from RPM to meters per second in order to be fused with the complementary filter. Such conversion is elementary, being only dependent on the wheel's radius R and the track thickness, which were measured and correspond to a combined value 0.032 meters (3.2 centimeters). The following formulas are used for conversion between RPM and m/s:

$$v = \frac{\pi * R_t}{30} * RPM \tag{2.25}$$

$$RPM = \frac{30 * v}{\pi * R_t} \tag{2.26}$$

With $R_t = 0.032\ m$, v the speed in meters per second, and RPM the speed in rotations per minute.

Pitch correction

The gyroscope data might be relevant depending on the terrain conditions in which the robotic system will operate. For this application, it is assumed the robot will function on a flat surface, hence strictly changing its heading. Although the IMU is calibrated upon initialization, when the robot is normally moving, the gyroscope outputs non-zero pitch and roll information with the latter not affecting the 1D speed (forwards or backwards). The pitch, on the other hand, directly influences the resulting speed either estimated with the IMU, or combined with the complementary filter.

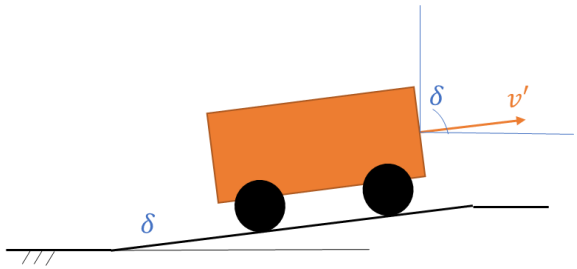


Figure 16. Illustration of the robotic system going up a ramp

Considering a pitch δ as depicted in Figure 16 above, the 1D speed component v with relation to the flat surface must be correct, and is computed as follows:

$$v = v' * \cos(\delta) \tag{2.27}$$

During normal operation, the maximum pitch recorded was 5° (about 0.09 radians). Due the fact the pitch will be, at most, 5°, the small-angle approximation can be used to simplify (2.27):

$$\cos(\delta) \approx 1 - \frac{\delta^2}{2} = 1 - \frac{0.09^2}{2} = 0.996$$

Due the fact the pitch contribution is extremely low, it will be neglected for this project. Note, however, that if the robotic system operates on a non-flat surface, it will present unexpected behavior.

2.4.3 Results

Several minor tests were performed to debug, get used to components, and validate implementations before finally testing the whole robotic system, more specifically the speed estimation and complementary filter algorithms. This section presents the major results obtained during a procedure in which a sequence of different forward and backward speeds are set for a certain period of time, as presented in Table 1 below. In addition to RPM, the ground speed is also presented, which is calculated with (2.25) and will be used for all results in this section.

Table 1. Complete test sequence

RPM	0	40	-50	82	-103	125	-142	167	182
Ground speed (m/s)	0.000	0.134	-0.168	0.275	-0.345	0.419	-0.476	0.560	0.610
Period (s)	1	3	3	2	2	2	1	1	1

Moreover, different results are presented and compared based on the RMSE, which can be computed with (2.1). However, in order to correctly validate the speed estimation procedure (either encoder- or accelerometer-based) and the complementary filter implementations, a ground truth speed should be available. Unfortunately, measuring the speed of a system in an accurate manner is not straightforward, and is mostly performed with expensive equipment or demands an extensive approach.

As measuring the speed was not the main goal of this thesis, an alternative and simpler concept was utilized: encoder readings present typically accurate results, being as low as ±0.04% for a 360 PPR encoder [33]; thus, by fitting the encoder measurements, one might consider such approximation the ground truth, at least currently for this project. Initially, the test sequence was applied and the encoder readings logged, which yielded the results shown in Figure 17 below, with a RMSE of about 0.12 m/s.

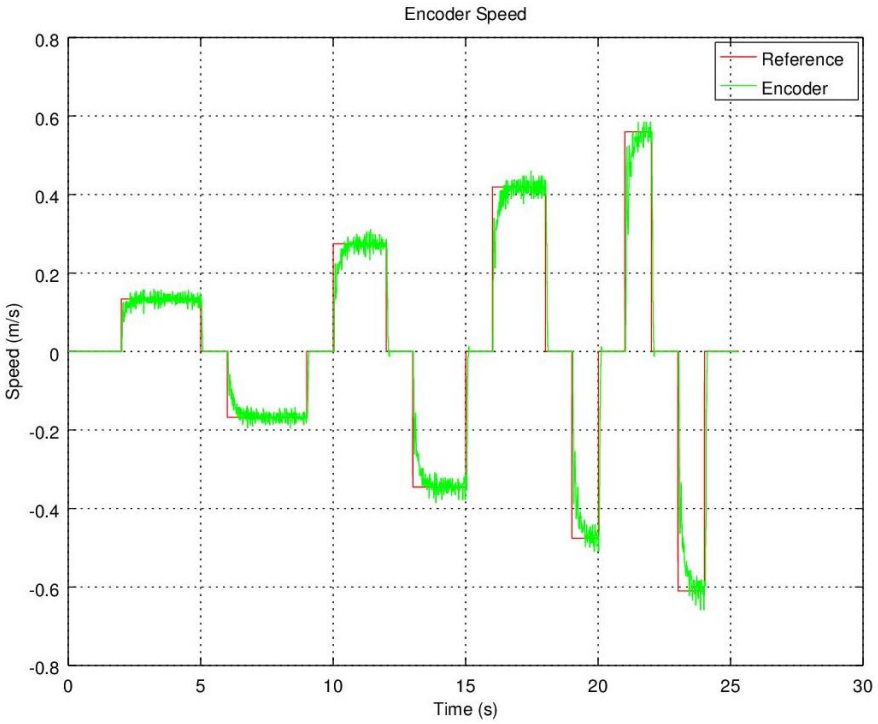


Figure 17. Encoder readings based on the test sequence

It is noticeable by observing Figure 17 above and the speed controller response (Figure 9) that the encoder readings follow an exponential curve. To address the ground truth issue, then, the reference values used for the test are fed to an offline EMA algorithm, in order to fit the encoder readings. However, the EMA implementation, as previously discussed, requires a specific number of samples N to be defined *a priori*. The number of samples N was varied from 0 to 100, and the RMSE computed for the exponential moving averaged (EMAd) reference and the encoder readings to select a proper N which produced the smallest RMSE as possible. The result of this procedure is depicted in Figure 18 below:

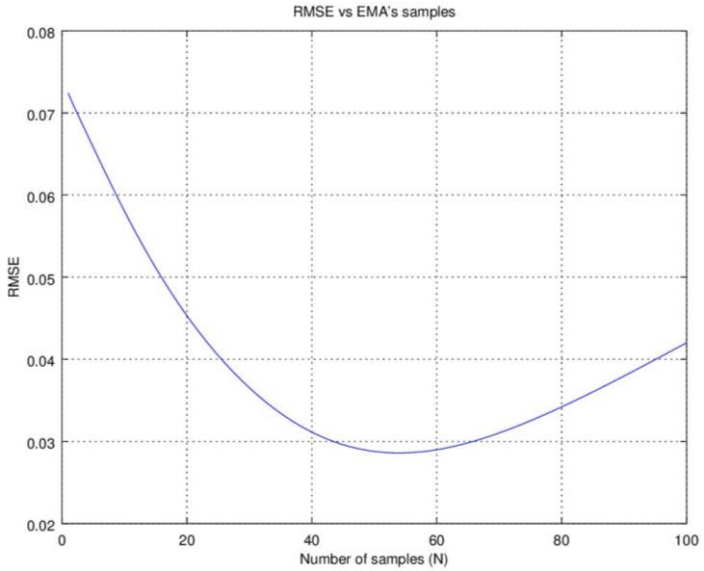


Figure 18. RMSE versus number of samples N

The lowest RMSE value obtained was about 0.03 m/s, for $N = 54$ ($\alpha \cong 0.036$). Furthermore, the EMAd reference for this specific number of samples is shown in Figure 19 below. Note the encoder readings fairly fit, thus from now on in this section, reference will correspond to the EMAd reference and is considered as the ground truth speed for all RMSE calculations. It is relevant to notice, however, that this EMA is a post-processing technique, being implemented offline exclusively.

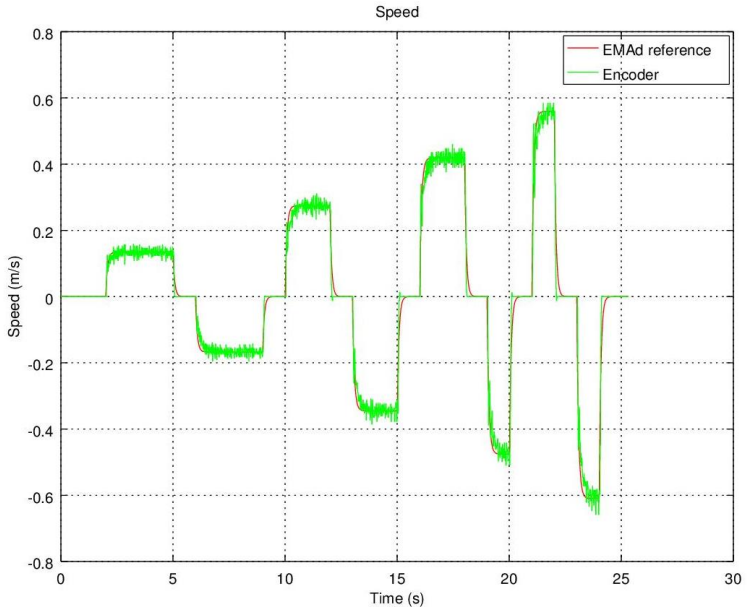


Figure 19. Exponential moving averaged reference for N=54, alongside the encoder readings

Subsequently, the accelerometer-based speed estimation was assessed with the same test sequence, with the results depicted in Figure 20. Note only the speed on the axis of interested is presented, and neither the reference was not modified nor RMSE computed for this graph

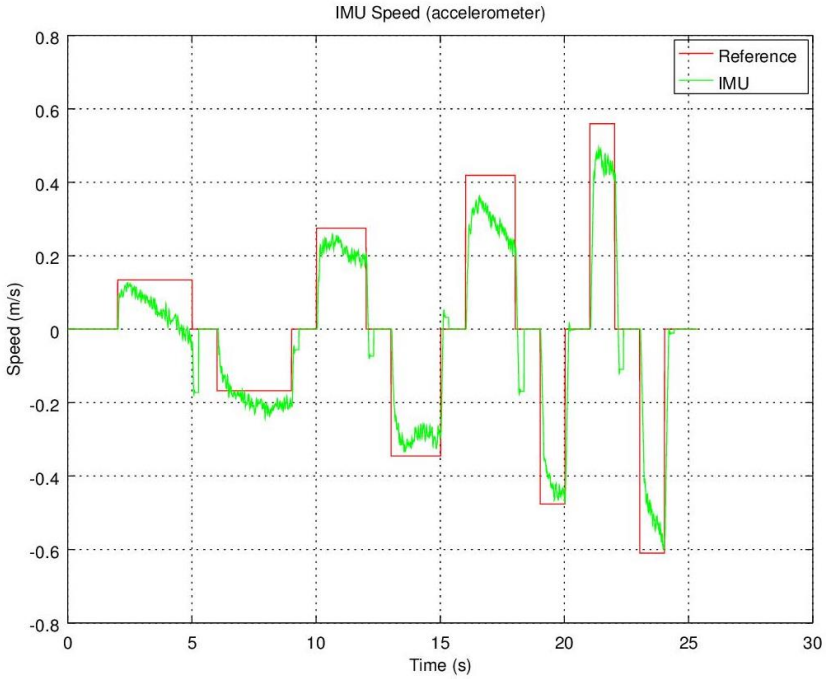


Figure 20. Accelerometer readings based on the test sequence

Moreover, it is relevant to notice the movement-end check impact on the estimated speed, which forces the output to zero only after twenty-five (25) consecutive zero readings, clearly visible when the reference is set to zero. Moreover, notice that estimation procedure does drift over time, which is evident for the first two iterations and the fifth, due the fact their longer duration, when this effect becomes more visible. Finally, it is clear the speed estimation is less accurate than the encoder readings, but outputs data three times faster (150 Hz) with the current data processing required by the speed estimation algorithm.

Complementary Filter

The IMU and encoder data are fused through a complementary filter, in order to reduce the overall noise of the derived speed before forwarding such information to the Pre-Kalman filter and, subsequently, to the Kalman filter itself, which is further explored in the next chapter. Both 1st and 2nd order complementary filters were implemented offline to simplify the procedure of tuning their respective gains. Additionally, considering the complementary filter implementation consists basically of hard-coding equations (2.4) for the 1st order or (2.7-2.11) for the 2nd order, the bottleneck in terms of how fast the robotic system can provide a filtered output is not influenced by the additional operations. Thus, the offline implementation for tuning the gains should present the same behavior as the real one, given a 150Hz frequency is respected.

Initially, the 1st order complementary filter was tested by varying the gain K from 0 to 1, which corresponds to not considering or only considering the accelerometer speed estimation, respectively. For each gain, RMSE is computed based on the EMAd reference value and the filter's output, which yielded the following graph:

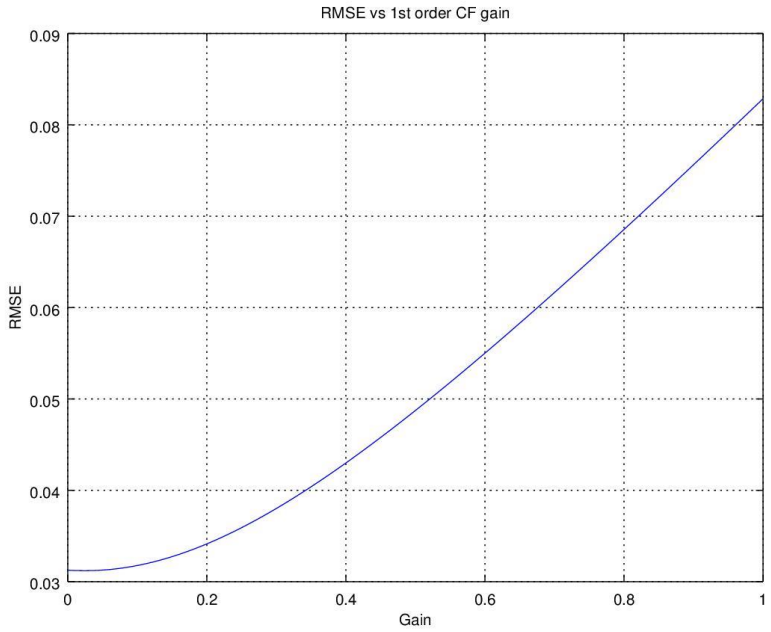


Figure 21. RMSE versus 1st order CF gain

The smallest and consequently the best RMSE obtained was 0.0312 m/s, which corresponds to a gain of $K = 0.02$. Note that this value represents the filter outputs reliable speeds when the accelerometer speed estimation is given a 2% weight, meanwhile the encoder readings 98% weight. With the gain properly defined, the complementary filter was implemented and the whole test performed once more. The results were logged and plotted, as shown in Figure 22 below. Note both the reference and the encoder values are hardly visible, with the filter smoothly fusing the data and outputting a more stable speed.

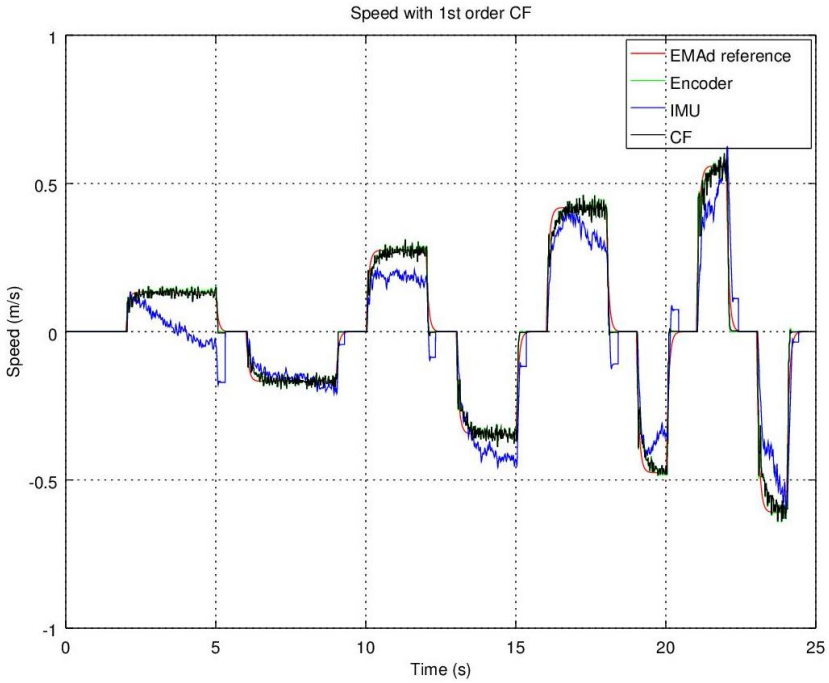


Figure 22. 1st order complementary filter results

Although the 1st order filter presented fairly good results, the 2nd order complementary filter was also implemented in order to check how a higher order structure would behave and decide whether its

more complex structure is worth to be chosen instead of the 1st order filter. The procedure applied to the 1st order filter is repeated, with the gain K being varied from 0 to 20 and each respective RMSE calculated. The results are depicted in Figure 23 below:

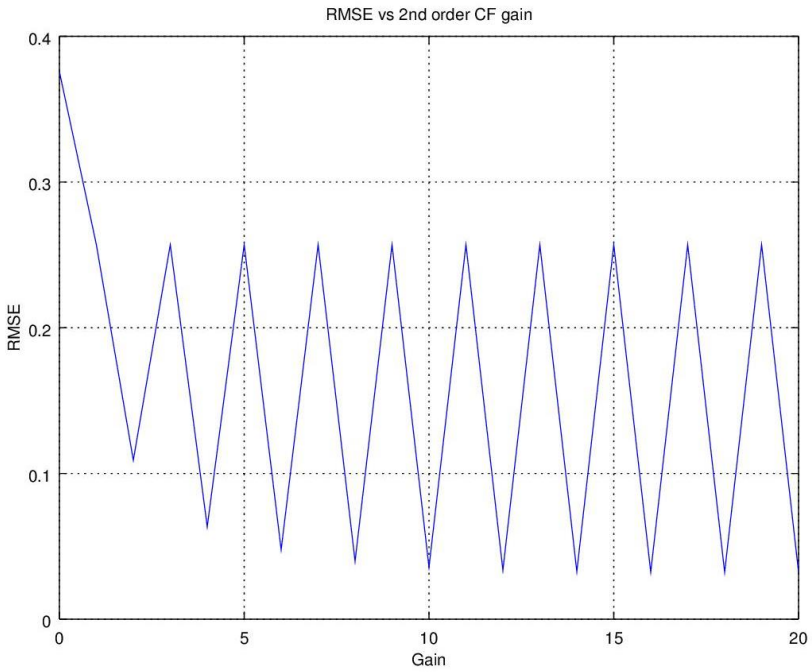


Figure 23. RMSE versus 2nd order CF gain

Based on Figure 23 above, it is evident the 2nd order CF is rather unstable for this system in terms of RMSE. However, in order to fully check its behavior, the filter was implemented and tested with $K = 18$, which corresponds to the best RMSE obtained (0.0322 m/s) – note that based on the RMSE, it is possible to conclude this implementation is worse than the 1st order one, that presented a smaller value: 0.0312m/s. The logged data was plotted and is shown in Figure 24 below. Note, however, that such implementation behaves properly and its output is similar to the 1st order, except when the speed goes to 0 – in which case the output presents an overshoot.

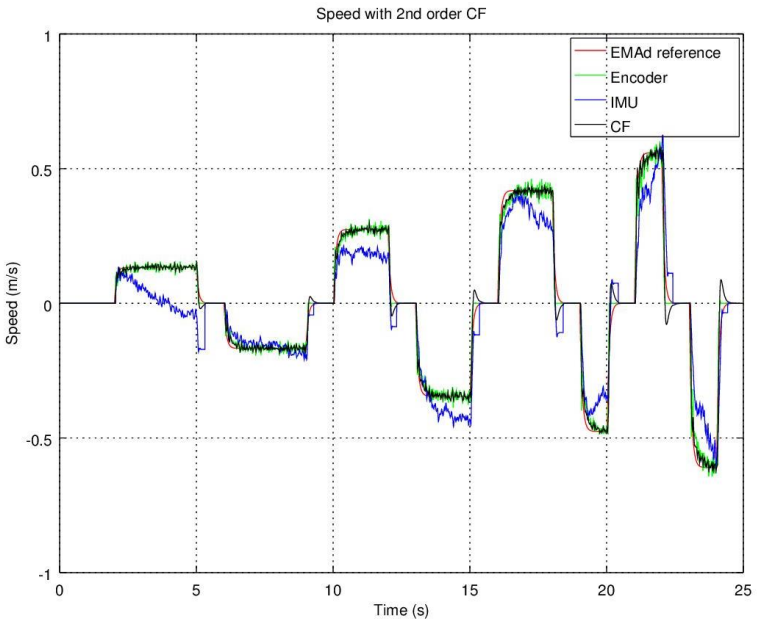


Figure 24. 2nd order complementary filter results

Comparison

In summary, both 1st and 2nd order complementary filter implementations had similar performances, with the former slightly outperforming the latter. However, the 2nd order version requires extra memory, is slightly harder to implement, and presents an undesirable behavior (i.e. overshoot) which may further influence the Kalman filter's behavior. Due to these facts, the 1st order complementary filter was chosen to be used. Another interesting point is that although the RMSE of the 1st order version is higher than the pure encoder readings, the output of the filter itself has a smaller standard deviation (0.25699 in comparison to 0.24778), thus is more stable. Figure 25 compares both filter versions' implementations, for the whole test sequence.

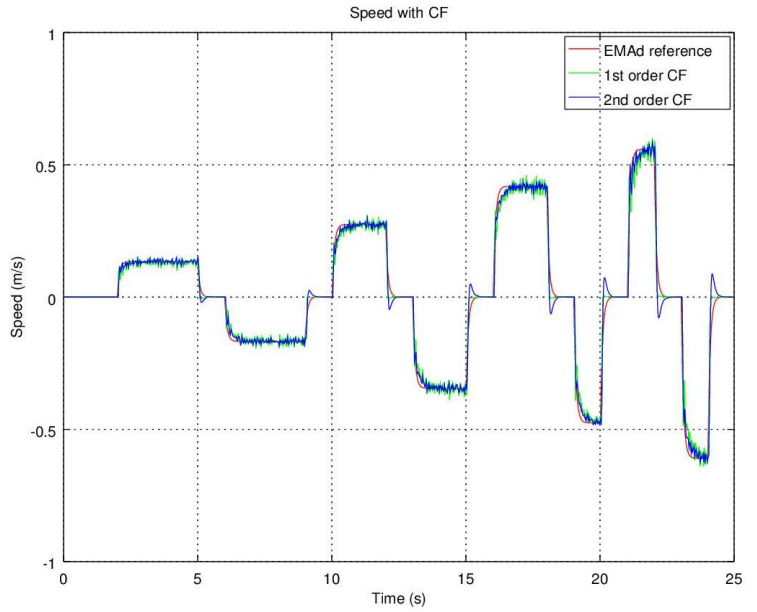


Figure 25. 1st and 2nd order complementary filter results

Not only does the complementary filter reduce the overall standard deviation, but it also improves the sampling frequency of the speed. Previously, the speed could only be updated once every 20 milliseconds (50 Hz) due to the encoder limitation. By combining the encoder readings with the accelerometer speed estimation, one can poll at a much faster rate: every 6 milliseconds (166 Hz), about 3.3x faster. It is advisable, however, to retrieve data at a lower rate due the fact the system might be late when asynchronous events (i.e. references update) triggered by a master (i.e. tracking system) must be processed. In practice, the refresh rate used corresponds to 140 Hz.

Finally, the robotic system comprises the implementation of this technique due the fact it is considered the low-level control system. Hence, it is responsible for driving the motors, providing the current (filtered) speed, and additionally the gyroscope data required by the Pre-Kalman filter.

2.5 Communication protocol

Communication between the robotic system and a master (e.g. tracking system), more specifically the MegaPi board – PC (or ideally Zynq) data exchange must respect guidelines in order to proper send or receive data. Thus, a (micro)protocol under certain bus configurations was implemented, and will be addressed in this section. Such protocol is serial-based and implements data transfer with the Most Significant Byte being sent first (MSB first), and neither CRC nor ACK are implemented, to keep it as simple and fast as possible.

Data transfer is implemented through an UART bus, with both devices configured as follows:

- Baud rate: 5000000 bps (62.5 kbps) – advisable to increase it in the future
- Word length: 8 bits
- Stop bit(s): 1
- Parity: None
- Flow control: None

The only and essential communication between the robotic and tracking systems is full-duplex, with both systems being able to send and receive data. Currently, two (2) message packets are supported, one for retrieving sensor data and another for setting new speed references. These packets are composed, generally, by a synchronization word (2 bytes), a command byte (1 byte), and data bytes. The former is used for synchronization purposes, the command is dedicated to specifying the requested functionality, followed by data bytes when applicable.

The generic data packet for requesting sensor data from the robotic system and its respective response are shown in Figure 26. On the other hand, setting speed references does not require any response from the robotic system, and its general packet is depicted in Figure 27.

Master requesting data from robotic system

	Sync Word		Cmd
	0xAA	0x55	0x00
Byte order:	**1**	**2**	**3**

Response from robotic system

	Sync Word		Cmd	Current speed (RPM)								
	0xAA	0x55	0x00	(float32_t) speed								
Byte order:	**1**	**2**	**3**	**4**	**5**	**6**	**7**					

	ω_{yaw} (deg/s)				ω_{pitch} (deg/s)				ω_{roll} (deg/s)			
	(float32_t) yaw				(float32_t) pitch				(float32_t) roll			
Byte order:	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

Figure 26. Master (tracking system) – Robotic system sensor data request (top) and response (bottom) data packets

Master sending speed references to robotic system

	Sync Word		Cmd	Right RPM				Left RPM			
	0xAA	0x55	0x01	(float32_t) rpm_r				(float32_t) rpm_l			
Byte order:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Figure 27. Master (tracking system) – Robotic system references data packet

Requesting sensor data corresponds to sending three bytes, the sync word and the corresponding command (0x00). The robotic system replies with: current speed, yaw, pitch and roll (4 bytes each). A synchronization word is also added in the beginning of such message; thus, the total number of bytes sums up to 19 bytes for the response. Setting speed references requires 11 bytes in total to be sent: sync word, command (0x01), followed by the right and left reference speeds (4 bytes each). Note that all speed values are in RPM so data is platform-independent and a simple conversion is required to be done in another platform. Furthermore, positive and negative references correspond to moving forwards and backwards, respectively. Gyroscope data, on the other hand, is pre-processed (i.e. converted to °/s) due to the fact its sensitivity and other device-specific configurations should not be relevant to the master.

Note that floating-point values are sent via serial, and in order to do so a *union* structure is used for both sending and retrieving these numbers. Check Appendix C: Embedded Software for an example on how to implement such technique.

2.6 Final Implementation

With each component of the robotic system properly defined and tested, it is possible to merge them to compose the whole final implementation. This section presents an overview of the complete robotic system in terms of structure, hardware and software, with the setup itself being depicted in Figure 28 below.

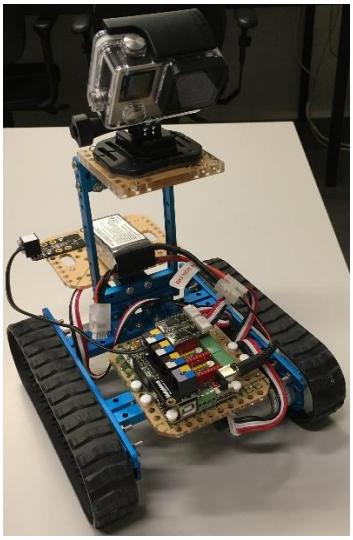


Figure 28. Robotic system final setup

In addition to the components previously discussed, a camera is attached to the robotic system, as shown in Figure 28 above. Note, however, that such device is only placed together with this system, but it is part of the tracking system which will be discussed in the next chapter. Moreover, the current setup does not comprise a dedicated board (i.e. Zynq) that would perform the high-level operations related to the camera, due the fact the tracking system is running on a PC that connects to the robotic system with an USB cable. Ideally, the PC should be replaced by another board and the communication between systems realized with direct UART connections – Tx, Rx, VCC and GND.

2.6.1 Hardware

With respect to hardware, the robotic system comprises two (2) encoder motors, their respective drivers, the MegaPi board, a RJ25 shield to interface with the IMU module (accelerometer and gyroscope), and a Bluetooth module which is currently unused. Additionally, it has an UART connection to an external tracking system, which makes uses of the µprotocol discussed in section 2.5. Finally, a 3-cell Li-Po battery is used to power the whole system (ideally the tracking system as well), capable of providing 14.4Wh (11.1V @ 1.3A) and thus enough power to all modules and more importantly, both encoder motors. The overview of the robotic system is depicted in Figure 29 below. Although it is not shown, the IMU module is connected to slot eight (8) of the RJ25 shield, the right and left encoder motors (and drivers) are attached to PORT1 and PORT2, respectively. Bluetooth utilizes UART3, and communication with the other system uses UART1.

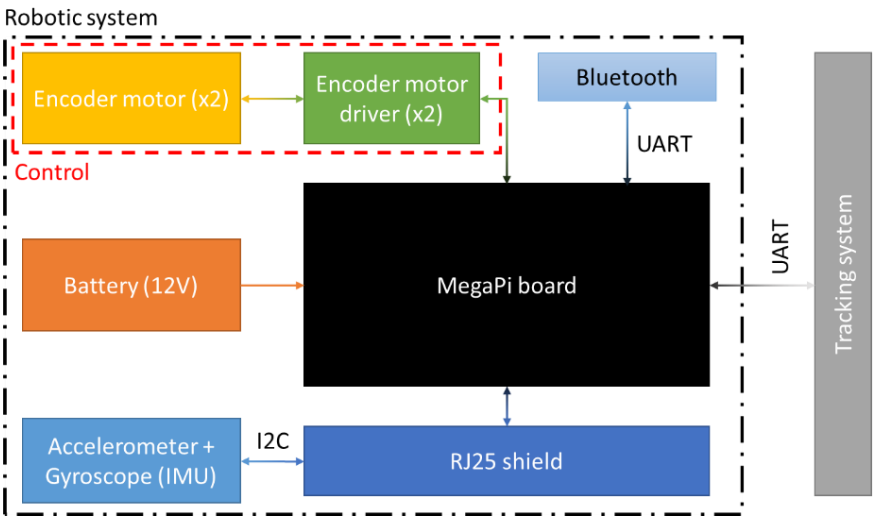


Figure 29. Final hardware architecture of the robotic system

Notice the IMU module was placed on the same platform as the battery (check Figure 28), which defines the reference axis for the module's data: the positive X axis points to the front, positive Y to the left and positive Z to the top of the robot, respectively. This configuration directly impacts the data signaling retrieved from the module, and was taken into consideration during development. Based on the module's orientation, heading (or yaw), pitch and roll are rotation around the XY, XZ and YZ planes, respectively, with the gyroscope and accelerometer data correspondence being summarized in Table 2 below.

Table 2. Movement type, axis, and signaling correspondence for the IMU module

Movement	Axis		Signal meaning	
	Accelerometer	Gyroscope	Positive	Negative
Heading	-	Z	Turning left	Turning right
Pitch	-	Y	Going downhill	Going uphill
Roll	-	X	Rolling right	Rolling left
Linear	X	-	Going forward	Going backward

2.6.2 Software

In terms of software, the robotic system is responsible for retrieving and processing IMU and encoder's data, alongside applying the complementary filter and driving the motors. Moreover, it must comply to the µprotocol discussed and react to asynchronous serial events. It is relevant to notice that the software runs in a single-core microcontroller, in a sequential manner, with the Arduino IDE being the development tool used. The first step is initializing peripherals, as shown in Figure 30 below:

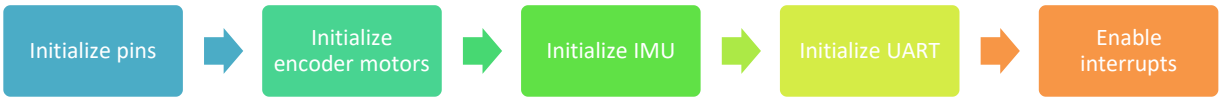


Figure 30. Robotic system's initialization procedure

Initialization is rather simple, with relevant I/O pins begin initialized first, followed by the encoder motors which requires setting up related timers for PWM and encoders' interrupts. Then, the IMU

module is calibrated and initialized, alongside the UART used for communication (or debugging), and finally all interrupts are enabled. After this step, the main loop runs, with a single execution depicted in Figure 31 below:

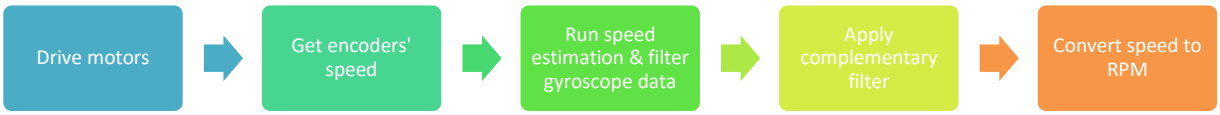


Figure 31. Robotic system's single loop execution sequence

Driving the motors corresponds to setting the latest speed references to the PID controllers. Next, the encoder readings are retrieved and converted to m/s, followed by the speed estimation procedure based on the (filtered) accelerometer readings. The complementary filter is then applied to both speeds derived, and finally internally stored in RPM. Although filtering the gyroscope was not previously discussed, another instance of the EMA was implemented specifically for noise reduction, as such information is forwarded to the tracking system. Moreover, the filter gain chosen during testing and for the final implementation was $\alpha \cong 0.33$, which corresponds to $N = 5$ samples. Similarly to the speed estimation procedure applied to the accelerometer readings, the gyroscope data should be fed to a window-filter, with a maximum and minimum window values equal to ±0.2 °/s, respectively, which were derived in practice.

Asynchronous serial events triggered by a master may happen at any moment, and should be handled properly. This is implemented within a receiver (RX) interrupt routine, that reflects the behavior of the state-machine depicted in Figure 32.

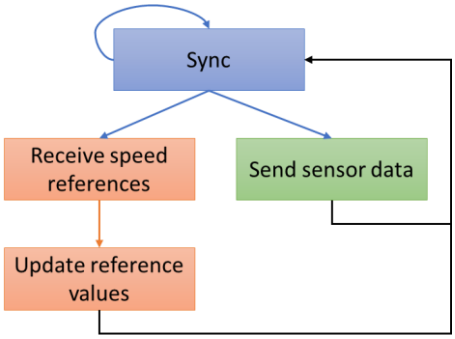


Figure 32. State machine of the serial event handler

Initially, the handler is in the synchronization state (*Sync*), in which at least three (3) bytes must be received in order to switch to the next state and further receiver or send data. If a correct synchronization word (2 bytes) is received, followed by a supported command (1 byte), the handler moves either to the *Receive Speed References* or *Send Sensor Data* states. The latter does not require any other data to be received, thus it immediately sends the requested sensor data to the master. When receiving speed references, it waits for a total of eight (8) bytes and update the internal reference values, which are later forwarded to the PID controllers if and only if the data is valid: values are within ±203.5 RPM, which is the absolute maximum rating of the encoder motors.

To avoid data inconsistency during transmissions, flags are used before and after each step, preventing incorrect data of being sent. However, this requires saving data from the previous loop, thus memory

usage is higher and the algorithm is more complex. Data inconsistency is avoided, in summary, in the following manner: before retrieving data (either encoder's or IMU's), the respective step flag is set to *false*, and the data retrieval is triggered; if by any chance an asynchronous serial event occurs during this procedure, the data that will be sent refers to the previous iteration because the respective flag is checked; finally, after correctly retrieving the data, it is copied to a fail-safe variable, alongside the flag being set to *true* (setting the flag occurs first).

In the beginning of the source code, relevant parameters are defined and commented, such as:

- Micro protocol configurations: baud rate, synchronization word and total amount of bytes to be sent or received;
- Gyroscope and accelerometer data configurations: bias (window-filter), index of the relevant axes, sign correction (accelerometer only) and EMA filter parameters;
- Speed estimation configurations: gravity value, robot's wheel (and track) total radius, and movement-end check maximum counter;
- Complementary Filter configuration: gain.

The high-level libraries used for this implementation are the "MeMegaPi" default library, and modified versions of the "MeEncoderOnBoard" and "MeGyro". Refer to Appendix C: Embedded Software for more details.

3. Tracking system

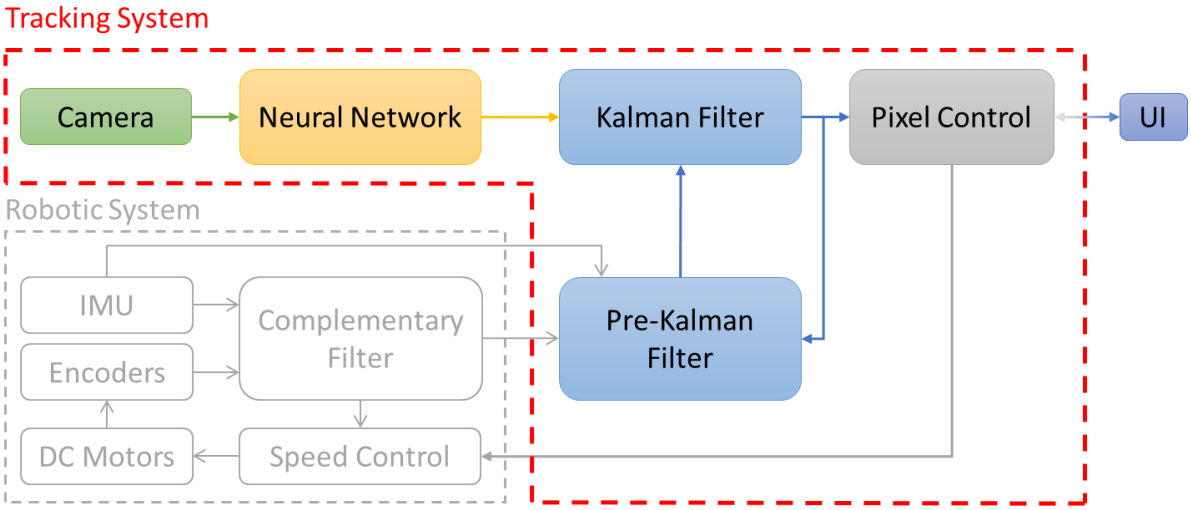


Figure 33. Tracking system outline

With the robotic system correctly implemented and validated, the tracking system is the next one to be assessed. This is the core system of this project, and is responsible for maneuvering the robotic system in order to maintain the target object's position with respect to the image (i.e. frame from the camera) coordinates outputted by a Neural Network. Furthermore, the target object's coordinates are fed to a Kalman filter, which addresses the current flaws of the neural network: (1) deviation with respect to the output position, (2) false-negatives, and (3) occlusion scenarios. The smoothed position coordinates are then forwarded to the controller which handles the speed and direction of the robotic system. Figure 33 above depicts the most important blocks of the tracking system, which are detailed within this chapter. Although both camera and Neural Network are not the focus of this project, due the fact it was part of another colleague's thesis, they are briefly discussed.

Due to limitations on the Neural Network's performance, which typically achieves 90% of accuracy during the training phase in most scenarios, and the localization algorithm which roughly estimates the coordinates (i.e. position) of the object in the image, a filtering technique is required to smooth the position output. Consequently, controlling where the object is in the image, thus indirectly controlling the distance the object is in relation to the robotic system, becomes more stable. Data coming from the robotic system (angular velocities and linear speed) is processed by the "Pre-Kalman Filter", which is responsible for estimating how the ego-motion of the robot impacts the movement of the object (being tracked) in the image. The ego-motion itself refers to how the robotic setup is moving in terms of linear speed (translation, in m/s) and angular velocity w.r.t. heading (rotation, in °/s). Its impact on the tracking procedure, on the other hand, is analogous to estimating both translation and rotation movements in terms of pixels per seconds (px/s). Moreover, the latter is denoted as "pixel speed" throughout this thesis, and corresponds to the movement of both coordinates (X and Y) of the target which are derived by the neural network. The pixel speeds (X and Y axes) are finally forwarded to the Kalman filter, responsible for filtering the neural network's output coordinates. Lastly, the filtered object coordinates are fed to the "Pixel Control" module, which controls the speed and direction of the robotic system in order to maintain the target object in a reference position.

Finally, a set of requirements and functionalities was defined for the tracking system, which must:

- Interface with a camera, and programmatically access a frame;
- Be compatible with a Linux-based platform;
- Be able to execute a (simple) Neural Network model, ideally in an accelerator;
- Provide stable position information of the target object;
- Use ego-motion data to improve the position information, whenever possible;
- Maintain the target object close to a reference position, with respect to the image;
- Send speed references to the robotic system;
- Retrieve sensor data from the robotic system: current speed and angular velocities;
- Execute all tasks faster than the frame rate provided by the camera;
- Provide a simple user interface.

3.1 Camera

Although sensor data is retrieved from the robotic system and used in the tracking system, the camera is considered the main sensor of this application. Not only does it usually represent a bottleneck due to its maximum frame rate, but also must be able to stream to an embedded platform. The latter is more problematic, due the fact most cameras are not capable of streaming at all. Moreover, handling a frame is different in hardware (i.e. FPGA) and software (i.e. microcontroller), thus even when streaming is a possibility, one must consider the interface. The most relevant camera interfaces used for streaming are: (1) USB, (2) HDMI output, (3) Wi-Fi and (4) direct I/O connections. The former requires a driver, do not meet real-time requirements, and might not meet the framerate, especially due the fact most USB cameras are webcams, thus framerate is not relevant as it is for this application. On the other hand, cameras with HDMI output require a more complex driver and are rather expensive because it is considered a “professional” feature, except for action cameras. Wi-Fi streaming is fairly common, especially for action cameras, but a delay is often present. Finally, interfacing a camera directly via its I/O pins present an even harder driver implementation, in addition to a low availability on the market.

Figure 34. GoPro Hero4 Black mounted on top of the robotic system

Considering all the aforementioned factors, it was decided to use a camera with both Wi-Fi and HDMI output options for streaming, with a reasonable price: the GoPro Hero4 Black [34]. Moreover, this camera can deliver up-to 240 frames per second (fps) at 720p resolution (1280x720 pixels), which is more than enough for the application, due the fact the bottleneck of the tracking system is the neural

network, which currently runs at 20 Hz (i.e. 20 fps) in a PC – this will be further detailed in this chapter. As this camera is considered an "action camera", it could be easily placed on the robotic system, and is shown in Figure 34 above inside a waterproof case.
Among other options, the user can choose the field of view of the camera for the specified resolution: Ultra-wide, Medium or Narrow. The former presents substantial lens distortion, and the latter is mostly used when a high frame rate is required. As distortions might impact the ego-motion translation to pixel motion, and 240 fps is rather excessive, the camera was configured to operate on Video Mode, with 720p resolution at 60 fps and Medium field of view.

Streaming can be done by either using a mini HDMI cable to connect the camera to a monitor, or another hardware with HDMI input circuitry, or by connecting a Wi-Fi device to the camera's access point. Initially, a platform with HDMI input capabilities were meant to be used, however the current implementation uses the Wi-Fi access point for streaming, due to its simplicity. There is a typical 0.5 seconds delay when streaming over Wi-Fi which cannot be avoided, although it could be removed if the HDMI option is used instead. Moreover, the stream resolution over Wi-Fi is restrained to 640x480 (480p), although the framerate remains unchanged (60 fps).

Accessing the frame programmatically when streaming over Wi-Fi, however, requires reverse engineering due the fact the protocol used is proprietary: in summary, a HTTP request is sent to the camera to begin the stream, and keep-alive packets must be sent periodically every 2.5 seconds to maintain it. The frames themselves can be accessed in different ways, but saving each image to a file and replacing it whenever a newer frame arrives proved to be sufficient, and more importantly, uncomplicated to implement. Due to the complexity of the protocol and the upcoming neural network algorithm, handling the stream and running the neural network itself were both implemented in a Python application, which is Linux compatible. Moreover, this piece of software launches a *ffmpeg* [35] process, which saves the stream frames. For more details on the implementation, refer to the last section of this chapter and Appendix C: Embedded Software.

3.2 Neural Network

Although the neural network is part of another colleague's work, it is necessary to briefly discuss it, in terms of expected inputs and outputs, in addition to timing. Generally, a convolutional neural network (CNN) is composed of layers with neurons that have learnable weights and biases, and, in a simplistic manner, filter the input data through convolutions, based on the weights of the neurons. Training is an essential procedure of a neural network implementation, in which several inputs are provided and the weight of each neuron is tuned. Several variations of neural network exist, although for this project a CNN is applied in frames from the camera. More specifically, two (2) neural networks are used, one meant for detecting the target (detection neural network), and another for localizing the target within the input image (localization neural network), which in the training phase (best-case scenario) was able to achieve around 90% accuracy. Both are denoted as Neural Network module, and its overall functionality is shown below:

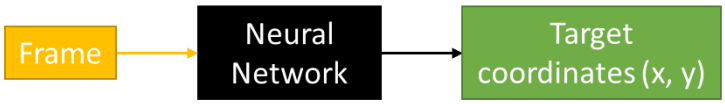


Figure 35. Neural network functionality overview

Whenever a frame is available to be processed, it is fed to the neural network. Initially, the detection neural network is applied, and whether the target is detected, valid coordinates (i.e. within the image boundaries) are outputted. If the detection fails, negative coordinates are provided as output. Specifically, the goal of the application is to follow the robot depicted in Figure 36, currently only from behind, hence the neural network was pre-trained with several images of the target object in different scenarios.

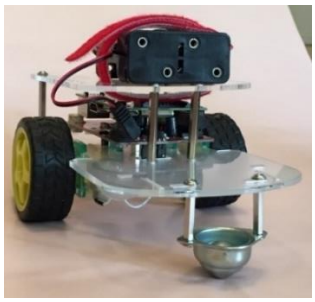


Figure 36. Target object

Another relevant factor is the frame resolution, in pixels, due the fact the training procedure constraints the required input size. For the neural networks used in this project, it is necessary to resize the incoming frame with a 480p resolution down to 360x240 and 224x224 pixels for the localization and detection layers, respectively. Although the initial goal was to have the neural network module implemented in hardware (i.e. FPGA), the current implementation is done in software, more specifically in Python, with the TensorFlow [50] library. During testing, the worst-case execution time of both neural networks observed was 40 milliseconds (25 Hz), thus framerates higher than 25 fps are useless in this scenario. The camera stream, however, was (soft-)limited to 20 fps thus the neural network has extra time (10 milliseconds) to process the frames and should not miss the deadline, except in unusual situations (e.g. high CPU load, cache problems). Notice that this limitation on the neural network module actually corresponds to the bottleneck of the system.

Finally, it is important to note the TensorFlow library has a GPU variation, which can be used if a GPU is available, decreasing the execution time to about 20 milliseconds (50 Hz) and thus partially addressing the bottleneck, although a higher framerate ($> 60\ fps$) is desired. Figure 37 below depicts a frame captured by the camera which is fed to the neural networks used (on the left), and the target coordinates are marked (on the right). Notice the position of the target does not correspond to the same coordinates for different frames, which is further addressed by the Kalman filter.

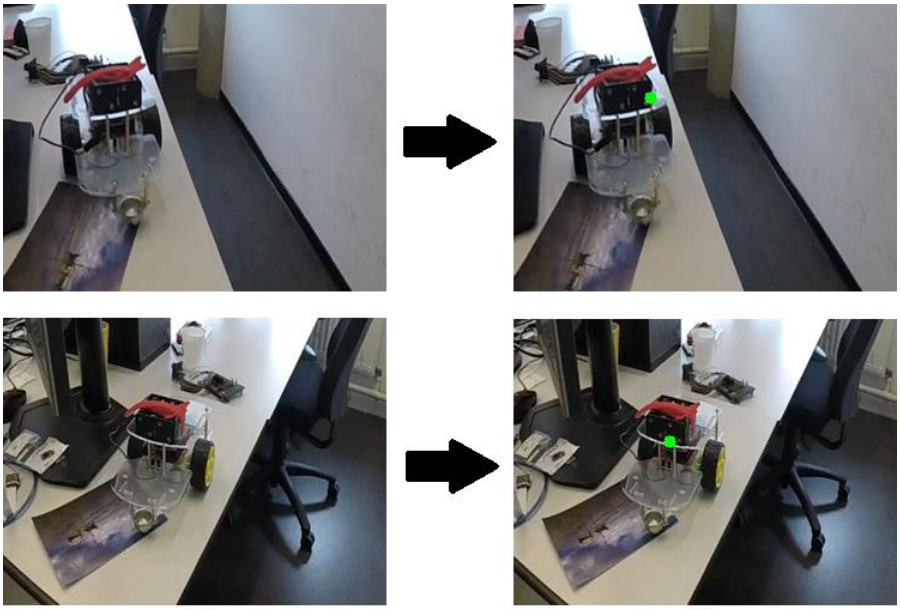


Figure 37. Sample input (a) and output (b) frames of the neural network

Although one might argue the (upcoming) Kalman filter can be indirectly embedded in the neural network itself, this design requires a more complex structure for the neural network such as a Recurrent Neural Network (RNN). Moreover, the major problem would be obtaining a training set in this scenario, which would need manual and careful annotation.

3.3 Pre-Kalman Filter

Based on the robotic system's output (linear speed and angular velocities) – namely the ego-motion -, a conversion must be performed in order to provide the correct input for the Kalman filter, which will be handling the target's object coordinates in pixels. More specifically, the interest is in how the ego-motion of the robotic system impacts the position of the target in an image, due the fact the Kalman filter derives the object's motion directly in pixels/second. In summary, linear speed (m/s) and angular velocities (°/s) are translated to 2D pixel speeds (px/s): V_x and V_y. The latter corresponds to how the target position is affected by the ego-motion itself, and is defined as "pixel speed".

There are specific techniques to transform real-world speed to pixel motion and vice-versa. However, algorithms such as an online camera external parameters calibration or SLAM [10, 36, 37, 38, 39, 40, 41] requires computationally expensive image processing techniques (i.e. sift key points and RANSAC) combined with a dedicated Kalman Filter, with many relying on the camera calibration procedure. Considering the application will be implemented in an embedded platform, it requires simpler, and more importantly, faster techniques. Thus, the "Pre-Kalman Filter" module is proposed, addressing specifically the computational and implementation complexities issues, and will be detailed in this section.

Initially, it is necessary to introduce the optical flow field of the camera, which relates both translational and rotational motions to pixel movement. More specifically, two types of field are usually discussed in literature: motion- and optical flow fields. The former is the real-world 3D motion of objects, meanwhile the latter is analogous to the projection of the motion field onto the 2D image.

The main idea of the Pre-Kalman filter is to abstract the real motion of the target, and estimate the optical flow field based strictly on the ego-motion data. According to [42] and [43], a pure translational motion of the camera (hence, the robotic system) corresponds to a point in the image moving towards or away from another point, called the focus of expansion (FOE). When the camera is rotating and translating at the same time, the optical flow field can be quite complex: the translational component and hence the optical flow field is cubic or even higher order for curved 3D surfaces, although the rotational component is always quadratic [42].

Furthermore, the optical flow field is a function of 2D pixel coordinates, a rotation matrix (extrinsic parameters), and the camera's focal length (intrinsic parameter). Notice that both intrinsic and extrinsic parameters are required, but not available and quite burdensome to determine on-the-fly, hence the Pre-Kalman filter module can use only the ego-motion data alongside specific coordinates to determine the pixel motion. Figure 38 below depicts the overview of the module, besides its inputs and outputs:

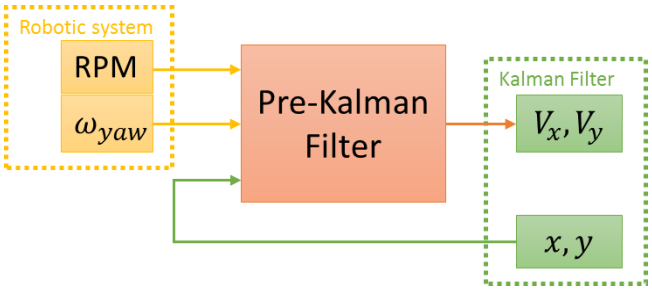


Figure 38. Pre-Kalman Filter overview

The input coordinates, however, must reflect the position of the target object, due the fact the interest in strictly on its movement. Hence, this module is initialized with "invalid" input coordinates, which forces it to be bypassed and is only activated once the following module (i.e. Kalman Filter) provides a valid (filtered) position.

Provided that the output of the neural network is quantized and rather unstable, computing the ego-motion for raw coordinates might negatively impact the overall behavior, thus it is desirable the input coordinates of the Pre-Kalman module are provided by the Kalman Filter itself. In order to correctly translate ego-motion to pixel speeds, both translational and rotational movement impacts are individually analyzed, approximated to 2D or 3D functions based on the available inputs and practical observations, and further combined as it is discussed in the upcoming subsections.

It is relevant to note that the camera must be placed aligned to the center of the robotic system and facing downwards as shown in Figure 28 (section 2.6), due the fact the optical flow view is strictly dependent on the camera's orientation and all approximations might differ. Moreover, the coordinate system used throughout this thesis is defined based on the image representation, with the origin being located on the top-left corner of the image, as depicted in Figure 39 below.

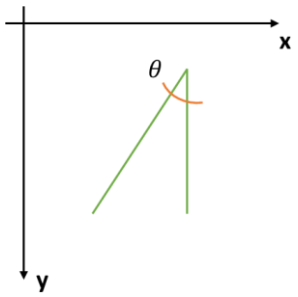


Figure 39. Pixel's coordinate system

Notice that negative pixel coordinates are considered valid for only positive values in-between 0 and the maximum image dimensions (either height or width) and angle θ, denoted as slope in this chapter, is computed with:

$$\theta = \tan^{-1}\left(\left|\frac{\Delta x}{\Delta y}\right|\right) \tag{3.1}$$

Although the image is resized on the final application to 320x240 pixels, it is relevant to note the image resolution used throughout this section corresponds to 640x480 pixels, due the fact analyzing smaller images is harder. However, simply resizing the image corresponds to dividing the final pixel speeds by two (2), in this case, hence there is no need to perform the upcoming steps once more.

3.3.1 Translation impact

Initially, the optical flow field for a pure translation was investigated. In order to illustrate the displacement of a few pixels, a checkboard pattern is positioned in front of the camera, and the robotic system is driven for a short period of time. For a pure translation movement, the initial test is straight-forward: a reference RPM is set to a constant value (+50 RPM), a video is recorded with the camera placed on the top of the robot, which is post-processed so the optical flow can be analyzed. Note that it is easier to evaluate the optical flow when the aforementioned pattern is put in front of the robot, as shown below:

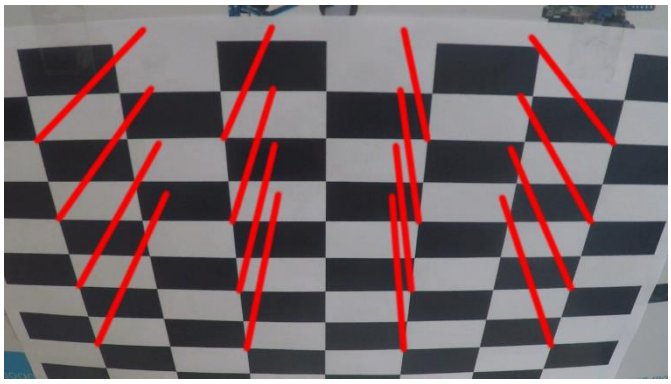


Figure 40. Optical flow field representation for pure translation

After careful consideration on several optical flow fields obtained for different forward and backward speeds, the following points could be derived:

- Pixel displacement is symmetric around the image's center column;

- Defining a slope and base speed is easier than directly deriving pixel speeds, although this implies using at least 2 approximation functions;
- The image should be divided both vertically and horizontally, with more vertical divisions due the fact slopes changes more often;
- The higher the speed, the larger the displacement, this increasing the speed corresponds to scaling a “base speed”;
- In terms of implementation, a polynomial (surface) fitting yields more accurate results in comparison to a look-up table, at a cost of extra computational time.

The modulus of the resulting pixel speed of a specific pixel can be approximated by multiplying a “base speed” value, which corresponds to the pixel speed for the minimum RPM. The corresponding angle, on the other hand, does not need scaling and can be directly derived. Potentially, all pixels of the image can be processed to compute an accurate approximation, however this is burdensome. Thus, one might apply the symmetry characteristic to reduce the overall data processing required, by dividing the image in sub-blocks as shown in Figure 41 below. Although the symmetry property can be applied, it requires a signal correction due to the fact the direction of the pixel speed is, in fact, mirrored with respect to the center of the image (check Figure 40 above).

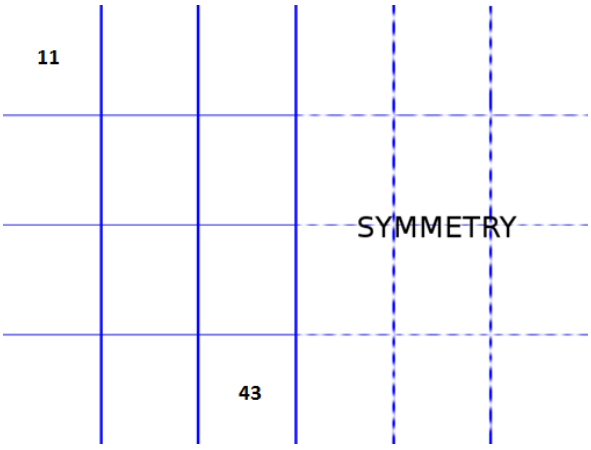


Figure 41. Symmetry grid

Note that each division has a dimension of (W/6 x H/4) pixels, with $W = 640$ and $H = 480$ being the image width and height in pixels, respectively. By indexing the 4x3 sub-block on the left as a matrix, the central pixel coordinates of each division can be computed as follows:

$$P_{m,n} = \big((2n-1) * x_d;\ (2m-1) * y_d\big)$$

$$x_d = \frac{W}{2M} = \frac{320}{M};\ y_d = \frac{H}{2N} = \frac{240}{N}$$

With $P_{m,n}$ being the central pixel coordinates of the mn division, m the row, n the column, M the number of rows (i.e. 6), and N the number of columns (i.e. 4). The number of rows and columns were chosen based on the observed optical flow fields previously observed, although increasing their quantities improves the upcoming approximations. Furthermore, for the given number of divisions and image dimension, the central pixel coordinates are: (53; 60), (160; 60), (267; 60), (53; 180), (160;

180), (267; 180), (53; 300), (160; 300), (267; 300), (53; 420), (160; 420) and (267; 420). The interest is, from now on, only on the left 4x3 sub-division and its corresponding central pixels.

Finally, the pure translation impact on each central pixel can be approximated as follows:

1. For each central pixel
    a. Compute its base speed (minimum RPM), in px/s;
    b. Compute its base angle, in degrees;
    c. Repeat (a-b) 5x, then average the result;
    d. Save central pixel coordinates, average base speed and angle;
2. Compute a polynomial fir for base speed and angle.

The angles are computed with (3.1), meanwhile the speeds as follows:

$$v_{pixel} = \frac{\sqrt{\Delta x^2 + \Delta y^2}}{\Delta t} \tag{3.2}$$

With Δx and Δy being the x- and y-coordinate displacement, in pixels, and Δt the total movement time, in seconds.

Scale Factor: $\gamma(RPM)$

As previously discussed, the base speed should be scaled based on the linear speed. Thus, the scale factor must be correctly estimated. By setting the RPM to a constant value, recording a video, and observing a single pixel, and repeating this for several RPMs, the scale factor function $\gamma(RPM)$ function can be estimated. Such procedure, however, must be performed for a certain RPM range: from 15 up to 203.5 which corresponds to the minimum and maximum RPM, respectively. Moreover, when the minimum RPM is used, the pixel speed is called the "base speed" and the scale factor is determined by:

$$\gamma(RPM) = \frac{v_{pixel}(|RPM|)}{v_{pixel}(|RPM_{min}|)} = \frac{v_{pixel}(|RPM|)}{v_{pixel}(15)} = \frac{v_{pixel}(|RPM|)}{v_{base}} \tag{3.3}$$

Note that there is no reason to compute both pixel speed components (x and y), due the fact the interest is in the resulting (combined) speed, which is calculated with (3.2). The described procedure was performed to a specific initial pixel with a 10 RPM increment between measurements to avoid excessive data. The following graph and speed scale function were obtained:

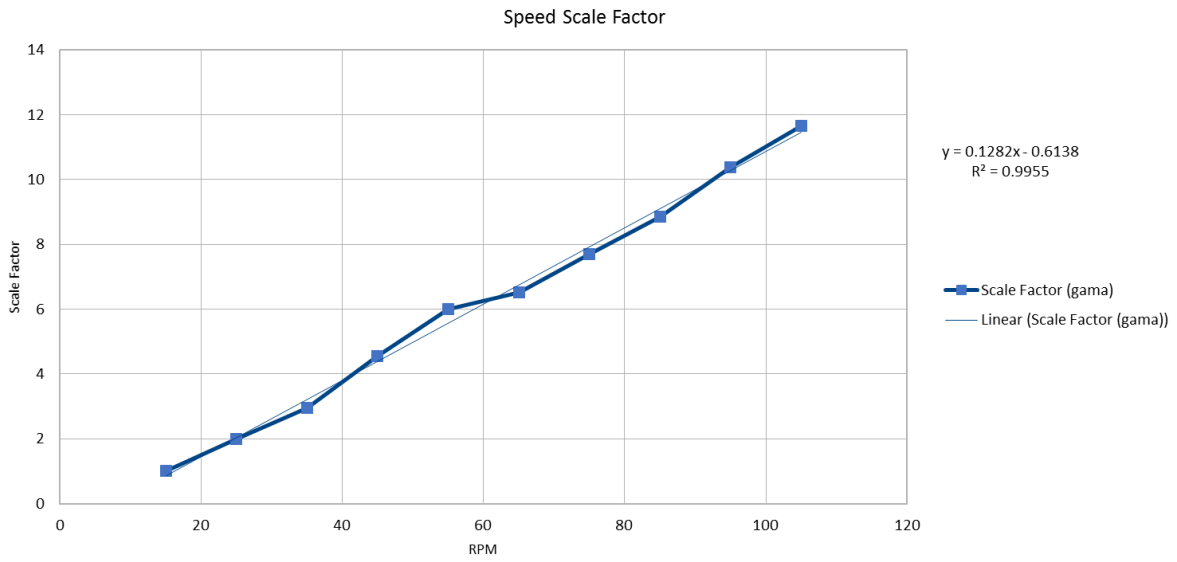


Figure 42. Speed scale factor function for pure translation

Fortunately, the scale factor can easily be approximated by a linear function, as shown above in the graph, and corresponds to:

$$\gamma(RPM) = 0.1282 * |RPM| - 0.6138 \tag{3.4}$$

The standard deviation obtained for this approximation is 1.6 for γ, and about 63 px/s for the pixel speed. The fact the scale factor approximation can be applied to any other pixel was validated by randomly picking other initial pixels, setting a known RPM within the valid range, observing its movement, and applying the above function to estimate both the scale factor, as well as the pixel speed which is computed through:

$$v_{pixel}(RPM) = v_{pixel}(RPM_{min}) * \gamma(RPM) = v_{base} * \gamma(RPM) \tag{3.5}$$

It is important to notice that $\gamma(RPM)$ should only be applied when $|RPM| \geq 5$, preventing unnecessary calculations when the robotic system is static or rotating around an axis. Handling the RPM signal is discussed in the upcoming subsections.

Base speed & angle

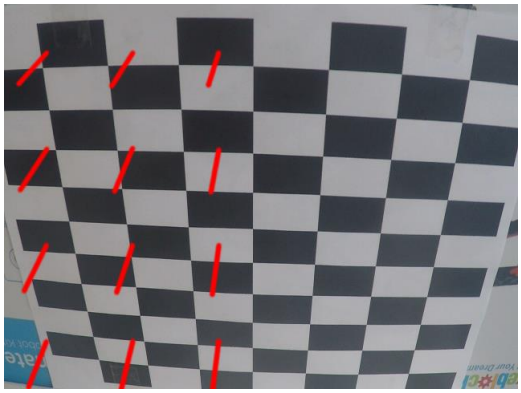


Figure 43. Optical flow field representation for the central pixels, during a pure translational motion

Both the base speed and angle can be derived with the same test procedure used for the speed scale factor γ, but for the minimum RPM (15) only. However, due to symmetry, only half of the image is considered ($x <= W/2$), and more specifically for the 3x4 grid division previously described (Figure 41). The central pixel of each division is then analyzed, with its speed and angle computed 5 times, and stored in a table, then fed to a 3D surface fit function (polyfitn [44]), which computes both base speed $v_{trans}(x, y)$ and base angle $\theta_{trans}(x, y)$ functions. Note that both functions only depend on the pixel coordinates (x, y), and their respective plots are shown below.

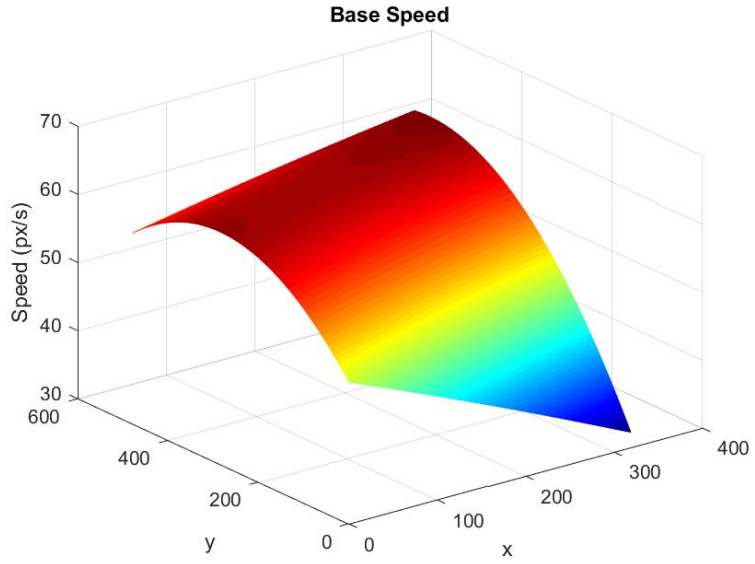


Figure 44. Surface fit for the base speed (pure translation)

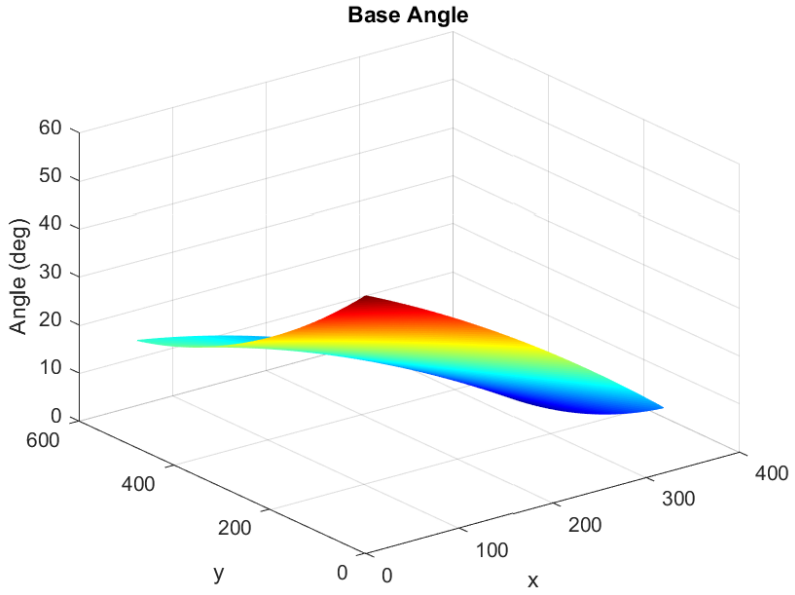


Figure 45. Surface fit for the base angle (pure translation)

Note that in Figure 44, the slowest pixel speed is located around the top middle portion of the image, meanwhile the fastest one around the bottom edges of the image, which matches the observed optical flow field. For the base angle (Figure 45), the surface is slightly more complex, although it is

simple to notice the highest angle is located around the left corner of the image, and the lowest around the center, which also matches the observed optical flow field. It is important to notice the angle is converted to degrees, whenever applicable, because the coefficients of the approximated functions would be $\frac{\pi}{180}$ times smaller otherwise, thus requiring more precision. The obtained polynomial for the base speed and angle, respectively, are:

$$v_{trans}(x,y) \cong -0.000012x^2 + 0.000166xy - 0.0545x - 0.00016y^2 + 0.09347y + 50.65 \quad (3.6)$$

$$\theta_{trans}(x,y) \cong -0.000105x^2 + 0.000124xy - 0.092x + 0.00009y^2 - 0.1068y + 53.62 \quad (3.7)$$

Polynomial order was chosen based on the available data points, required precision and standard deviation from the original values. The standard deviations obtained for the base speed and angle were 0.4414 px/s and 0.898°, respectively.

Resulting function

Both base speed and angle are combined to finally calculate the pixel speeds by applying equations (3.4), (3.5), (3.6) and (3.7) with the respective linear speed (in RPM) and pixel coordinates. Furthermore, due to the symmetry applied and considering both forward and backward movements, it is required to correct the signaling of the resulting speeds, thus two additional functions are defined as follows:

$$signal_x(x) = \begin{cases} +1, & if\ x > W/2 \\ -1, & otherwise \end{cases} \quad (3.8)$$

$$signal_{RPM}(RPM) = \begin{cases} +1, & if\ RPM \geq 0 \\ -1, & otherwise \end{cases} \quad (3.9)$$

$signal_x(x)$ is related to the symmetry characteristic, and influences only the x-speed component. $signal_{RPM}(RPM)$, on the other hand, is related to forward/backward movement, and impacts only the y-speed component. Moreover, if the condition $x > W/2$ is satisfied, the x coordinate value must be updated before applying it to the other equations: $x = W - x$. In summary, the following algorithm is employed for the whole procedure:

1. If $|RPM| \geq 5$, compute:
    a. $signal_x(x)$ and correct x, if needed ($x = W - x$)
    b. $signal_{RPM}(RPM)$
    c. $C(RPM,x,y) = \gamma(RPM) * v_{trans}(x,y)$
    d. $\theta_{trans}(x,y)$
    e. Both speed components
        i. $v_{x_{trans}} = C(RPM,x,y) * \sin\big(\theta_{trans}(x,y)\big) * signal_x(x)$
        ii. $v_{y_{trans}} = C(RPM,x,y) * \cos\big(\theta_{trans}(x,y)\big) * signal_{RPM}(RPM)$
2. Else, $v_{x_{trans}} = v_{y_{trans}} = 0$

For a better understanding, consider the following example that was actually used for validation purposes: the speed of the robot is set to 75 RPM, and the coordinates of the pixel of interest are (102; 141), which are the only parameters needed. The pixel speeds, then, are computed as follows:

1. $signal_x(x) = signal(102) = -1$
2. $signal_{RPM}(RPM) = signal(75) = +1$
3. $C(RPM, x, y) = C(75{,}102{,}141) = \gamma(75) * v_{trans}(102{,}141) = 9.0 * 57.35 = 516.24$
4. $\theta_{trans}(x, y) = \theta_{trans}(102{,}141) = 31.66\, deg$
3. $v_{x_{trans}} = C(75{,}102{,}141) * \sin(31.554) * signal_x(102) = -270.94\ \frac{px}{s}$
4. $v_{y_{trans}} = C(75{,}102{,}141) * \cos(31.554) * signal_{RPM}(75) = 439.42\ \frac{px}{s}$

The resulting speeds are illustrated in Figure 46 below:

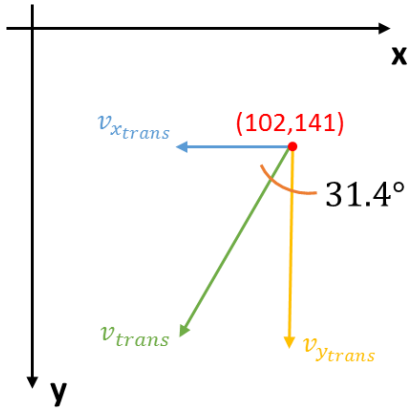


Figure 46. Illustration of the resulting speeds for pure translation

In order to validate the calculated pixel speeds, both speed and angle are manually calculated with (3.1) and (3.2), considering a time span of 0.5 seconds, and the final pixel coordinates (-30, 341):

$$v_{pixel} = \frac{\sqrt{\Delta x^2 + \Delta y^2}}{\Delta t} = \frac{\sqrt{(-30 - 102)^2 + (341 - 141)^2}}{0.5} = 479.27\ \frac{px}{s}$$

$$\theta = \tan^{-1}\left(\left|\frac{\Delta x}{\Delta y}\right|\right) = \tan^{-1}\left(\left|\frac{(-30 - 102)}{(341 - 141)}\right|\right) = 31.43°$$

Note that both the manually derived speed and the angle are close to the values obtained through the approximated functions, thus the pure translational motion approximation was applied.

3.3.2 Rotation impact

In addition to pure translation, it is necessary to analyze the impact of a pure rotation motion, in order to combine both results and finally compute the resulting pixel speeds in both axes. The procedure is closely related to the previously applied for pure translation, with the only difference being the resulting speed of the robotic system which must always be zero (pure rotation). This is achieved by setting a positive RPM value for one of the motors, and (the same but) a negative RPM for the other. For instance, if the desired motion is a pure rotation to the left, one should set +25 RPM for the right

motor, and -25 RPM for the left one. A sample optical flow field is depicted in Figure 47 below for a pure rotation motion.

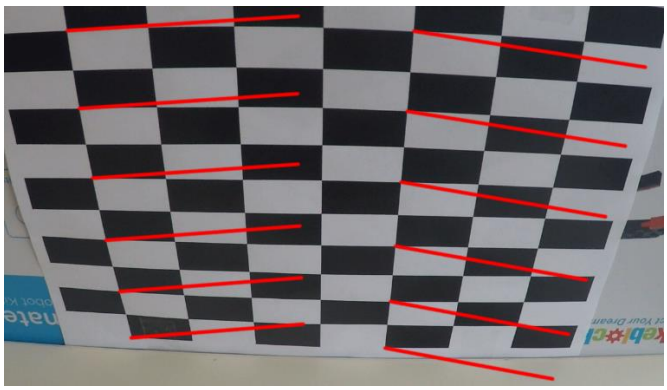


Figure 47. Optical flow field representation for pure rotation

Rotation information is retrieved from the IMU, more specifically the gyroscope of the robotic system. Although rotation around all axes are important, for this application only one is considered, which relates to changes in yaw (heading). Rolling and pitch are assumed not to happen, due to practical constraints: it is extremely difficult to set a constant pitch or roll value, which is a valid assumption considering the robot will operate in a flat surface, as previously discussed. Moreover, the robot moves slowly, with +-200 °/s (w.r.t. yaw) being considered the absolute maximum angular velocities.

A pure rotation motion presents similar behavior with respect to the relevant points previously derived during the pure translation analysis. It is important to notice, however, that the gyroscope data (w_{yaw}) is utilized instead of the RPM in this case, hence the scaling factor is dependent on the angular velocity. Moreover, symmetry can be applied once more, but requires extra signal correction due the fact the pixel speeds are mirrored with respect to two axes (check Figure 47 above), and the same 3x4 grid is used, with their respective central pixels.

Scale Factor: $\rho(\omega_{yaw})$

Initially, the rotation scale factor function must be computed, which depends on the yaw angular velocity. Note that the yaw depends on the set RPM, although RPM is not relevant in this case. By setting the RPM to a constant value for both sides (and thus approximately constant yaw), recording a video, and observing a single pixel, and repeating this for several yaw values, the scale factor function $\rho(\omega_{yaw})$ can be estimated. This procedure is performed for a certain range, from 3 to 110°/s, which corresponds to the minimum and maximum yaw, respectively. Furthermore, the base speed for the pure rotation case is defined as being the pixel speed for the minimum yaw, and the scale factor is determined by:

$$\rho(\omega_{yaw}) = \frac{v_{pixel}(|\omega_{yaw}|)}{v_{pixel}(|\omega_{yaw_{min}}|)} = \frac{v_{pixel}(|\omega_{yaw}|)}{v_{pixel}(3)} = \frac{v_{pixel}(|\omega_{yaw}|)}{v_{base}} \tag{3.10}$$

Once more, only the resulting speed is calculated with (3.2). Moreover, the described procedure was performed to a specific initial pixel with an 8°/s (i.e. 7 RPM) increment between measurements to avoid too much data. The following graph and speed scale function were obtained:

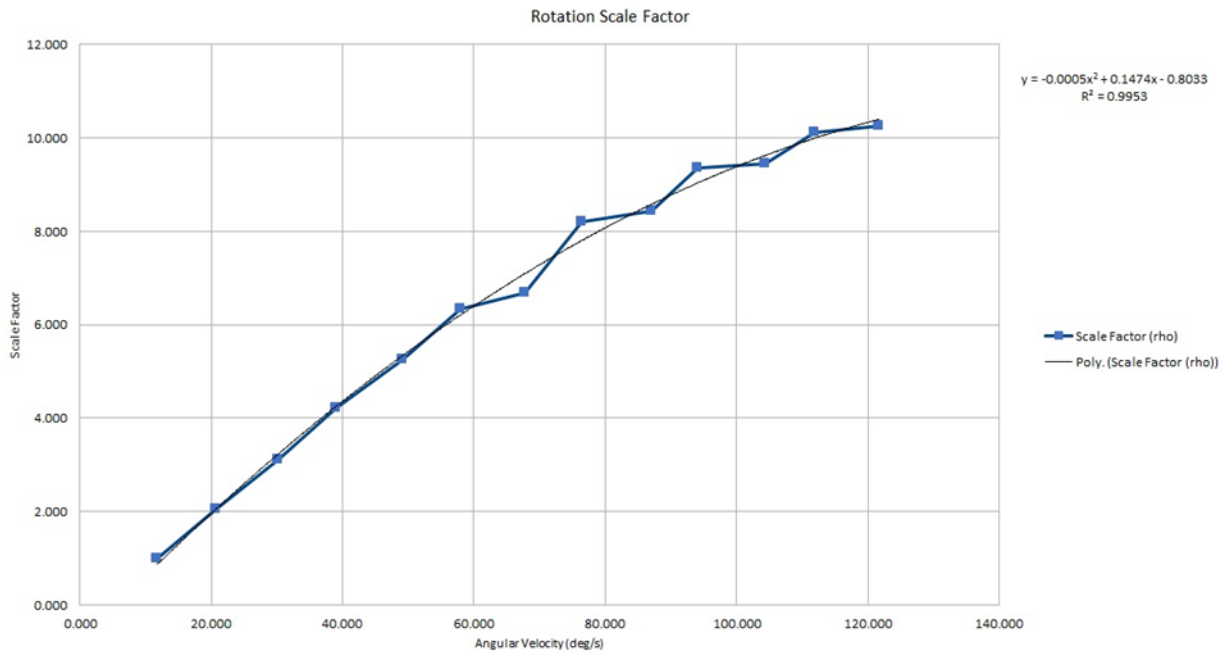


Figure 48. Speed scale factor function for pure rotation

In contrast to the scale factor function obtained for the pure translation case, the pure rotation motion requires a second order polynomial function, as shown above in the graph, and corresponds to:

$$\rho(\omega_{yaw}) = -0.0005 * |\omega_{yaw}|^2 + 0.1474 * |\omega_{yaw}| - 0.8033 \quad (3.11)$$

The standard deviation obtained for this approximation is 0.31 for ρ, and about 37.4 px/s for the pixel speed. Once more, the scale factor was validated by randomly picking other initial pixels, setting a known yaw within the valid range, observing its movement, and applying the above function to estimate both the scale factor, as well as the pixel speed which is computed through:

$$v_{pixel}(\omega_{yaw}) = v_{pixel}\left(\omega_{yaw_{min}}\right) * \rho(\omega_{yaw}) = v_{base} * \rho(\omega_{yaw}) \quad (3.12)$$

It is important to notice that $\rho(\omega_{yaw})$ should only be applied when $|\omega_{yaw}| \geq 3$, preventing unnecessary calculations when the robotic system is only vibrating due to normal movement. Handling the yaw signal is discussed in the upcoming subsections.

Base speed & angle

Figure 49. Optical flow field representation for the central pixels, during a pure rotational motion

The considerations applied to the pure translational scenario are once more used in order to determine the base speed and angle for the pure rotational motion: only half of the image is considered, and the central pixels of the 3x4 sub-division are analyzed. The test procedure, on the other hand, is performed as described in the previous sub-section, for the minimum yaw (3°/s). The 3D surface fit function is applied after the necessary data is collected, and both base speed $v_{rot}(x, y)$ and base angle $\theta_{rot}(x, y)$ functions are determined. Similarly to the pure translation case, both functions only depend on the pixel coordinates (x, y), with their respective plots shown below:

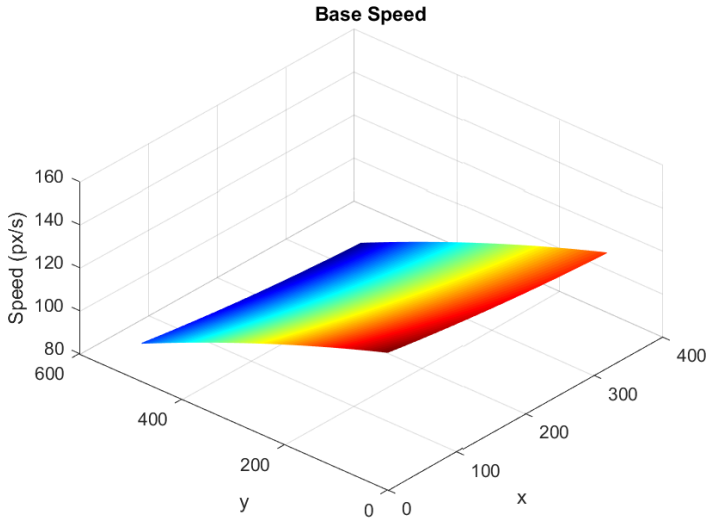


Figure 50. Surface fit for the base speed (pure rotation)

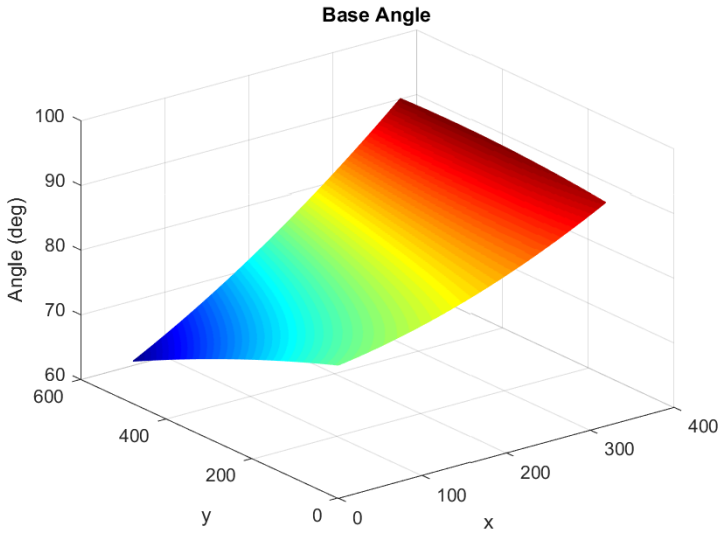


Figure 51. Surface fit for the base angle (pure rotation)

For the pure rotation scenario, the slowest pixel speeds are located around the bottom middle portion of the image (Figure 50), meanwhile the fastest ones around the top edge of the image. The surface is slightly more complex for the base angle (Figure 51), although it is simple to notice the highest angle values are located around the bottom middle of the image, and the lowest around the bottom corner. Both cases match the optical flow field previously observed.

The obtained polynomial for the base speed and angle, respectively, are:

$$v_{rot}(x,y) \cong 0.00005x^2 + 0.000002xy - 0.0496x - 0.00004y^2 - 0.0793y + 143.81 \quad (3.13)$$

$$\theta_{rot}(x,y) \cong 0.00008x^2 + 0.0001xy + 0.0183x - 0.000012y^2 - 0.0238y + 80.461 \quad (3.14)$$

The standard deviations obtained for the base speed and angle were 0.6548 px/s and 0.346°, respectively.

Resulting function

Both base speed and angle are combined to finally calculate the pixel speeds by applying equations (3.11), (3.12), (3.13) and (3.14) with the respective yaw velocity (in °/s) and pixel coordinates. Although signaling must be corrected in order to address the symmetry applied and considering both "turning right" and "turning left" movements, another function is defined and used in addition to equation (3.8):

$$signal_{\omega_{yaw}}(\omega_{yaw}) = \begin{cases} +1, & if\ \omega_{yaw} \geq 0 \\ -1, & otherwise \end{cases} \quad (3.15)$$

In contrast to the pure translation, $signal_x(x)$ influences the y-speed component. $signal_{\omega_{yaw}}(\omega_{yaw})$, on the other hand, impacts both x- and y-speed components. The algorithm

applied to estimate both speed components for the pure rotation case is extremely similar to the pure translation one, and is presented below:

1. If $|\omega_{yaw}| \geq 3$, compute:
    a. $signal_x(x)$ and correct x, if needed ($x = W - x$)
    b. $signal_{\omega_{yaw}}(\omega_{yaw})$
    c. $C(\omega_{yaw}, x, y) = \rho(\omega_{yaw}) * v_{rot}(x, y)$
    d. $\theta_{rot}(x, y)$
    e. If $\theta_{rot}(x, y) > 90° \rightarrow \theta_{rot}(x, y) = 90°$
    f. Both speed components
        i. $v_{x_{rot}} = C(\omega_{yaw}, x, y) * \sin(\theta_{rot}(x, y)) * signal_{\omega_{yaw}}(\omega_{yaw})$
        ii. $v_{y_{rot}} = C(\omega_{yaw}, x, y) * \cos(\theta_{rot}(x, y)) * signal_{\omega_{yaw}}(\omega_{yaw}) * signal_x(x)$
2. Else, $v_{x_{rot}} = v_{y_{rot}} = 0$

Notice that step (e) is necessary for this case due the fact the base angle might result in a value greater than 90°, which is not possible in practice – during experiments, the maximum angle obtained was about 89°. Moreover, due the fact this algorithm is alike the one previously presented for the pure translation case, the whole procedure will not be shown for the pure rotation. However, the same calculations were performed during development in order to validate the pure rotational motion, with the manually derived speed and angle being compared to values obtained through the approximated functions, yielding adequate results. For demonstration purposes, the results of a single calculation for $\omega_{yaw} = +20\ deg/s$, pixel coordinates (380; 279) and 0.5 seconds total movement time, are presented below:

$$Approximation: \begin{cases} v_{x_{rot}} = 1.9447 * 109.2 * \sin(90°) * 1 \cong 212.36\ px/s \\ v_{y_{rot}} = 1.9447 * 109.2 * \cos(90°) * 1 * 1 = 0\ px/s \end{cases}$$

$$Manual\ calculation: \begin{cases} v_{pixel} = \dfrac{\sqrt{(483 - 380)^2 + (281 - 279)^2}}{0.5} \cong 206.04\ px/s \\ \theta = \tan^{-1}\left(\left|\dfrac{(483 - 380)}{(281 - 279)}\right|\right) \cong 88.9° \end{cases}$$

3.3.3 Combined pixel speeds

With both translation and rotation impacts correctly analyzed and defined, it is necessary to combine them to compute the resulting pixel speeds in both axes by simply adding their respective components. The interest is in the combined pixel speeds V_x and V_y, which are computed as follows:

$$V_x = v_{x_{trans}} + v_{x_{rot}} \tag{3.16}$$

$$V_y = v_{y_{trans}} + v_{y_{rot}} \tag{3.17}$$

By substituting each component, (3.16) and (3.17) become:

$$V_x = C_{trans}(RPM, x, y) * \sin\left(\theta_{trans}(x, y)\right) * signal_x(x) + C_{rot}\left(\omega_{yaw}, x, y\right) * \sin\left(\theta_{rot}(x, y)\right) * signal_{\omega_{yaw}}\left(\omega_{yaw}\right) \quad (3.18)$$

$$V_y = C_{trans}(RPM, x, y) * \cos\left(\theta_{trans}(x, y)\right) * signal_{RPM}(RPM) + C_{rot}\left(\omega_{yaw}, x, y\right) * \cos\left(\theta_{rot}(x, y)\right) * signal_{\omega_{yaw}}\left(\omega_{yaw}\right) * signal_x(x) \quad (3.19)$$

In order to validate the combined pixel speeds, the robotic system is configured to run for a brief period with both translation and rotation movement, all data is logged and further processed, considering the pixel with coordinates (235; 253). By applying the above formulas for every sample, the final pixel coordinates computed are (235; 238), meanwhile the (real) coordinates determined manually is (252; 241): a deviation of 17 and 3 pixels in x and y, respectively. The combined pixel speeds, likewise both pixel coordinates for each iteration of the simulation are depicted in Figure 52 and Figure 53 below.

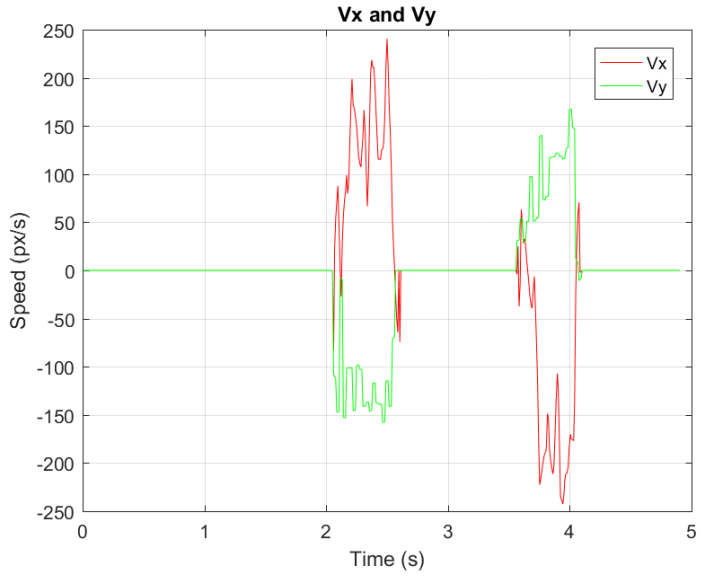


Figure 52. Pixel speeds computation based on real data

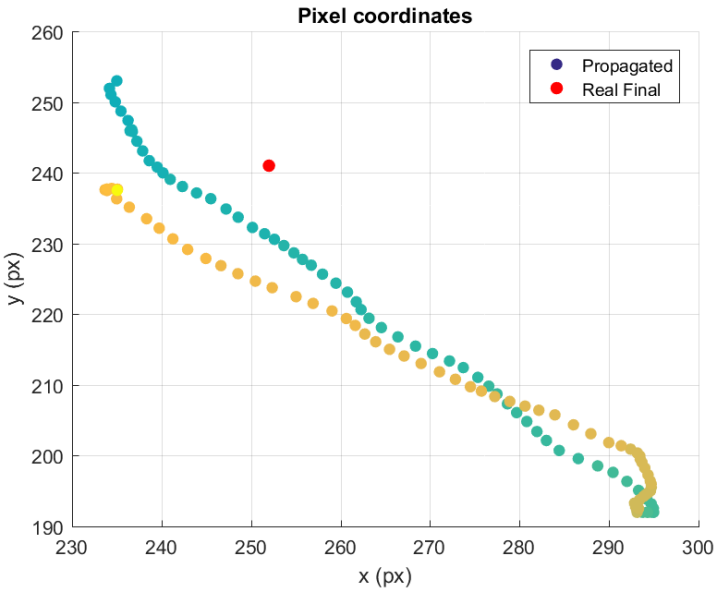


Figure 53. Pixel coordinates computation based on real data, and the actual final coordinates (red)

Note that real pixel coordinate is propagated based on the determined speeds and fed to the module once more, which corresponds to a greater error accumulation for the long run. During normal execution, on the other hand, the real pixel coordinates are provided every couple iterations by the Kalman filter itself, thus the error only accumulates for a brief period of time. Hence, in practice the Pre-Kalman filter response is much better.

Finally, the translation and rotation impacts division proved to be sufficient for this application. Although the data analysis needed for this procedure is rather burdensome to gather and process, this technique is much simpler and requires less processing power in comparison to standard techniques. The main advantages and disadvantages of the Pre-Kalman Filter module are summarized in Table 3 below:

Table 3. Overview of the advantages and disadvantages of the Pre-Kalman filter module

Advantages	Disadvantages
Simple and fast implementation	Camera must be static
	Application-specific
	Calibration required

Considering this method is far simpler and faster than most found in literature [10, 36, 37, 38, 39, 40, 41], and the fact the implementation should be running in an embedded system, this module is considered a favorable alternative. Not only does it use floating-point divisions at all, but it only requires trigonometric functions (i.e. sine and cosine) support in terms of dependencies. Refer to Appendix B: Pre-Kalman Filter Equations for a summary with the relevant equations presented in this section.

3.4 Kalman Filter

There are several types of filters, with a few specific ones being applied in regular odometry and other robotic applications. Mainly Kalman filters (KF) are implemented for linear systems, although depending on the system's model, the filter operations may become computationally expensive, being considered a sensor fusion technique [47]. For non-linear systems, extensions of the Kalman filter may be used, such as the Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF), with the latter being potentially useful when the platform supports parallelism. Moreover, another relevant option is the Particle Filter which behaves similarly to the KF itself, and it has been applied in tracking applications [45]. Notice that the aforementioned filters are based on Bayesian probability, which, in simple terms, determines what is likely to be true, based on past information [46]. Extensive explanation of the Kalman Filter theory is not the goal of this section, thus a brief description is provided, with the focus begin on the design of the filter itself.

Sensors do not output perfect information: not only are they noisy, but also frequently affected by external influences. More specifically, the latter refers to process noise, such as an ice path on the path of a car being tracked. Furthermore, purely relying on the sensor information is rather a simplistic approach, especially when the system can be modeled and past information used in order to infer information about the present more accurately. Consider, for instance, measuring temperature of a

room: on normal conditions, the temperature cannot greatly increase within seconds (i.e. 22°C to 35°), thus this information can be used to better predict the present temperature, based on previous samples. However, the heating system might have mal-functioned and the temperature indeed increased in an unusual way. This must modify the previous belief, and instead of being conservative one should trust more on the sensor readings, in this case. On the other hand, the characteristics of both sensor and system must be considered in order to properly reason about what is happening in reality. The KF was invented by Rudolf E. Kálmán to address this sort of problem in a mathematically optimal way, being initially applied in the aerospace sector and further in a variety of domains, such as aircrafts, submarines, financial market, chemical plants, among others [46]. In summary, whenever a sensor is part of the system and especially if sensor fusion is required, a Kalman filter or a similar technique is used. The KF is a model-based filter, and ponders (i.e. dynamically computes weights) between the model prediction and real measurements, considering noise characteristics of both. Although the KF can be generalized for colored noise and cross covariances, noises are assumed to be white gaussian and no correlation between process and measurement noises is expected for this thesis.

For this application, a Kalman filter was chosen due to its simplicity in terms of implementation and understanding, alongside being the optimal filter for the system, which is linear. Moreover, the KF is widely applied in tracking applications, being able to indirectly estimate speed in addition to position, for instance, and is easily parallelized for two dimensions (i.e. x and y). The most important characteristics for this system, however, are its capability of providing an output when (1) the sensor fails (i.e. false negative from the neural network), that might occur due the neural network's accuracy – 90%, in the best-case scenario –, (2) or in case of occlusions. Meanwhile, the deviation of the target position is also smoothed.

The standard equations of the KF can be divided into two simple steps, predict and update: the former (a.k.a. time update) propagates the model, meanwhile the latter (a.k.a measurement update) computes the filter's output based on the new measurement, and updates the covariance matrices. These equations [2] are shown below, with all variables being matrices:

- Predict step

$$\hat{x}_{k|k-1} = F * \hat{x}_{k-1|k-1} + B * u_k \quad (3.20)$$

$$P_{k|k-1} = F * P_{k-1|k-1} * F^T + Q_k \quad (3.21)$$

- Update step

$$S_k = H * P_{k|k-1} * H^T + R \quad (3.22)$$

$$K_k = P_{k|k-1} * H * {S_k}^{-1} \quad (3.23)$$

$$\tilde{y}_k = z_k - H * \hat{x}_{k|k-1} \quad (3.24)$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k * \tilde{y}_k \quad (3.25)$$

$$P_{k|k} = (I - K_k * H) * P_{k|k-1} \quad (3.26)$$

With $\hat{x}$ being the state matrix, F the state transition model, B the control input, P the error covariance matrix, Q the process covariance matrix, H the observation model, S the innovation covariance matrix, R the measurement covariance matrix, K the Kalman gain matrix, $\tilde{y}$ the residual, z the measurements matrix and I the identity matrix. It is extremely important to define the notation used for the indices, which respects the following (M is a generic variable):

- $M_{k-1|k-1}$: refers to the previous state, which combines all the previous states and observations;
- $M_{k|k-1}$: refers to the *a priori* state, which combines the current state estimate without observations;
- $M_{k|k}$: refers to the *posteriori* state, which combines the current state estimate with observations.

Note the model can be propagated without measurements due the fact it is time-based, hence whenever the neural network outputs a false negative, the KF can still output a value. In the upcoming sub-sections, the filter matrices and parameters will be further detailed, except the Kalman gain matrix which is computed internally and does not need to be defined.

3.4.1 Filter Design

Several steps are necessary to design the filter, which requires different matrices to be properly defined and validated. Initially, the available input and required output data of the filter are analyzed, with the overview of the module being depicted in Figure 54:

- Inputs
  - Target coordinates outputted by the neural network (in pixels);
  - Pixel speeds computed by the Pre-Kalman filter (in px/s).
- Outputs
  - Filtered target coordinates (in pixels), which are forwarded to the pixel control;
  - (Optional) Target estimated speed (in px/s).

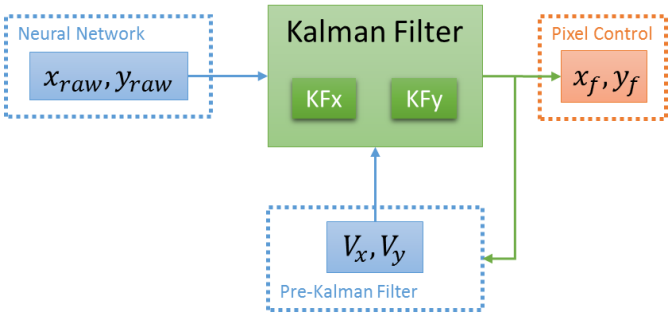


Figure 54. Kalman Filter overview

In order to reduce complexity, during the design only a single dimension (x) is contemplated instead of two, due the fact analysis is valid for both and would only increase the matrices sizes and overall complexity. However, the final implementation comprises two instances of the Kalman filter (KFx and KFy in Figure 54), one for each dimension, and can be run in parallel.

Additionally, the filter requires extra prediction steps to keep track of objects that might disappear during run time due to occlusion, or a false negative outputted by the neural network module, as previously described. Occlusion might occur for several frames, and false negatives, on the other hand, are related to the neural network's accuracy and might happen for a few frames (e.g. 6 frames for a 60 fps camera). It is also important to notice that the inputs are provided at different rates, which must be considered during implementation and testing.

State Transition model and variables

Based on the available inputs, it is possible to define and analyze which model must be used. For this specific application, system modelling is straight-forward: the goal is to track an object in an image, thus Newton's (linear) equation of motion [46] can be applied (w.r.t. pixels instead of distance):

$$x = x_0 + v * \Delta t \tag{3.27}$$

With x and x_0 being the final and initial positions (in pixels), respectively, v the target's speed (in px/s) and Δt the time. Moreover, constant acceleration and jerk can also be included, with their equations being shown below:

$$x = x_0 + v * \Delta t + \frac{a * \Delta t^2}{2} \tag{3.28}$$

$$x = x_0 + v * \Delta t + \frac{a * \Delta t^2}{2} + \frac{j * \Delta t^3}{6} \tag{3.29}$$

With a being the acceleration (in px/s^2) and j the jerk (in px/s^3). However, incorporating acceleration, jerk, or both, overcomplicates the system by augmenting the size of matrices (and thus computation time), and does not necessarily increase the filter's performance [46]. More importantly, it also requires handling excessively large numbers (10^6), especially when jerk is used, which further increases computation time. Taking into account these facts, the 1st order model which considers only the position and speed was chosen to be used, with the state variables and matrix being defined as follows:

$$\hat{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \tag{3.30}$$

$$F = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{3.31}$$

With x being the target's position (in pixels), $\dot{x}$ its speed (in px/s), and Δt the time between prediction steps. Notice that, simplifying equation (3.20) to $\hat{x}_{k|k-1} = F * \hat{x}_{k-1|k-1}$, it becomes clear how (3.30) and (3.31) relates to the model itself:

$$\begin{cases} x_{k|k-1} = x_{k-1|k-1} + \dot{x}_{k-1|k-1} * \Delta t \\ \dot{x}_{k|k-1} = \dot{x}_{k-1|k-1} \end{cases}$$

Note that the object speed ($\dot{x}$, 1st derivative) is a hidden variable, which is estimated and updated internally by the filter, and this model is applied for both x- and y-coordinates.

Control Input model and variables

In addition to the state transition model which corresponds to the 1st order Newton's equation of motion, there is another input to the filter: the robot's ego-motion impact, or the pixel speeds derived by the Pre-Kalman filter module. Such information is relevant and is included in the model through the control input model and variables, being the trigger for the prediction step. It is relevant to note that the prediction step is executed every 7 milliseconds (140 Hz), due the fact ego-motion data is

available at this rate. For this implementation, we define the control input model and variable as follows:

$$B = \begin{bmatrix} \Delta t \\ 0 \end{bmatrix} \tag{3.32}$$

$$u_k = [v_x]_k = v_{x_k} \tag{3.33}$$

With v_{x_k} being the combined pixel speed (in px/s), in this case only for the x-axis. Now, (3.20) can be fully unfolded, yielding:

$$\begin{cases} x_{k|k-1} = x_{k-1|k-1} + \left(\dot{x}_{k-1|k-1} + v_{x_k}\right) * \Delta t \\ \dot{x}_{k|k-1} = \dot{x}_{k-1|k-1} \end{cases}$$

These equations complete the modelling step and fully describe the system. Note that the new position x is computed based on the target's motion ($\dot{x}_{k-1|k-1}$) plus the combined pixel speed (v_{x_k}): their sum composes the final speed. Note that the robot's ego-motion can be not used as a control input, which will be discussed in another sub-section, and once more this model is used for both filters (x and y coordinates).

Process Noise and Initial Error Covariance

The process noise matrix Q includes the standard deviation related to the external influences, namely the process. The variance of the process $\sigma_{process}$ comprises unpredictable changes on the system, such as bumps on the way when tracking a car, for instance. Moreover, the car might be slipping, and even the wind may impact the model, and these interferences must be taken into account through defining a coherent variance for the process, usually done experimentally [46, 47]. More specifically for this project, one can consider the absolute maximum speed the target could move in any direction, due the fact the target is another robot which can unpredictable maneuver. By reproducing the behavior of the target, which does not move fast, it was possible to roughly determine the standard deviation (and thus variance): on average, the maximum speed inferred did not exceed 6 px/s for $\Delta t = 0.05s$ (20 Hz, worst-case scenario) on both axes. Due the fact this value was derived in practice, only maneuvers were considered, and in order to cover about 97% of the possible cases, the process noise defined corresponds to at least 3σ, thus $\sigma_{process}$ should be 18, but was define as $\sigma_{process} = 20$. According to [46] and [47], the process noise covariance matrix for a 1st order filter based on the presented Newton's equation of motion is defined as follows:

$$Q_k = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix}_k * {\sigma_{process}}^2 = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix}_k * 20^2$$

The error covariance matrix P, on the other hand, can be initialized in several ways, depending on the knowledge one has about the system. Because this matrix is updated every iteration, it is commonly initialized as a diagonal matrix with the respective variance of the state variables [47] – the covariances between position and speed are dynamically computed by the filter itself. If the initial state (i.e.

position and speed) is known, it is common to initialize the matrix with zeros. Otherwise, the diagonal must be filled with either high values or the maximum expected variance of the variables. The generic form of the error covariance matrix for the system is as follows:

$$P_0 = diag({\sigma_x}^2, {\sigma_{\dot{x}}}^2) = \begin{bmatrix} {\sigma_x}^2 & 0 \\ 0 & {\sigma_{\dot{x}}}^2 \end{bmatrix} \tag{3.34}$$

Due the fact the starting position is unknown, the initial variance of the position is set to the image width (320 pixels) and height (240 pixels) for x and y, respectively. Although the speed is also unknown at start up, one may reason about it due the fact this will be update during run-time by the filter itself: maximum speeds of 40 and 20 px/s were considered and used for x and y axes, respectively, as their standard deviations. The maximum speed used for the y-axis is smaller due the fact the target moves slower with respect to this axis, typically less than half the speed in comparison to the other axis. Considering these values, (3.34) was set to:

$$P_{0_x} = \begin{bmatrix} 17.88^2 & 0 \\ 0 & 40^2 \end{bmatrix}, \qquad P_{0_y} = \begin{bmatrix} 15.49^2 & 0 \\ 0 & 20^2 \end{bmatrix}$$

Measurement Noise and Observation Model

The measurements are also noisy, due to the sensor's characteristics and environmental factors, and are taken into account by the KF. The measurement noise R is easier to define, as its variance can usually be calculated from real measurements. For this project, the measurements are provided by the neural network, which outputs two values – X and Y coordinates. A single coordinate corresponds to a single measurement, and does not comprise any speed measurements. Consequently, R is reduced to a scalar value – the measurement variance – as shown below:

$$R = var(x_{sensor}) = {\sigma_{x_{sensor}}}^2 \tag{3.35}$$

In order to determine the measurement variance for both axes, the neural network output was analyzed for short videos (4 videos in total, 60 fps, 3-6 seconds duration) in which both target and robotic system are moving (i.e. dynamic). All frames were then manually annotated with the target's position, and fed to the neural network. Both manual annotations and neural network's output were compared by calculating the respective standard deviations for each axis, and finally the greatest standard deviation among the videos (i.e. worst case) were used: $\sigma_{x_{sensor}} \cong 10$ and $\sigma_{y_{sensor}} \cong 10$. Once more, in order to cover as many cases as possible, 3σ values were used and rounded up, thus (3.35) becomes, for x and y, respectively:

$$R_x = 30^2, \qquad R_y = 30^2$$

The observation model H is used to combine the measurements with the state variables, and due the fact there is only a measurement for the position, it can be defined as follows for this project:

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

Initial Conditions

Finally, the initial conditions are the only missing definition before the complete filter structure is properly set. The initial state (i.e. position and speed) is assumed unknown, thus they are both set to zero, as follows:

$$\hat{x}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

3.4.2 Design Space Exploration

Before moving forward with testing the KF, it is important to present a few variations which are required for this application. More specifically, alternate formulations, smoothing techniques, steady-state filtering and adaptive filtering are explored in this sub-section. The former addresses mainly numerical problems (i.e. precision impact), and computational complexity, likewise steady-state filtering. Smoothing, on the other hand, tries to further stabilize the filter at a cost of introducing delays, and adaptive filtering addresses convergence of the filter when abrupt changes occur (e.g. target maneuvering). Due the fact only adaptive filtering was used in the final implementation, it will be thoroughly discussed meanwhile the other techniques are briefly explained.

Furthermore, a few comparisons are required in order to refine both design and final implementation of the Kalman filter. The following points and questions are discussed in the end of this section, which required simulations:

- Control input influence: How does the control input (ego-motion) impact the performance?
- Higher order models: Do higher order models perform better?
- Coupling: How does coupling (both x and y coordinates combined) impact the performance?
- Adaptive Filtering: Is adaptive filtering necessary? Which technique?

All these points were analyzed with a simulated KF in MATLAB, configured as discussed in the previous section. Tests are based on noisy non-realistic data, which respect the standard deviations defined. Moreover, the period was set based on the worst-case scenario, with the neural network providing data at 20 Hz and a maximum jitter of 10% to test the robustness of the filter ($\Delta t = 0.05s \pm 10\%$). The prediction step is executed Positive results were kept during the upcoming analysis, with modifications being done to the (simulated) KF implementation.

Alternate formulations

The Kalman Filter equations may be modified but remain mathematically equivalent, depending on the system and their specific characteristics. Note, however, that alternate formulations are usually constrained by additional requirements. For instance, in order to increase precision, one might implement a square root filtering technique [47] at a cost of increasing the complexity and implicitly defining the covariance matrix R as either diagonal, or constant. Among others, the following formulations were considered relevant:

- Sequential KF: avoids matrix inversions, as the measurement-update is done with a single measurement at a time. However, the covariance matrix must either be diagonal or constant;
- Information filtering: applied when the number of measurements is much larger than the number of states, and both R and Q are constant (implies constant Δt). Moreover, it is better when the uncertainties are large ($P \to \infty$);

- Square Root and U-D filtering: Both methods require orthogonal transformation algorithms [47], with the measurement-update alike the sequential KF implementation. In general, both requires greater computational effort while mitigating numerical problems in implementations – which can be a concern in 16- and 8-bit precision controllers

Considering only one measurement is received, and that the covariance matrix R is constant for this application, one may be tempted to use the Sequential KF formulation. However, there is actual no need because the only inversion (performed in matrix S) is reduced to a division when a single axis is considered, which is the case for this application. Hence, no alternate formulation was necessary. For bigger matrices, however, the Sequential KF formulation may greatly reduce the computational complexity, especially for embedded systems and real-time applications.

Smoothing

Smoothing corresponds to stabilizing even further the output of the filter, and basically 3 different types of smoothers can be used [47]:

- Fixed-Interval: computes the optimal state estimates of a batch of measurements at each time j, mostly used for post-processing;
- Fixed-Lag: computes the optimal state estimate at each time *j*, while using measurements up to and including time ($j + N$). N is fixed and corresponds to the "lag" (i.e. delay);
- Fixed-Point: computes the optimal state estimate at time j, considering all future measurements – used for camera pose estimation [46], for instance.

Notice that all techniques introduce delays, thus none are applied for this project due the fact one should cope with the slow refresh rate of the camera and neural network combination. However, fixed-lag smoothing could be of great use if the refresh rate is higher (e.g. 240 fps) and the introduced lag does not largely impact the result.

Steady-State Filtering

Although the standard KF is time-variant, implementations for time-invariant systems with also time-invariant process- and measurement-noise covariances exist and compose a specific class, namely steady-state KF. Such alternative is primarily employed in embedded systems with memory and computational effort constraints, as the Kalman gains are not dynamically computed in real-time. Even though the steady-state KF is not optimal as the Kalman gains are not computed at each time step, its performance is nearly indistinguishable from that of the time-varying filter [48], when there is no jitter present. Note that the steady-state KF is still a dynamic filter, with "steady-state" referring only to the Kalman gain matrix computation. Among others, the alpha-beta filter [49] is a steady-state KF that is applied to a two-state Newtonian system with position measurement [48], such as the one discussed in this project. The Kalman gain for this filter is defined as:

$$K = \begin{bmatrix} K_1 \\ K_2 \end{bmatrix} = \begin{bmatrix} \alpha \\ \frac{\beta}{\Delta t} \end{bmatrix} \tag{3.36}$$

The gains are computed as follows:

$$K_1 = -\frac{1}{8} * \left[\lambda^2 + 8\lambda - (\lambda + 4)\sqrt{\lambda^2 + 8\lambda}\right] \tag{3.37}$$

$$K_2 = \frac{1}{4} * \left[\lambda^2 + 4\lambda - \lambda\sqrt{\lambda^2 + 8\lambda}\right] \tag{3.38}$$

And λ is defined as:

$$\lambda = \frac{\sigma_{process} * \Delta t^2}{\sigma_{measurement}} \tag{3.39}$$

Hence, α and β can be compute with the following formulas:

$$\begin{cases} \alpha = K_1 \\ \beta = \Delta t * K_2 \end{cases}$$

With Δt corresponding to the sampling period, in seconds. This filter is particularly useful for embedded systems, especially when the sampling frequency is constant. As it can be seen above, lambda is a function of the sampling frequency and the advantage of not computing the Kalman gain every iteration is lost in case the refresh rate frequently changes or an adaptive filtering technique is used, with the filter performing poorly in both scenarios. Due the fact the robustness of the filter would be reduced, and considering the application might have jitter, the steady-state KF was not applied.

Control Input Influence

The control input corresponds to the robot ego-motion impact. By considering it ($u_k \neq 0$), one segregates the speed of the object being tracked from the influence of the ego-motion, thus the actual target speed (in px/s) is tracked. When the control input is not considered ($u_k = 0$), the target estimate speed has an offset (Figure 55b) in comparison to when it is considered (Figure 55a), as depicted in Figure 55. It is relevant to notice the control input is directly related to the prediction step, which is executed every 7 milliseconds (140 Hz) and corresponds to the refresh rate of the ego-motion data.

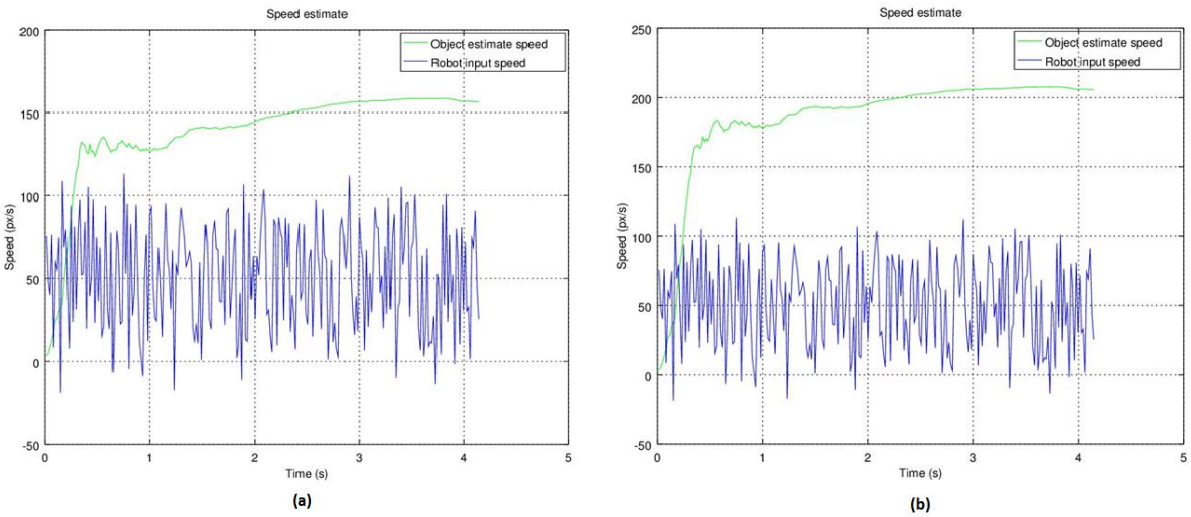


Figure 55. Control input influence on estimated speed, when being (a) and not begin (b) considered

The filter performance is greatly degraded when the robot, target, or both are maneuvering: the latter is the worst case in which the filter takes a long time to converge to the real values. Such scenario was simulated, and the results are shown in Figure 56 below:

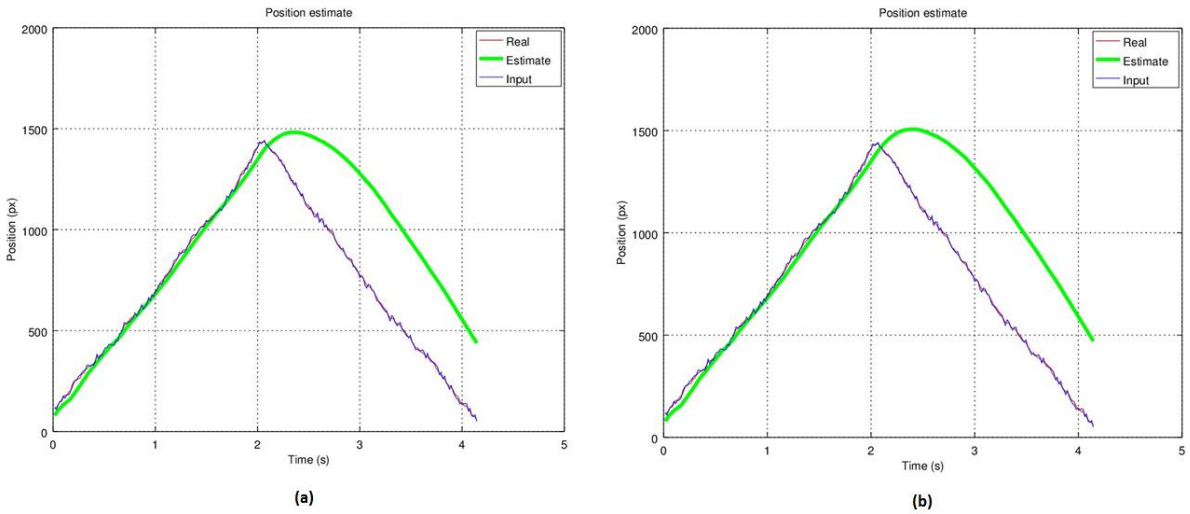


Figure 56. Control input influence on estimated position, when being (a) and not begin (b) considered

Notice the case where the control input is not taken into account (Figure 56b), the KF takes longer to converge after the target maneuvers, in contrast to the other case (Figure 56a). In order to properly compare the results, the RMSE with respect to the input (fake) data, and the output of the (simulated) KF was calculated for both scenarios being considered, which resulted in $RMSE_{with} \cong 231.47\ px$ and $RMSE_{without} \cong 249.67\ px$. Hence, the filter indeed performs better with the control input being considered. Although the filter currently performs poorly in maneuvering scenarios, adaptive filtering will be integrated to specifically address this issue. For now, it is enough to conclude the control input should be considered.

State Transition model order

As previously discussed, both acceleration and jerk can be included in the Newtonian model for the system (equations 3.16 and 3.17). However, performance is not always increased by doing so [46], thus higher models are further explored. Initially, only the acceleration is considered, due the fact including jerk increases complexity and in the case the performance decreases for the acceleration-only scenario, it is not worth augmenting the order of the system even further. Although the derivation of the matrices and equations of the filter for this model are not presented, the position estimate for this case is shown below and the RMSE computed corresponds to $RMSE_{acc} \cong 191.10\ px$.

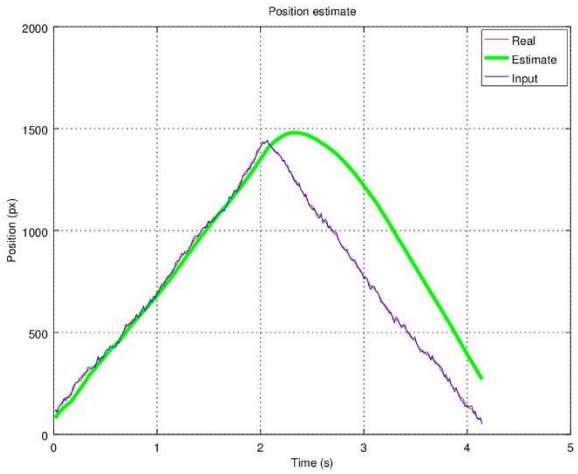


Figure 57. Position estimate when acceleration is included in the model

Although the RMSE is smaller when the acceleration is included (in comparison to $RMSE_{with}$), the estimated acceleration values might become extremely large (Figure 57a), and the Kalman gain for the acceleration is rather high with respect to the others (Figure 57b), as shown in Figure 57. The former corresponds to trusting the (hidden) acceleration estimate over the speed and position for the model - not optimal due the fact the filter trusts the model less, overall -, meanwhile the latter impacts on the computational complexity, which is greatly increased due to the matrix operations that must be performed.

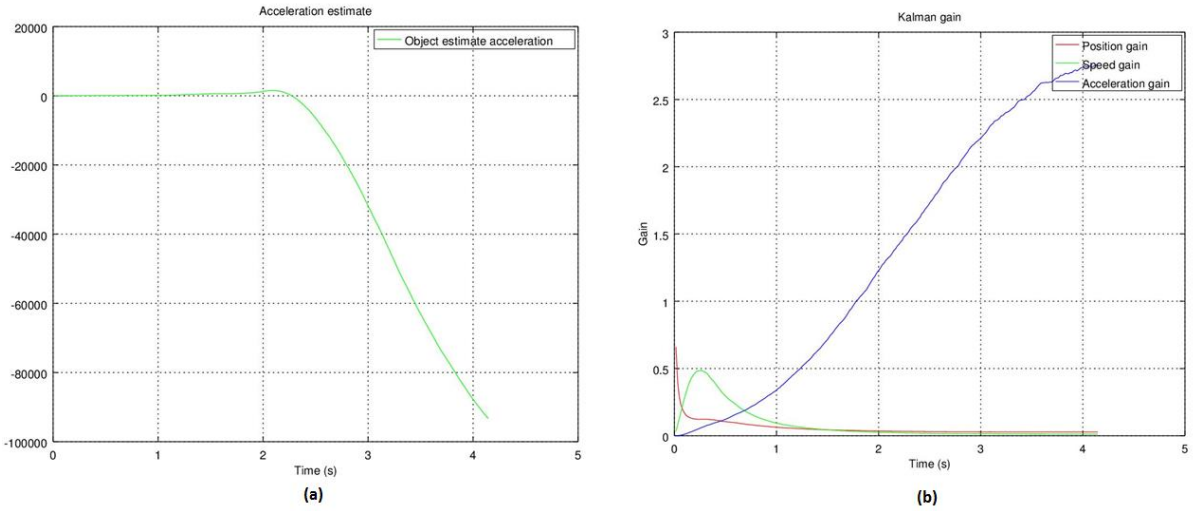


Figure 58. Acceleration estimate (a) and Kalman gains (b) when acceleration is included in the model

Considering the aforementioned facts, including the acceleration corresponds to increasing complexity without greatly impacting the performance of the filter, hence the simplest model as previously discussed is chosen to be used.

Coupling

One might argue both coordinates (x and y) can be coupled in a single Kalman Filter, instead of being implemented in different instances. Coupling the coordinates corresponds to simply augmenting all matrices of the filter by a factor of two (2), which has no impact at all on the output(s), as can be seen in Figure 59 below which depicts a simulation for both coordinates:

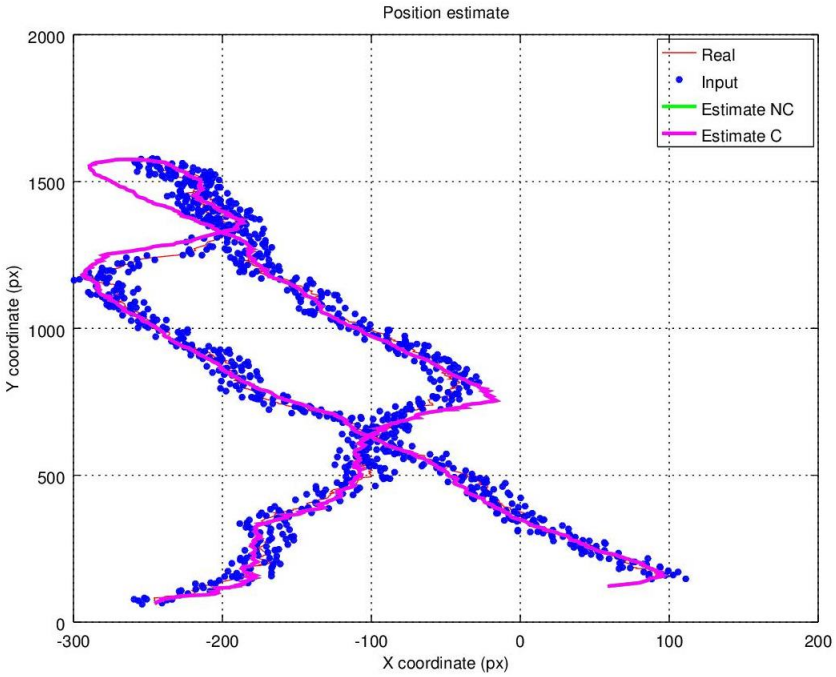


Figure 59. Position (x and y) estimate for both coupled (C) and uncoupled (UC) implementations

Note that for this simulation, the RMSE for each coordinate is computed, and corresponds to the same value: $RMSE_{C\ or\ NC} \cong (17.34, 13.16)\ px$. Although the performance is the same when both coordinates are coupled, the uncoupled version is less computationally expensive due the fact it operates over smaller matrices. Furthermore, each uncoupled filter can be better tuned with respect to the adaptive filters discussed in the next sub-section, and is able to run in parallel as there is no data dependency. Considering these aspects, the uncoupled filter was chosen to be implemented.

Adaptive Filtering

The KF considers all previous samples in order to compute its output, and may be biased in case of sudden changes in the system. This directly impacts the responsiveness of the filter, which slows down and might take several iterations before converging to the real value. To address this issue, adaptive filtering techniques are widely used in applications in which abrupt changes or discontinuities may occur, such as tracking. There are several techniques for adaptive filtering, and one may combine different ones depending on the desired behavior and application. Only two are discussed for this application, which are simple and sufficient for tracking applications [46, 47]: Fading Memory and Adjustable Process Noise. Implementing both is adding unnecessary redundancy, hence each one is implemented separately and the best one chosen to be implemented.

Adjustable Process Noise: ϵ version

Initially, the adjustable process noise technique for detecting maneuvering targets is explored, in which the process noise is scaled accordingly based on a metric involving the residual ($\tilde{y}_k$). Basically, the process noise is increased when the residual is large, or decreased otherwise. Hence, for the former, the KF will favor the measurement more. There are several ways of applying this idea, but only the Continuous Adjustment [47] is explored in this project due to its simplicity in terms of design and implementation, and more specifically the method from Bar-Shalom [48]. Such technique consists of normalizing the square of the residual for every iteration as follows:

$$\epsilon_k = \tilde{y}_k{}^T * S_k * \tilde{y}_k \tag{3.40}$$

With $\tilde{y}_k$ being the residual and S_k the measurement covariance matrix. Note that when the former is a scalar, (3.40) simplifies to:

$$\epsilon_k = \frac{\tilde{y}_k{}^2}{S_k} \tag{3.41}$$

Squaring the residual ensures the signal is always greater than zero, and normalizing by the measurement covariance scales the signal so that one can distinguish when the residual is markedly changed relative to the measurement noise [46]. Implementation is straight forward, with ϵ_k being computed in every filter iteration and compared with a maximum value. If such limit is exceeded, the process noise matrix Q_k is scaled up by a constant factor, or down otherwise.

For the adjustable process noise, one must initially analyze the normalized square of the residual (ϵ) for each uncoupled filter, as shown in Figure 60, in order to properly define the threshold value. Additionally, in order to compare the filter with and without the adjustable process noise technique, the RMSE for the latter is computed, and corresponds to $RMSE_{no-adaptive} \cong (9.19, 5.65)\ px$.

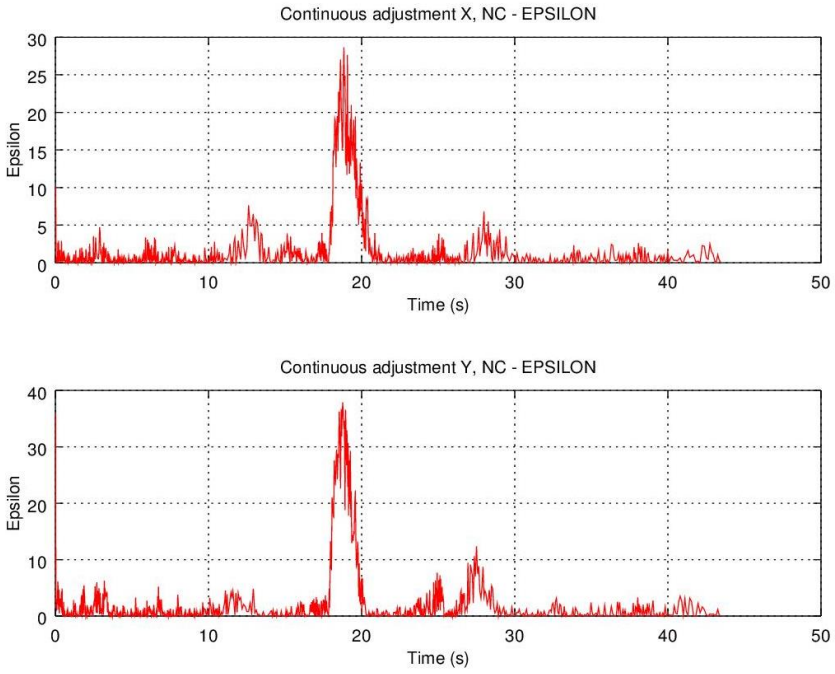


Figure 60. Initial ϵ values for both coordinates

Based on the graphs shown in Figure 60 above, thresholds of 4 and 7 were defined for the x and y filters, respectively, as they must react to changes rapidly. The scale factor for the process noise covariance matrix is then adjusted in order to obtain the lowest RMSE, and corresponds to 1000 for both cases: $RMSE_x(scale = 1000) \cong 9.44$ px, $RMSE_y(scale = 1000) \cong 8.10$ px. Instead of directly checking the position estimate, the residuals of both coordinates are more interesting for now due the fact they should be within $\pm 1\sigma$, and are depicted in Figure 61 below. The position estimate will only be depicted fort the best adaptive filtering technique.

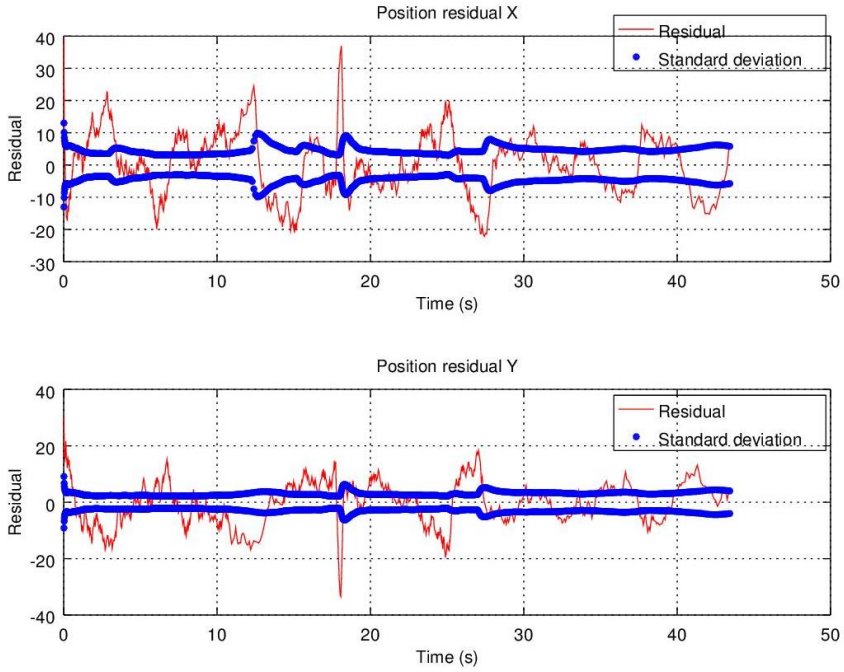


Figure 61. Position residuals for the continuous adjustment technique

Although the residuals are within an acceptable range of $\pm 6\sigma$ in general, being suppressed by the filtering technique applied, the filter is scaled quite often – peaks on Figure 61 for the standard deviation corresponds to a scale up or down procedure –, which increases execution time due the additional operations, especially for scaling the process noise matrix. The ϵ values when the technique is applied, on the other hand, are reduced as expected as shown below.

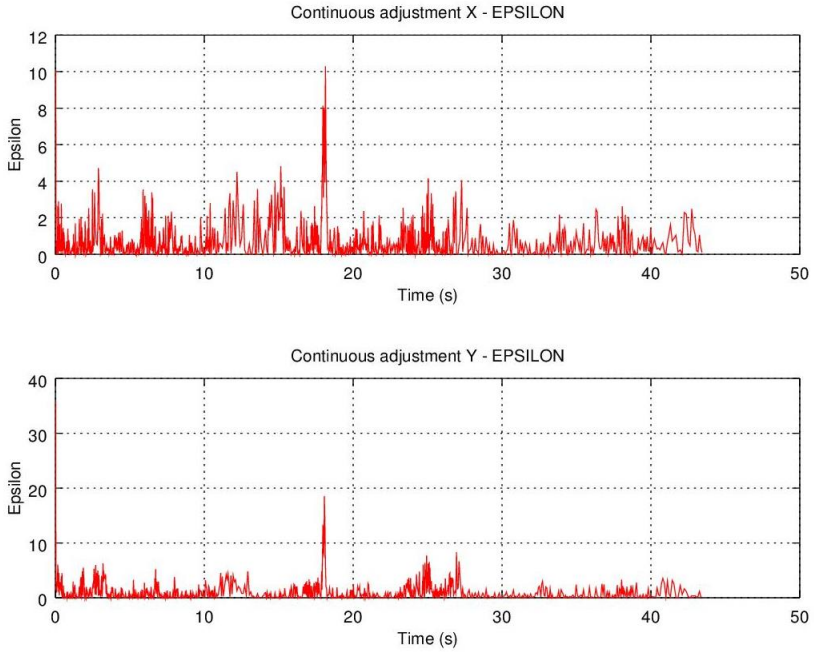


Figure 62. ϵ values for the continuous adjustment implementation

Despite the continuous adjustment algorithm is interesting for applications in which there is not many abrupt changes, it might be the best option for this system. However, before deciding whether this adaptive filter should be implemented, the fading memory technique is discussed.

Fading Memory

Fading memory consists of giving more weight to recent samples and less to older ones by computing the error covariance matrix in a slightly different manner. Its implementation for low order kinematic filters is very simple, which drastically reduces the amount of computation required in comparison to other adaptive filtering techniques [47]. The only modification in the original KF formulation is on the error covariance matrix computation (3.21), which becomes:

$$P_{k|k-1} = \boldsymbol{\alpha^2} * F * P_{k-1|k-1} * F^T + Q_k \tag{3.41}$$

Where α is a scalar, greater or equal to 1, and typically in-between 1.01 and 1.05. The higher α is, the less weight older samples have. Note that for the specific case where $\alpha = 1$, (3.39) reduces to (3.21) and the filter is a standard KF. The idea of this modification is straight-forward: by increasing the error covariance matrix, the KF is more uncertain about the estimate and hence gives more weight to the measurements provided by the neural network. This technique was applied for this project, and will be further explored in order to determine the best value for α.

Due the fact the maximum value usually does not exceed 1.05 [46, 47], this adaptive technique was compared for $1.01 \leq \alpha \leq 1.05$, with 0.01 interval and the respective RMSEs computed for both coordinates. Finally, the chosen alphas correspond to the smallest RMSEs obtained: $RMSE_x(\alpha = 1.02) = 4.51$ px, $RMSE_y(\alpha = 1.01) = 3.55$ px.

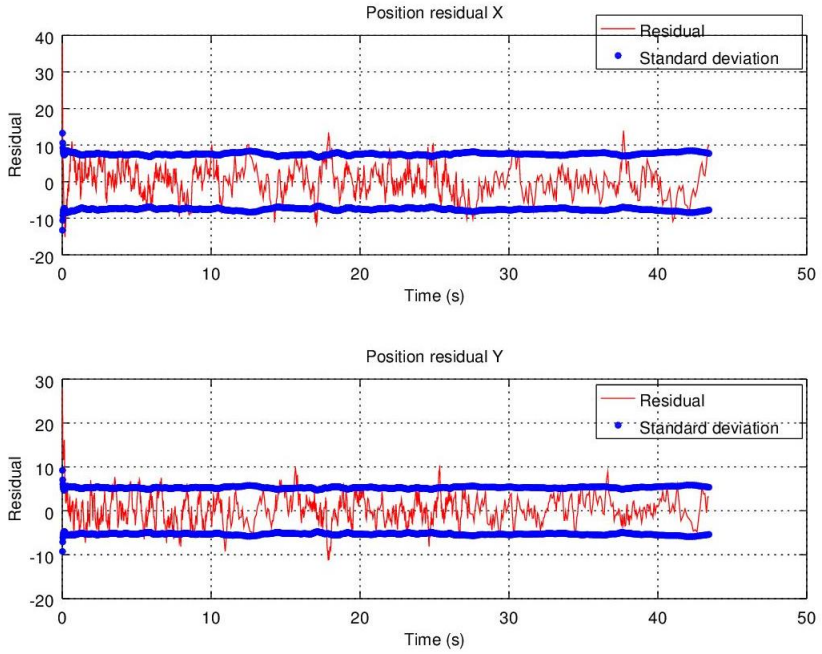


Figure 63. Position residuals for $\alpha_x = 1.02$ and $\alpha_y = 1.01$

In comparison to the adjustable process noise technique, the residuals are smaller overall. Note that by calculating epsilon when using the fading memory filter, it becomes evident that there is no need to implement an additional adaptive technique, as shown below:

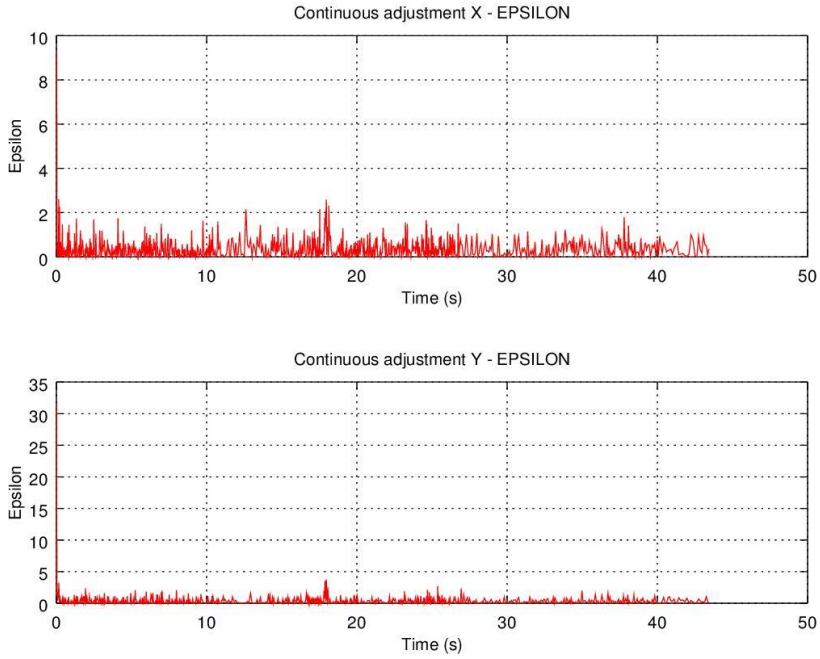


Figure 64. ϵ values for the fading memory implementation

An overview of the RMSE values computed for both adaptive filtering techniques is summarized in Table 4. Note the adjustable process noise performs worse than when not using an adaptive filter technique, and ultimately the fading memory implementation improved about 49% and 63% for x and y coordinates, respectively.

Table 4. RMSE comparison between adaptive filter techniques

	Coordinate	Adaptive filter technique			Improvement (%)
		Not implemented	Adjustable Process Noise	Fading Memory	
RMSE (px)	x	9.19	9.44	4.51	49.08
	y	5.65	8.10	3.55	62.83

Not only is the fading memory technique simple to implement, but the residuals for this technique are better than the adjustable process noise. Hence, the fading memory technique was chosen as the adaptive filter to be used, which complies to the literature [46] due the fact in this application there might be several maneuvers from both robot and target. Although implementation of the fading memory filter is simpler, such technique modifies the KF by trusting the samples over the model, and more specifically, the most recent ones. The adjustable process noise algorithm, on the other hand, only gives more weight to the samples when a certain threshold is reached, instead of all the time. Moreover, this algorithm might be interesting when few abrupt changes occur.

Finally, the fading memory filter was included in the simulation, and its respective position estimate is depicted in Figure 65.

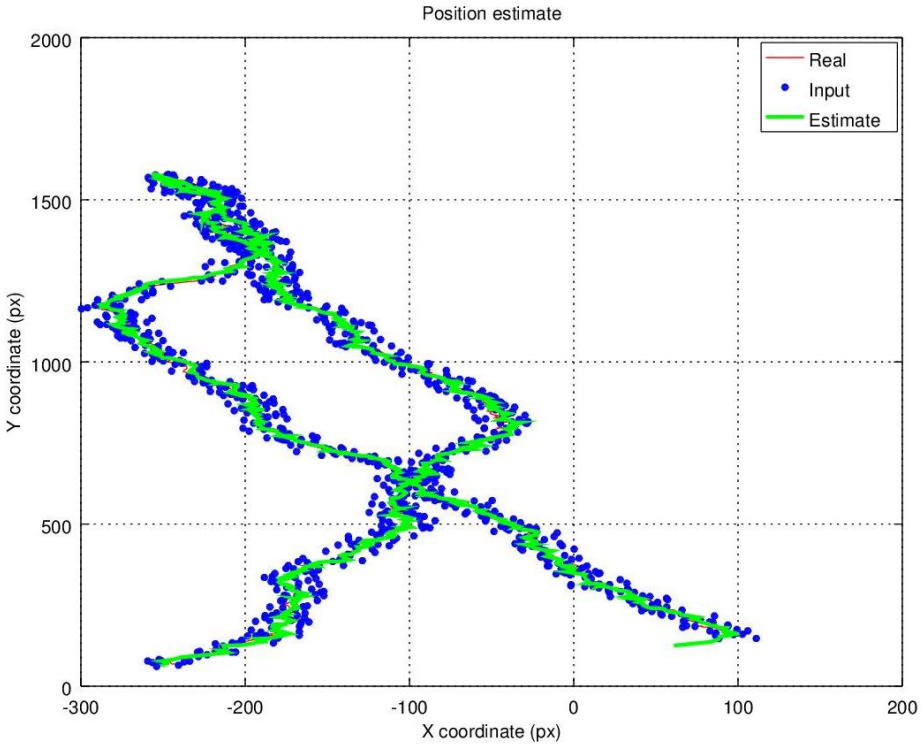


Figure 65. Position (x and y) estimate for the fading memory implementation

3.4.4 Real Data Analysis

With the parameters and additional techniques correctly implemented in the simulation environment, the next step was to process real data to fully validated the design. The following procedures were applied in order to assess how the filter performs in practice:

- Static setup: robot is always static, target is static;
- Dynamic setup: robot follows a pre-defined path, meanwhile the target is dynamic, and visible around 90% of the time.

A total of eight (8) videos were recorded, four (4) for each setup. Frames are extracted from all videos, and manually marked for the static setup scenario only, due the fact this procedure is burdensome for the other cases. The neural network is then fed with the frames (320x240 resolution), and its outputs logged. Moreover, data from the robotic system (ego-motion) is merged with the output of the neural network, for the dynamic setup scenario. Finally, the combined data is processed in MATLAB with the full Kalman filter implementation, with the position and speed estimates being plotted for each video, alongside the residual for the static setup case only (it requires the real coordinates of the target). Finally, a different video is composed and marked with both raw (neural network) input and (Kalman) filtered output in order to better observe the results - raw inputs, filtered outputs, and prediction outputs are marked with green, red, and magenta circles, respectively. The latter (prediction outputs) corresponds to the prediction of the filter when the neural network outputs a false-negative. Note that the neural network timing was respected (videos were sub-sampled: 60 fps to 20 fps/Hz), alongside the refresh rate of the ego-motion data (140 Hz), for both setups.

Static setup

For the static setup, only data from the neural network is relevant, due the fact the robotic system is not moving, thus ego-motion values correspond to zero (0). In this case, the Pre-Kalman filter module would always output pixel speeds equal to zero (0) as well, not impacting the filter. Note, however, that the prediction step is executed at 140 Hz due the fact it is dependent on this data. Only the finest two (2) results (out of 4) are presented in this section, with a sample frame extracted from both cases being shown in Figure 66. Notice both are marked with the raw input (green circle) and filtered output (red circle).

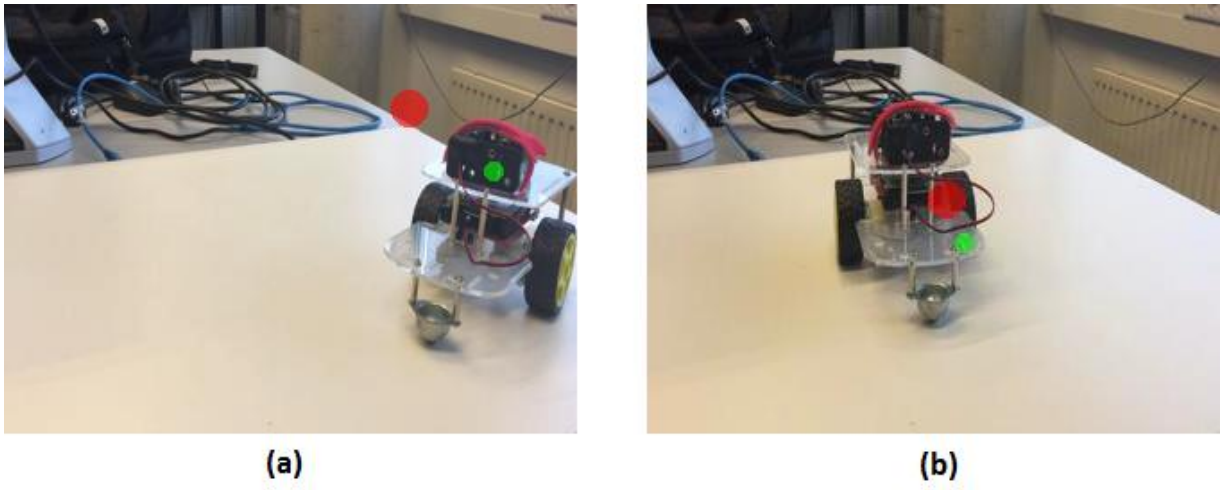


Figure 66. Marked sample frames extracted from output videos

The position and speed estimates for both cases and all iterations are shown in Figure 67 and Figure 68 below:

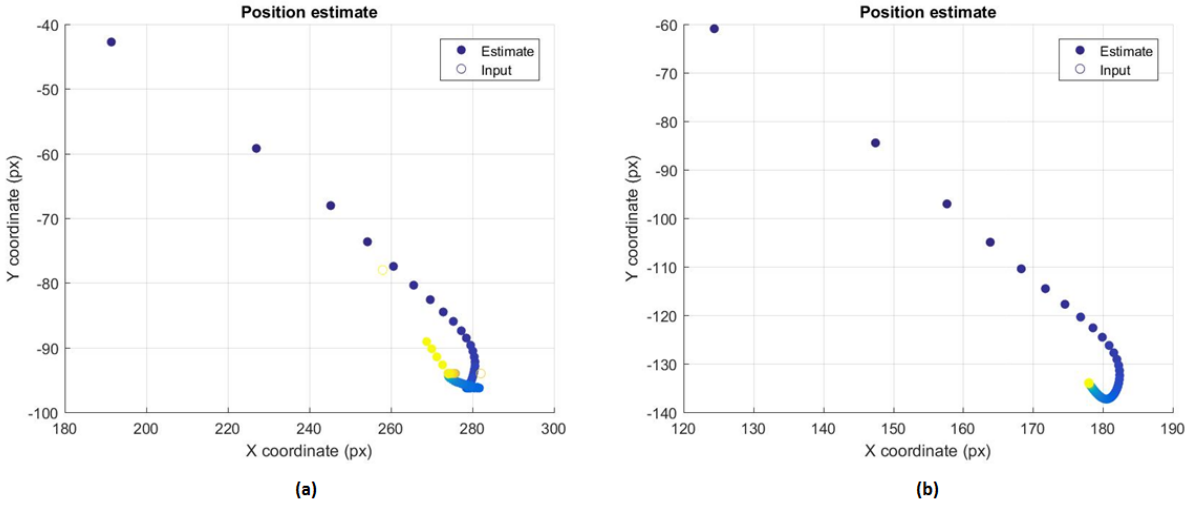


Figure 67. Position estimate for both static cases

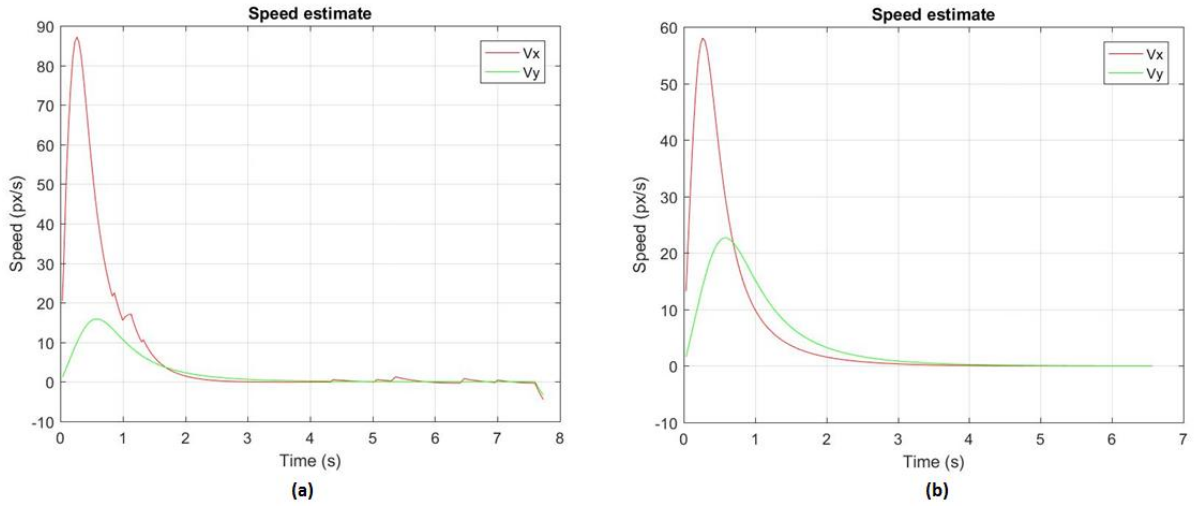


Figure 68. Speed estimate for both static cases

With respect to the graphs above, the Y coordinates were negated in order to reflect the actual pixel coordinate on the image and the estimates (filled circles) are chronologically ordered with a gradient from blue to yellow (Figure 67). Note the neural network does not deviate much, being completely stable for case (Figure 67b), thus the final speed estimates converge to zero. Moreover, both cases are rather similar, and converge to the measurements (open circles) after approximately 1 second (20 complete iterations).

Considering this setup is static, and consequently the target's position also, the residual can be further analyzed:

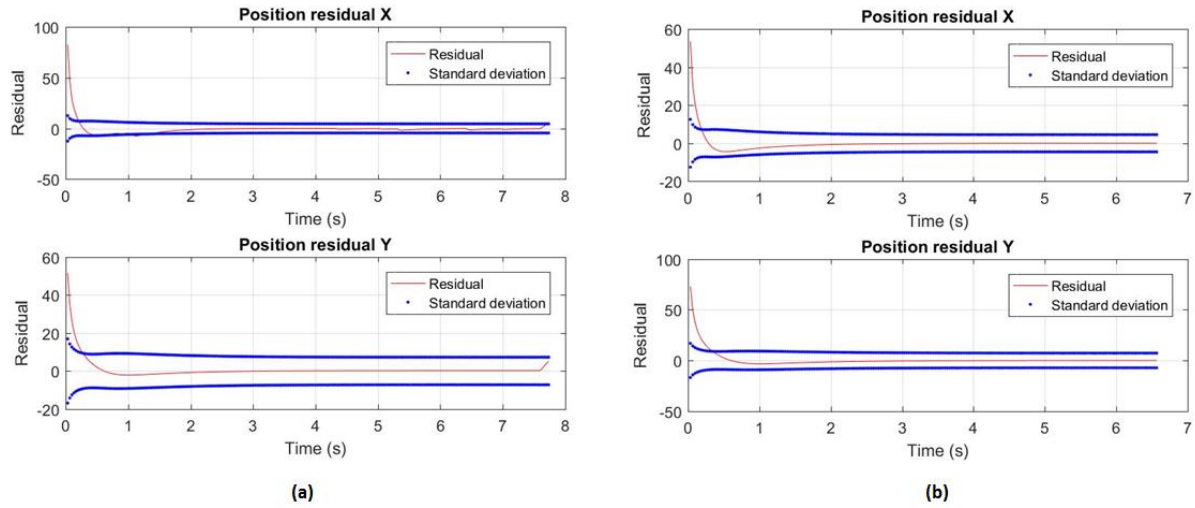


Figure 69. Position residuals for both static cases

The residuals of both cases are within $\pm 1\sigma$ after initialization, with the Y coordinate slightly outperforming X. Not only does the filter converge, but additionally presents an exceptional behavior for this setup. This setup validates the previously discussed design of the Kalman filter, although the control input is set to zero (0) and the fading memory technique is not properly tested in these cases, because the target is static. However, the adaptive filtering was applied for the simulations presented and does not degrade the performance of the filter.

Dynamic setup

Differently from the previous case, for the dynamic setup the ego-motion and neural network data must be merged, due the fact both robot and target are moving – thus, the Pre-Kalman filter module is included in the simulation. The former follows a predefined route, and the latter is moved around manually with a string attached to its front. Moreover, in these cases the neural network might not be able to identify and localize the target, due to its accuracy or occlusions that might occur for a few frames. As such cases frequently happens in reality, and in the final implementation must be treated accordingly, for now every time they occur, the update step is obviously not executed (i.e. no measurement available) but the respective output from the prediction step is stored. In fact, the final implementation will handle these cases in this way, but only for a few frames (i.e. measurements), with the filter being reset otherwise.

Only the best (and clearest) result (out of 4) is presented for this case, with a sample frame extracted from the same scenario being shown in Figure 70. Notice Figure 70b is marked with the raw input (green circle) and filtered output (red circle), meanwhile Figure 70a is marked with the prediction output (magenta).

Figure 70. Marked sample frames extracted from output video

Although only one video is processed, an additional test was performed to further explore the Pre-Kalman filter impact and response to the ego-motion. It corresponds to considering (Figure 71a) or not (Figure 71b) the ego-motion data, with the respective position and speed estimates for each case shown below:

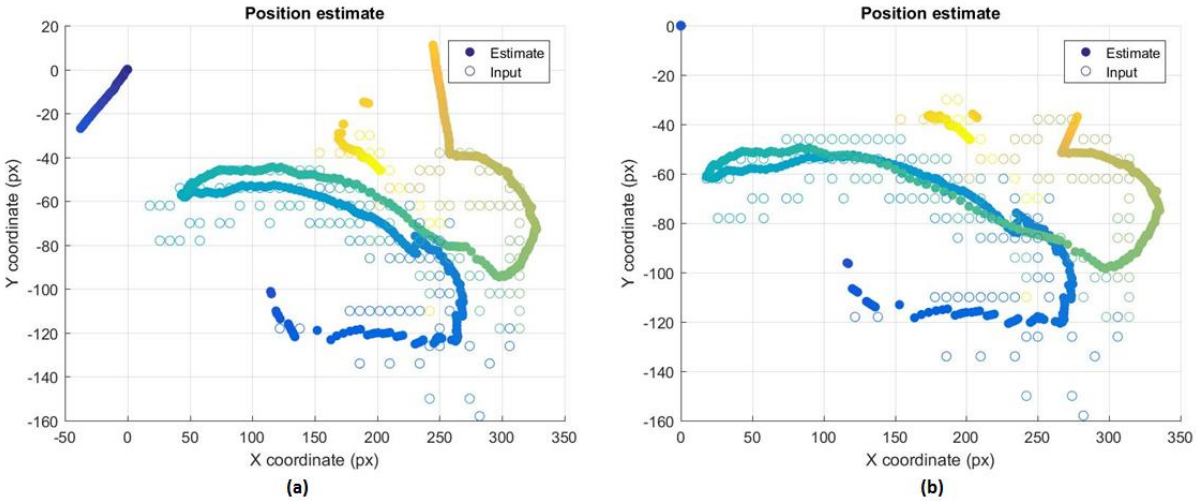


Figure 71. Position estimate for dynamic setup, considering (a) or not (b) the ego-motion

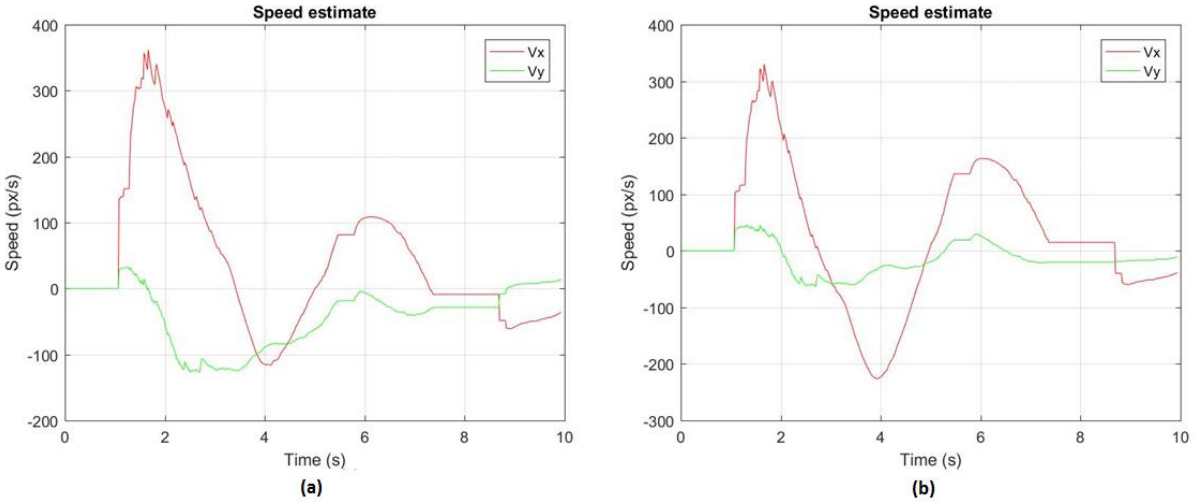


Figure 72. Speed estimate for dynamic setup, considering (a) or not (b) the ego-motion

Once more the Y coordinates were negated to reflect the actual pixel coordinate on the image and the estimates (filled circles) are chronologically ordered with a gradient from blue to yellow. On the other hand, the neural network does substantially deviate in the dynamic scenario, but is clearly smoothed by the Kalman filter. Moreover, both cases present similar results except during the first (dark blue) and last iterations (orange/yellow), due the fact the ego-motion data impacts the Kalman filter: in the beginning, the robot is moving forward, the pixel coordinates under consideration (initially 0) are expected to move diagonally, from top to bottom, to the left; in the end, the opposite happens with respect to the direction, due the fact coordinates are on the right side of the image, hence move from bottom to top (considering the image coordinate system). Finally, although not visible, the false-negatives and occlusion cases are far better predicted for the case in which the ego-motion is considered. Notice the speed estimates (Figure 72) reflect the previously discussed behavior on considering or not the control input: when the ego-motion is not considered (Figure 72b), the output of the filter intrinsically combines the target speed (in px/s) and the ego-motion impact, thus not corresponding to the actual target speed only that happens in the other case (Figure 72a).

In summary, the designed Kalman filter presented an exceptional performance, in both static and dynamic cases, including the ego-motion data. Moreover, it validates both the Pre-Kalman module and the Kalman filter itself, in addition to the fading memory technique.

3.4.5 Final Design

Final design and implementation of the Kalman filter comprises the following characteristics:

1. The 1st order Newtonian (linear) model is used;
2. The control input corresponds to the ego-motion impact, and is considered;
3. The process and measurement noise matrices are properly defined for the application;
4. Each coordinate is implemented in a different (but similar) instance of the Kalman filter that may execute in parallel;
5. The fading memory technique is used for handling abrupt changes.

Additionally, the equations of each instance of the Kalman filter (for X and Y coordinates) may be hard-coded to avoid matrix operations. Not only does it reduce complexity, but also execution time and enables tuning the fade memory algorithm applied. This section briefly presents the final equations that were implemented for the Kalman filters designed. However, in order to keep this section short, only the equations for a single instance of the filter are presented, and each matrix element is represented by M_{ij} with M being the element of the ith row and jth column:

- Predict step
  - $\begin{cases} x_{k|k-1} = x_{k-1|k-1} + \dot{x}_{k-1|k-1} * \Delta t \\ \dot{x}_{k|k-1} = \dot{x}_{k-1|k-1} \end{cases}$
  - $\begin{cases} P_{11_{k|k-1}} = {\alpha_c}^2 \left[P_{11_{k-1|k-1}} + \Delta t * \left(P_{12_{k-1|k-1}} + P_{21_{k-1|k-1}} + \Delta t * P_{22_{k-1|k-1}} \right) \right] + Q_{11_k} \\ P_{12_{k|k-1}} = {\alpha_c}^2 \left[P_{12_{k-1|k-1}} + \Delta t * P_{22_{k-1|k-1}} \right] + Q_{12_k} \\ P_{21_{k|k-1}} = {\alpha_c}^2 \left[P_{21_{k-1|k-1}} + \Delta t * P_{22_{k-1|k-1}} \right] + Q_{21_k} \\ P_{22_{k|k-1}} = {\alpha_c}^2 * P_{22_{k-1|k-1}} + Q_{22_k} \end{cases}$

- Update step
  - $S_k = P_{11_{k|k-1}} + {\sigma_{x_{sensor}}}^2$
  - $\begin{cases} K_{11_k} = \frac{P_{11_{k|k-1}}}{S_k} \\ K_{21_k} = \frac{P_{21_{k|k-1}}}{S_k} \end{cases}$
  - $\tilde{y}_k = x_{sensor} - x_{k|k-1}$
  - $\begin{cases} x_{k|k} = x_{k|k-1} + K_{11_k} * \tilde{y}_k \\ \dot{x}_{k|k} = \dot{x}_{k|k-1} + K_{21_k} * \tilde{y}_k \end{cases}$
  - $\begin{cases} P_{11_{k|k}} = P_{11_{k|k-1}} * \left(1 - K_{11_k}\right) = P_{11_{k|k-1}} - K_{11_k} * P_{11_{k|k-1}} \\ P_{12_{k|k}} = P_{12_{k|k-1}} * \left(1 - K_{11_k}\right) = P_{12_{k|k-1}} - K_{11_k} * P_{12_{k|k-1}} \\ P_{21_{k|k}} = P_{21_{k|k-1}} - K_{21_k} * P_{11_{k|k-1}} \\ P_{22_{k|k}} = P_{22_{k|k-1}} - K_{21_k} * P_{12_{k|k-1}} \end{cases}$

Notice no matrix operations are required at all by implementing the equations presented above, in addition to only two floating-point divisions being required.

3.5 Pixel Control

With the Kalman filters completely defined, tested and implemented, the next step was to address the final part of the application: given the target position is known within the image, how should the robotic system be driven in order to follow the target, ideally maintaining it in a reference position? Initially, controlling the distance between the robot and target was explored, but due the fact this would either require implementing an online camera external parameters calibration technique, or an ultrasonic sensor, another approach is introduced which abstract the distance. Moreover, the latter would likely perform poorly, as the robot and target must be correctly aligned.

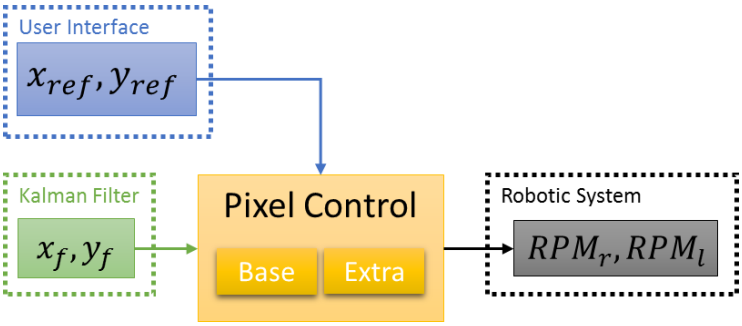


Figure 73. Pixel Control overview

The Pixel Control module presented in Figure 73 above is then explored, which implements two PID controllers: one meant for controlling the base speed for both encoder motors, meanwhile the other for the direction of the robotic system. Moreover, the controllers may run in parallel as there is no direct data dependency. Both controller's outputs are then merged, and the RPM references (right and left) are forwarded to the robotic system. Notice the filtered target position (x_f, y_f) is the input for the PID controllers, while the references (x_{ref}, y_{ref}) are provided by the user.

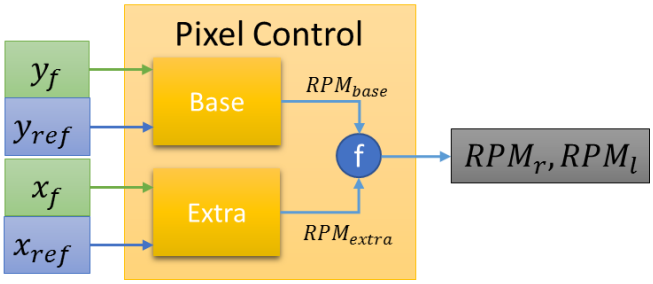


Figure 74. Detailed pixel control overview

More specifically, x_f is fed to the "Extra" PID controller, due the fact this coordinate only influences the direction, meanwhile y_f is provided to the "Base" PID controller, because it impacts the forward or backward speed, as depicted in Figure 74. The merged outputs are computed as follows:

$$\begin{cases} RPM_r = RPM_{base} + RPM_{extra} \\ RPM_l = RPM_{base} - RPM_{extra} \end{cases}$$

Note that these equations assume the "Extra" PID controller outputs $RPM_{extra} > 0$ when $x_f < x_{ref}$. After implementing both PID controllers, their respective gains were tuned in practice for a sampling

frequency $f_s = 20\ Hz$ (constrained by the neural network), with the proportional (K_p), integral (T_i) and derivative (T_d) gains presented in Table 5 below.

Table 5. Gains of the base and extra PID controllers

PID Controller	Gain		
	Proportional	Integral	Derivative
Base	0.4	100	0.00001
Extra	0.1	100	0.001

In order to further analyze the behavior of the controllers implemented, their respective response for the reference sequences $x_{ref} = [160, 240, 160, 80]$ and $y_{ref} = [96, 80, 120, 80]$ are shown below:

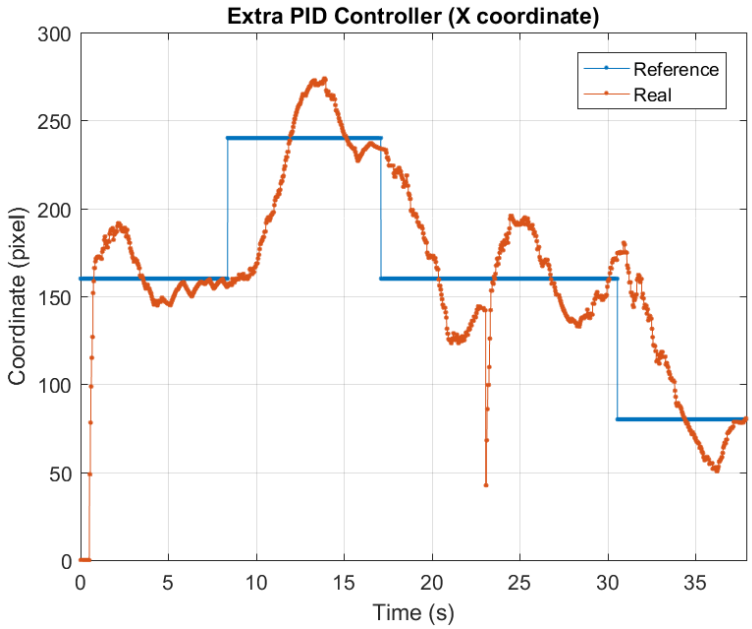


Figure 75. Extra PID response (X coordinate)

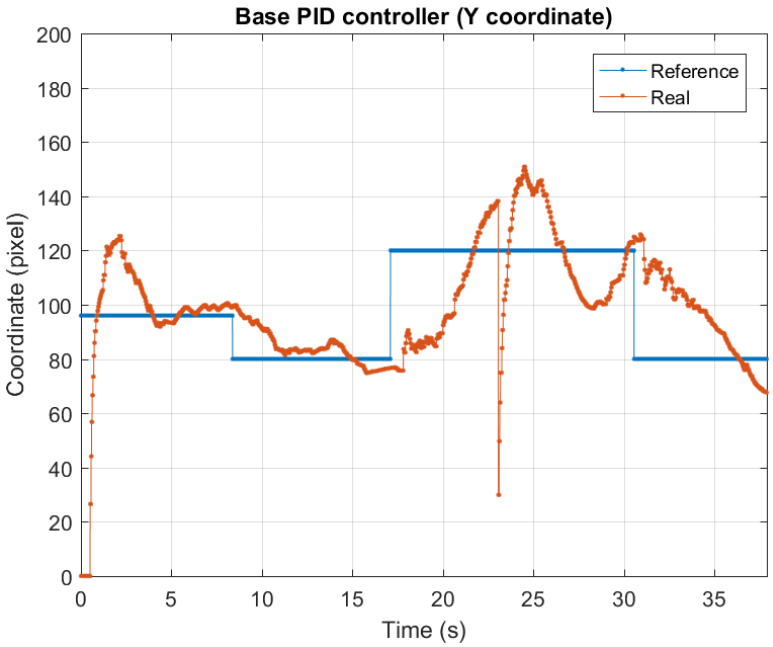


Figure 76. Base PID response (Y coordinate)

The discontinuity ($t \approx 23s$) in both graphs above is due to the Kalman filter being reset during normal operation, caused by consecutive false negatives outputted by the neural network. Furthermore, notice there is always a small delay in the beginning and whenever the references are changed, mainly due the wireless transmission which will be further analyzed. Although the PID controllers were tuned in practice and are stable, both present an oscillatory behavior in addition to overshoot, with the system taking roughly 5 seconds to come closer to the reference values. It is relevant to note the base controller slightly outperforms the extra controller, hence rotating the robot is harder than translating it – which is expected. Finally, both controllers should be further tuned to increase their performance, especially if the delay is reduced. Figure 77 below depicts the system response in terms of both X and Y (2D) coordinates:

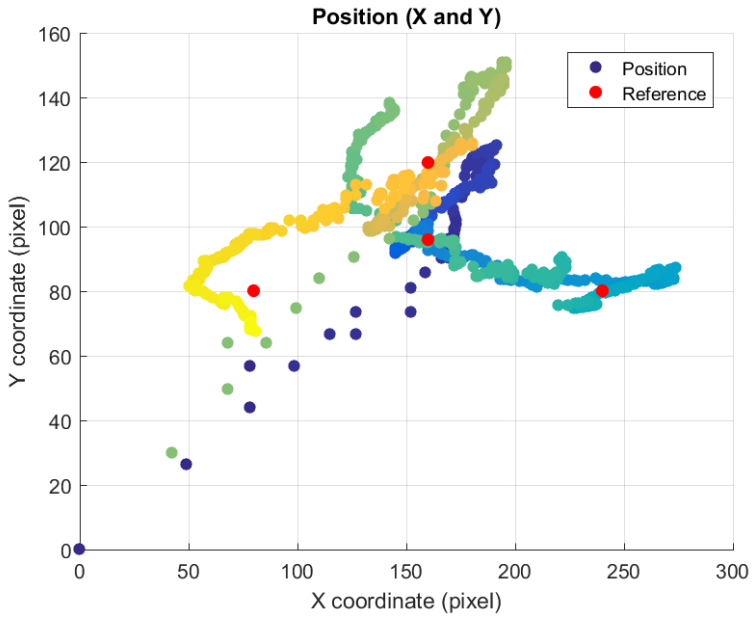


Figure 77. System response in terms of coordinates

In summary, the proposed module is capable of abstracting the real distance between robot and target, in addition to properly driving the robotic system in order to keep the target around the desired reference coordinates. However, it is relevant to notice that the 0.5 seconds delay of the stream directly impacts the overall responsiveness of the system, hence both controllers could be greatly improved in the future and react faster. This issue is further addressed in Chapter 4. Additional information about the PID controller, its implementation and tuning procedure can be found in Appendix A: PID Controller.

3.6 Final Implementation

Considering all components of the tracking system were properly defined and tested, combining them is necessary for the final implementation. As the initial goal was to interface the camera and execute the tracking in an embedded system, software components were developed in C/C++ and tested in an embedded Linux during development (Zynq). However, executing this application fully in the embedded platform requires the neural network is already implemented in hardware, which was not the case in the end of this project. Hence, the application was ported to a PC running Linux, but

remained compatible and can be executed in an embedded Linux. This section presents an overview of the complete tracking system in terms of hardware and software, and relevant topics will be further explored. Figure 78 below depicts the final setup, with the tracking system running on a PC, connected via USB to the robotic system, and via Wi-Fi to the camera:

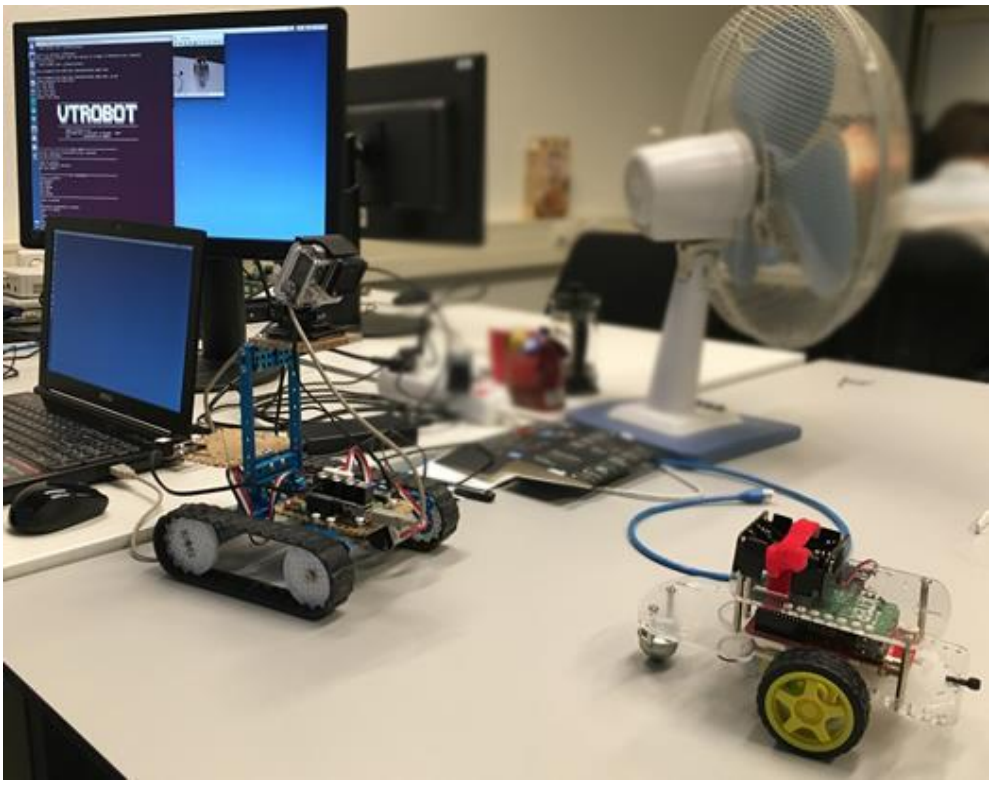


Figure 78. Final implementation with target and both robotic and tracking systems running

3.6.1 Hardware

The tracking system is fairly simple in terms of hardware, and includes the main processor, a camera connected to it, and a serial communication to the robotic system. Specifically for the final implementation, the main controller is a PC running Linux (Ubuntu 14.04) in a Intel Core® i7 processor, connected to the GoPro Hero4 via a direct Wi-Fi connection (access point), and exchanges data with the robotic system through an USB-to-UART connection, with the UART communication complying to the μprotocol discussed in section 2.5. The overview of the tracking system is depicted in Figure 79 below:

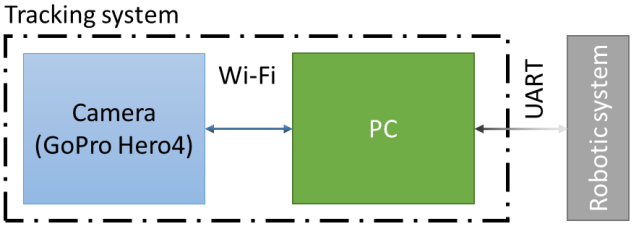


Figure 79. Final hardware architecture of the tracking system

Although the main processor is currently a PC, and the camera is connected via Wi-Fi, these parts are easily changeable. For instance, one might use an embedded platform (i.e. Zynq) instead, and connect the camera via HDMI, without changing the functionality of the tracking system.

3.6.2 Software

The tracking system, in comparison to the robotic system, has a much more complex software architecture, mainly due the Wi-Fi connection needed – the neural network is simple. The fundamental functions had to be analyzed to be able to define tasks and segregate them into other applications, if needed. In terms of activities this system has to:

- Handle the camera stream;
- Process each frame of the stream with the neural networks (detection & localization, must be in Python using TensorFlow[50]);
- Display a marked image to the user, with the filtered coordinates of the target;
- Handle data exchange between robotic and tracking systems;
- Implement the Pre-Kalman filter;
- Implement two (2) Kalman filters, one for each coordinate;
- Implement two (2) PID controllers;
- Provide a user interface in order to manually set the reference coordinates;
- Coordinate, merge, and forward data accordingly.

Taking these activities into consideration and after several discussions and different implementations, three (3) applications are necessary:

1. The main application (MAPP): responsible for the Pre-Kalman filter, Kalman filters, PID controllers, (simplistic) user interface, and communication with the robotic system;
2. The GoPro/Neural Network (GPNN): start and keep the camera stream alive, in addition to running the neural network and displaying a marked image with the current target position;
3. The stream capture: responsible for retrieving, resizing and saving in a file the frames of the camera stream.

Not only do all applications execute concurrently, but also exchange data directly (i.e. message queues) and indirectly (i.e. files). In general terms, the Stream Capture captures frames from the camera stream, which are saved in a file. The GPNN then opens the image, fed it to the neural network(s) and sends the raw coordinates (x_{raw}, y_{raw}) of the target through a message queue to the MAPP, which process the raw coordinates and send back to the GPNN the filtered coordinates (x_f, y_f). The overview of the applications and their relations is presented below:

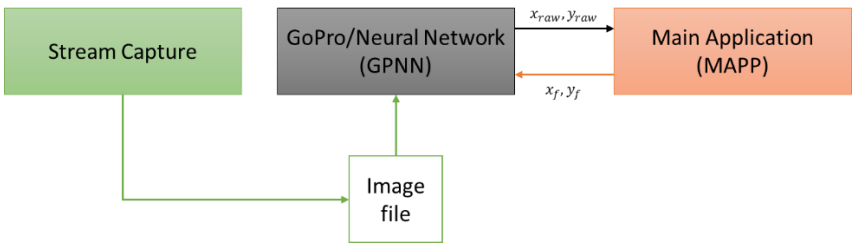


Figure 80.Overview of the applications and their relations

In order to simplify usage, the user should only run the MAPP, which is responsible for starting the GPNN programmatically, which in turn starts the Stream Capture. Consequently, whenever the main application is terminated, the others also are. Moreover, each application is implemented in different

programming languages: Stream Capture is a complete third-party application, which actually uses the *ffmpeg*; GPNN was developed in Python and; MAPP in C/C++.

Initially, the stream capture is briefly discussed due the fact it is the simplest one, and corresponds to an *ffmpeg* application, launched by GPNN with a single bash command and necessary arguments. The framerate is limited to 20 fps, which matches the neural network requirements. A single iteration of the stream capture comprises retrieving a frame from the camera stream, resizing it (from 640x480 to 320x240 resolution), and saving the resulting image in a file, as depicted in Figure 81 below.

Figure 81. Simplified single execution of the stream capture application

The GPNN initially elevates its priority (if permitted) to the maximum (-20) in order to reduce delay (it runs in a PC), and has three (3) different threads, one responsible for handling the camera's stream (i.e. starting and maintaining it), another dedicated to the neural networks (detection and localization, if necessary), and one more dedicated to process data incoming from the MAPP which will not be discussed. The former starts the camera stream by sending a specific HTTP request, then launches the stream capture (*ffmpeg*) application, signals to the neural network thread it has done the previous steps and loops forever sending keep alive (UDP) packets to the camera every 2.5 seconds, in order to maintain the stream. The overview of its execution is shown in Figure 82.

Figure 82. Execution flow of the camera handler thread of GPNN

The other thread responsible for applying the neural networks comprises more steps. After initializing the models of the neural networks, it blocks until a signal from the camera handler thread is received. The neural network handler then opens the image file (saved by the stream capture application), deletes the file and resizes the image to 224x224 resolution which is required by the detection network. Moreover, the resized image is fed to the detection network, and if the target is detected, the localization network is fed with the original (320x240 resolution) frame. Subsequently, the raw coordinates are sent to the MAPP, and the current frame marked and displayed. In case the target is not localized, a negative value is sent to the MAPP, indicating so, and the frame is not marked (only displayed). The overview of the execution of the neural network handler thread after initialization is depicted in Figure 83, and a sample marked frame in Figure 84, with the green and yellow dot corresponding to the neural network raw and filtered outputs, respectively.

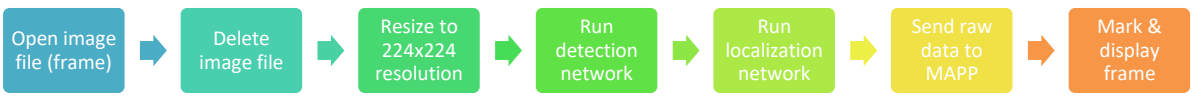


Figure 83. Execution flow of the neural network handler of GPNN

Figure 84. Sample marked frame output of the GPNN

Main Application (MAPP)

The main application (MAPP) is the most complex piece of software of the system, being responsible of launching and communicating with the GPNN, handling the communication with the robotic system, providing a simple user interface, executing the Pre-Kalman filter, two Kalman filters, and two PID controllers. In terms of implementation, specific drivers were developed and tested for the Kalman filters and PID controllers, in addition to the communication with the robotic system. Main functionalities were segregated into threads, and in total nine (POSIX) threads were used and are summarized below with their respective functionalities:

- NN Handler: Handles incoming data from the concurrent python application. Data is then forwarded to the Coordinator;
- MegaPi Handler: Handles all incoming/outgoing data from/to the MegaPi. It forwards data sensor to the Pre-KF and set the respective RPMs received from the Coordinator;
- Pre-KF: Process sensor data from MegaPi Handler, computing the ego-motion impact on the object being tracked. The pixel of interest is retrieved from the Kalman filters (KFx and KFy), and both pixel speeds are forwarded to the respective Kalman filter
- KFx and KFy: Kalman filter implementations for X and Y coordinates, respectively. Data coming from Pre-KF is integrated in the update step, and prediction is performed when data is received from the Coordinator. The filters' outputs are sent to the PID controllers (bPID or ePID);
- bPID and ePID: PID controllers for pixel distance control, with former being responsible for computing the (b)ase speed which depends on the object'ś Y coordinate. The latter computes the (e)xtra speed in order to set the robot to the correct direction which depends on the object's X coordinate. Both controllers send their output (RPM) to the Coordinator which is computed based on the provided reference(s) from the Coordinator (which might come from the User Interface);
- Coordinator: coordinator of the system, which makes high-level decisions and manages the whole operation. Retrieves, processes, and send data to relevant threads;
- User Interface: Provides a simple menu for the user, which can set the reference coordinates for the PID controllers. New references are forwarded to the Coordinator, and subsequently to the controllers.

The overview of the software architecture is shown in Figure 85 below, where each box represents a thread (except the gray boxes), and arrows corresponds to message queues between them (except the connection to the robotic system).

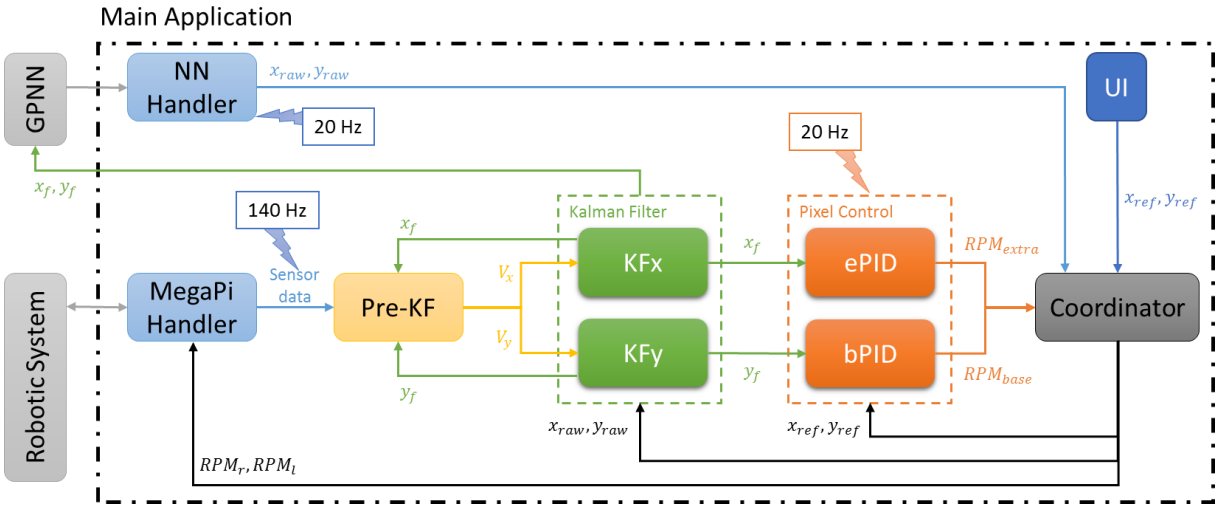


Figure 85. MAPP software architecture

In summary, the MegaPi Handler forwards sensor data (current speed and angular velocities of the robotic system) to the Pre-KF thread, which derives both pixel speeds based on the ego-motion data, and the coordinates of the pixel of interest outputted by the Kalman filters. The latter, on the other hand, apply the designed Kalman filters and either predicts, updates, or both: whenever data (i.e. control input) is received from the Pre-KF, it performs the prediction step (should occur at 140 Hz); whenever measurement data is received from the Coordinator (coming from the NN Handler), it updates (should occur at 20 Hz) and forwards its output to the respective PID controller in addition to the GPNN application, in order to mark the image and display to the user. The controllers execute periodically, and initially check for new references from the Coordinator (coming from the UI), subsequently actuating, running the PID loop and forwarding their outputs. The Coordinator merges the RPMs outputted by the controllers and forwards to the MegaPi Handler (thus, to the robotic system itself), check for new references coming from the user interface and transmits them to the controllers, if necessary. Moreover, it retrieves the neural network raw data (should occur at 20 Hz) and delivers it to the respective Kalman filters. The user can set new reference values through the menu provided by the UI, by either manually inputting X and Y, or selecting a standard option, as depicted in Figure 86 below, where banner “VTROBOT” stands for “Visual Tracking Robot”, a nickname given to the project during development. Note that in total the application requires about 10 to 15 seconds to initialize and normally execute, mainly due the time the camera stream takes to start.

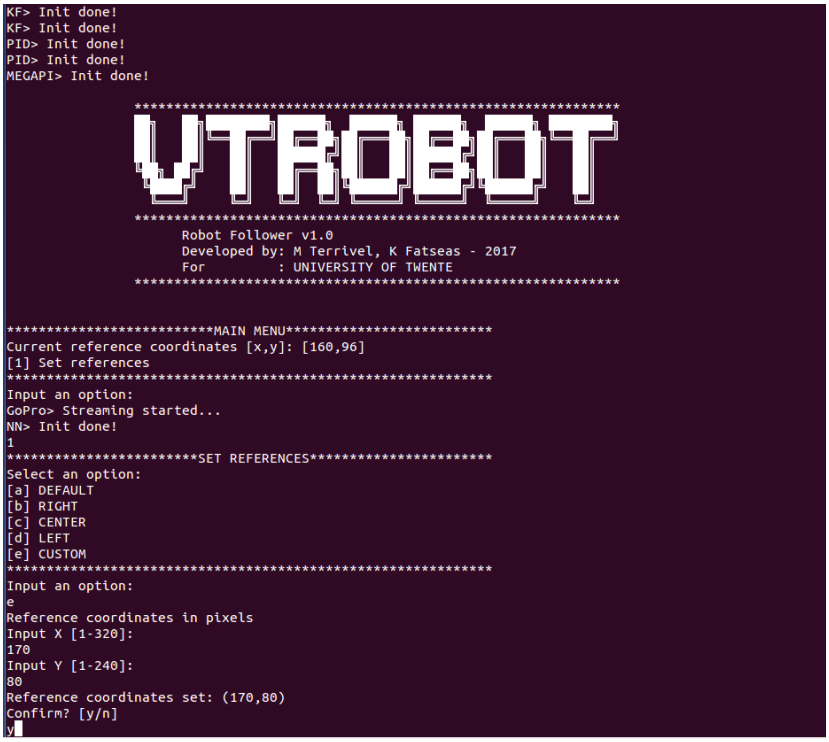


Figure 86. Banner and menu provided by the UI

This process also increases its priority to the maximum (-20), in order to reduce any delays or jitters, in addition to launching the GPNN and registering termination signals to gracefully shutdown. Initialization (main function) follows the procedure depicted in Figure 87 below:

Figure 87. Initialization procedure of the MAPP

It is relevant to state the Coordinator is responsible for handling false positives of the neural network, resetting the Kalman filter if necessary, alongside the PID controllers. In order to address this case, a maximum number of false negatives is defined, and correspond to $35\% * 20\ fps = 7$, or 35% of the frame rate (20 fps). Note this value was initially set to 10% (considering the neural network's accuracy is 90%) of the frame rate, but later modified to the aforementioned value due the fact the neural network performed worse in practice. Whenever a negative value is received by the coordinator (i.e. no car detected by the GPNN), a counter is incremented, and if it has reached the maximum number of false negatives, both the PID controllers and Kalman filters are reset. Otherwise, the Kalman filters keep performing the prediction step, thus the target can still be tracked up to 7 consecutive false negatives.

Communication between (POSIX) threads are implemented through (POSIX) message queues. On the other hand, tasks are simpler and thus are agglomerated in the *main* C++ source file. In terms of priorities, all threads are considered critical, hence have the same priority (1), except the user interface which is not essential and is set to priority 0. All message queues have a max number of messages (i.e. tokens) set to 1, due the fact the threads read and process them as fast as possible,

although this is not guaranteed for every iteration because the application is running in a PC. Moreover, each task executes in a cooperative manner, suspending themselves for a brief period (1-100 microseconds) or blocking when sending/receiving a message, if synchronization is necessary.

Both handlers use low-level drivers to communicate with the other systems, and lightning bolts in Figure 85 correspond to periodic tasks, triggered by Linux (real-time) timers with a 0.1 millisecond accuracy, enough for the refresh rates used (20 and 140 Hz). Moreover, notice the MegaPi Handler only delivers sensor data at 140 Hz, but do not send data to the robotic system at this rate – data is sent based on the PID controllers rate (20 Hz), which are merged by the Coordinator. Finally, it is important to note that one (1) serial communication (USB-to-UART) is used by the MegaPi Handler, namely USB0 (/dev/ttyUSB0), for communicating with the robotic system. Note, however, that the device name might change, and should be modified accordingly in the *main.c* file – if the Zynq platform is used, for instance, the serial port could be UART1 (/dev/ttyPS1).

Although each thread was not addressed singularly, major aspects of the architecture were explained, and relevant code snippets can be found in Appendix C: Embedded Software.

3.6.3 Drivers

For the tracking system only three (3) drivers are currently needed: one for communication purposes, more specifically to communicate with the robotic system (namely *megapi*), and other two for the PID controller (namely *pid*) and Kalman filter (namely *kf*) implementations. All drivers were segregated in header and source files in which specifics can be easily modified in order to comply with future modifications. Implementation details will not be discussed, although code is fully commented and analyzing it should be simple.

MegaPi

The driver developed for communicating with the robotic system applies simple serial communication principles. To correctly send or retrieve data, the driver must be initialized with the corresponding serial port the robotic system is connected to – currently this is USB0, corresponding to "/dev/ttyUSB0" –, which has to be visible in the Linux machine. The configuration structure is defined as follows:

```
1.  // MegaPi config structure
2.  typedef struct megapi_t
3.  {
4.      int fd;                 // File descriptor for tty port
5.      bool init;              // Init flag
6.  } megapi_t;
```

The synchronization words used by the protocol are defined in the source file, with other relevant defines: data packet sizes, baud rate, among others. Its usage is simple, with the summary of the user functions explained in Table 6. This driver is used specifically by the MegaPi Handler thread of the MAPP for exchanging data with robotic system.

Table 6. User functions of the MegaPi driver

Function call	Explanation
bool megapi_init(megapi_t *mp, char *port)	Initialize driver and handler
bool megapi_setRPM(megapi_t *mp, RPM_t *rpms)	Send RPM references to the module – both right and left RPMs
bool megapi_getSensorData(megapi_t *mp, sensorData_t *data);	Get sensor data from module – latest speed and angular velocities

After being correctly initialized with the *megapi_init(…)* function, the link is established and ready to be used via the other functions which can send new RPM references or retrieve the latest sensor data from the robotic control system.

PID controller

The PID controller was also implemented, and comprises many functionalities such as defining minimum and maximum input and output values, minimum error, and offset for the output (in case a feed-forward is needed). Only the former is actually used in the final implementation to have a complete security layer, for both inputs and outputs of the controller. Moreover, the user is able to set a constant sampling frequency (standard behavior) or dynamically compute the time between loop calls instead. The configuration structure is defined as follows:

```
1.  // PID config structure
2.  typedef struct pidController_t
3.  {
4.      // Gains
5.      float kp;               // Proportional
6.      float ti;               // Integral (1/ki)
7.      float td;               // Derivative (1/kd)
8.      // Error
9.      float error[3];         // Last 3 errors are needed [e(i) e(i-1) e(i-2)]
10.     float minError;         // Minimum error
11.     // Output
12.     float output[2];        // Last 2 outputs are neede [out(i) out(i-1)]
13.     // Output with offset
14.     float offOut;           // Only used for feed-forward control
15.     // Time-related
16.     float dT;               // Period, in seconds
17.     float dT0;
18.     // Security-related
19.     float minInp, maxInp;   // Minimum/maximum input values
20.     float minOut, maxOut;   // Minimum/maximum input values
21.
22.     // Others
23.     struct timespec sT;     // Start & end time structures for computing dT
24.     bool init;              // Init flag
25.     bool first;             // First iteration flag
26.     bool secInp, secOut;    // Apply security values for inputs/outputs?
27.     bool useMinError;       // Use max error?
28. } pidController_t;
```

Its usage is simple, with the summary of the user functions explained in Table 7. This driver is used specifically by the ePID and bPID threads of the MAPP for determining the speed and direction of the robotic system.

Table 7. User functions of the PID driver

Function call	Explanation
bool pid_init(pidController_t *pid, float kp, float ti, float td, float dT0)	Initialize controller and handler
bool pid_setInputs(pidController_t *pid, float minVal, float maxVal)	Set and enable security layer for input values
bool pid_setOutputs(pidController_t *pid, float minVal, float maxVal)	Set and enable security layer for output values
bool pid_setError(pidController_t *pid, float minVal)	Set and enable minimum error value
bool pid_loop(pidController_t *pid, float ref, float sensor, bool useDt0)	Control loop itself, which should be called periodically
bool pid_reset(pidController_t *pid)	Reset controller

After being correctly initialized with the *pid_init(…)* function, the controller is ready to be used via the *pid_loop(…)* function. Other functionalities are optional, but are available through the other functions. Details about the PID controller itself can be found in Appendix A: PID Controller

Kalman Filter

Finally, the last driver addressed is the Kalman filter implementation, which is customized due the fact no matrix operations are needed – each matrix element is singularly computed instead. Moreover, this implementation is capable of dynamically determining the time between prediction steps, as well as being reset whenever needed. The configuration structure is defined as follows:

```
1.  // Filter config structure
2.  typedef struct kf_t
3.  {
4.      /* Details on model:
5.       * x^ = | x |   F = | 1 dT |        Q = | dT^4/4 dT^3/2 |*sigma_process^2
6.       *      | x.|       | 0  1 |            | dT^3/2 dT^2   |
7.       * P0 = | sigma_x^2          0 |    K = | K1 |
8.       *      |          0 sigma_x.^2 |        | K2 |
9.       *  B = | dT |      u = vr (self-speed)
10.      *      |  0 |      H = | 1 0 |     R = sigma_measurement^2
11.      */
12.     // Main variables
13.     float alpha2;           // Fading Memory filter, default = 1.00 - this is ALPHA^2
14.     float pos, spd;         // Position/Speed (x_hat)
15.     Q_t Q;                  // Process noise covariance matrix
16.     P_t P;                  // Error covariance matrix
17.     float S;                // Innovation covariance matrix
18.     K_t K;                  // Kalman gains
19.     float dT;               // Delta time between samples
20.
21.     // Variances (std = sqrt(var))
22.     float proVar;           // Process variance
23.     float posVar, spdVar;   // Position/Speed variances (initial guess only)
24.     float senVar;           // Sensor (measurment) variance
25.
26.     // Others
27.     struct timespec sT;     // Start & end time structures for computing dT
28.     bool idle;              // Enable flag for timing purposes
29.     bool init;              // Init flag
30.     bool first;             // First iteration flag
31.     float pos0, spd0;       // Aux for reset
32.     float dT0;
33. } kf_t;
```

Notice the model is fully detailed in the beginning. Its usage is simple, with the summary of the user functions explained in Table 8. This driver is used specifically by the KFx and KFy threads of the MAPP for filtering the neural network's output.

Table 8. User functions of the PID driver

Function call	Explanation
bool kf_init(kf_t *kf, float alp, float p0, float v0, float proStd, float posStd, float spdStd, float senStd, float dT0, bool start)	Initialize filter, handler, and optionally the filter itself
bool kf_set(kf_t *kf, bool enable)	Enable or disable the filter (timing purposes)
bool kf_predict(kf_t *kf, float vr, bool useDt0)	Perform the prediction step
bool kf_update(kf_t *kf, float sensorVal)	Perform the update step
bool kf_reset(kf_t *kf);	Reset filter

After being correctly initialized with the *kf_init(…)* function, the filter is ready to be used via the other functions. Further details about the predict and update steps can be found in Appendix C: Embedded Software.

4. System analysis

Until this moment, both robotic and tracking systems have been extensively detailed. However, the complete system will be analyzed in this chapter, with the focus being in two relevant aspects: real-time behavior and delay impact on control performance. The former refers to modelling the current software implementation with simple assumptions, such that real-time tools can be used in the future to determine bounds on the throughput, prior to actual deployment of the system. The latter, on the other hand, addresses the delay in the control loop and its impact on the control behavior. Focus will be given to the delay impact, due the fact the model considered for the real-time analysis is rather simple. Moreover, as the current implementation runs on a Linux PC, there are no tight timing guarantees possible.

4.1 Delay impact

Although the final application yielded reasonable results, it is rather slow due to, mainly, the following facts:

1. The neural network is implemented in software, and is capable of running at, at most, 25 Hz (20 Hz was used instead), hence is the bottleneck of the system;
2. The camera streams over Wi-Fi, which results in a 0.5 seconds delay corresponding to the time between the time each frame is captured by the camera and sent to the neural network executed on a PC – namely, the (camera) stream delay;
3. Data exchange between tracking system (i.e. PC) and robotic system introduce delays to the response of the latter.

Considering that increasing the frequency in order to address (1) is currently not a possibility, and that (3) corresponds to a much smaller delay with respect to item (2), only the latter will be analyzed in more detail. Moreover, the largest delay of the system is due to the stream delay, which greatly affects the overall responsiveness of the system. Initially, we model the final setup in Simulink considering only the extra PID controller, a 1st order plant, and the delay itself, as depicted in Figure 88 below. It is reasonable to consider the plant as a 1st order system, due the fact the interest is on the delay impact itself.

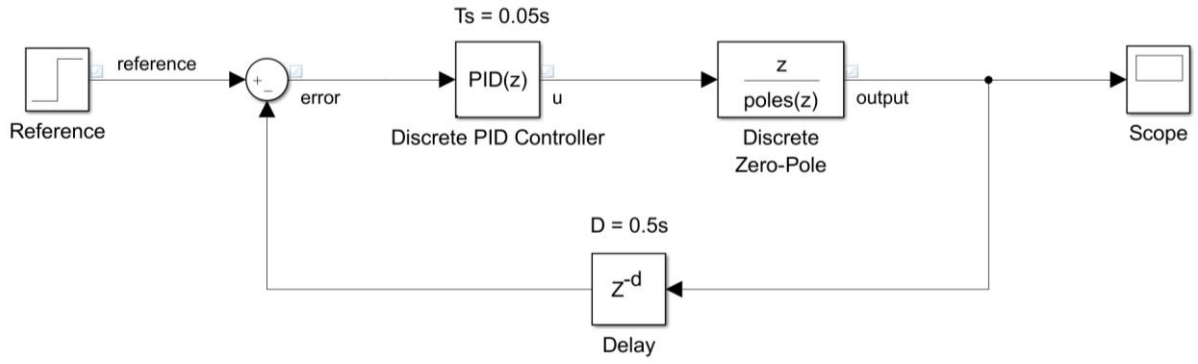


Figure 88. Simulink model for delay analysis, with $D = 0.5s$ and $T_s = 0.05s$

Moreover, notice the model under analysis is discrete, with the sampling period, delay, and (extra) PID gains analogous to the values used in the practical setup: $T_s = 0.05s$, $D = 0.5s$ and $K_p = 0.1$,

$T_i = 100$, $T_d = 0.001$, respectively. The discrete PID controller, plant, and delay formulas are presented below, correspondingly:

$$K_p + \frac{T_s}{T_i} * \frac{1}{z-1} + T_d * \frac{1}{1 + \frac{T_s}{z-1}} \tag{4.1}$$

$$\frac{z}{z - e^{-T_s}} \tag{4.2}$$

$$\frac{1}{z^{D/T_s}} \tag{4.3}$$

Note that (4.1), (4.2) and (4.3) are in the Z-domain (discrete system), with the plant being modeled as a 1st order system [51]. Moreover, the delay is presented in terms of the delay time D and sampling period T_s, both in seconds. Considering the values previously presented for the (extra) PID controller gains, sampling and delay times, the above equations are reduced to, respectively:

$$0.1 + \frac{0.0005}{z-1} + \frac{0.001}{1 + \frac{0.05}{z-1}} \tag{4.4}$$

$$\frac{z}{z - e^{-0.05}} \cong \frac{z}{z - 0.9512} \tag{4.5}$$

$$\frac{1}{z^{0.5/0.05}} = \frac{1}{z^{10}} \tag{4.6}$$

Based on (4.4), (4.5) and (4.6), one is already able to reason about how the delay impacts the system: poles are added to the control system due the D/T_s ratio, direct impacting the stability. Notice that for a virtually zero delay ($D = 0$), no poles are added at all. The model presented in Figure 88 considering equations presented above was then simulated, for two cases: with ($D = 0.5s$) and without ($D = 0s$) delay. Their step response and root locus are presented in Figures 89 and 90 below, respectively. Notice a discrete system is stable when all poles and zeros are within the unit circle.

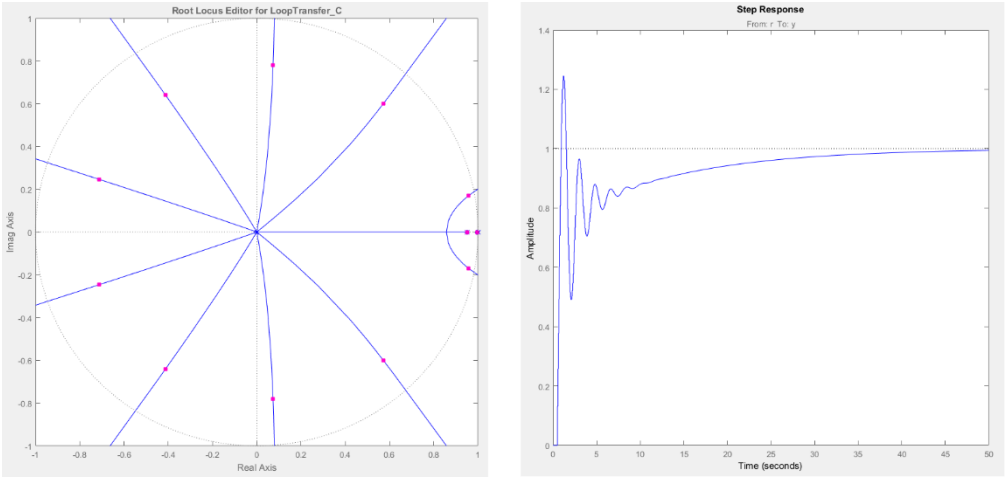


Figure 89. Root locus (a) and step response (b) for $D = 0.5s$

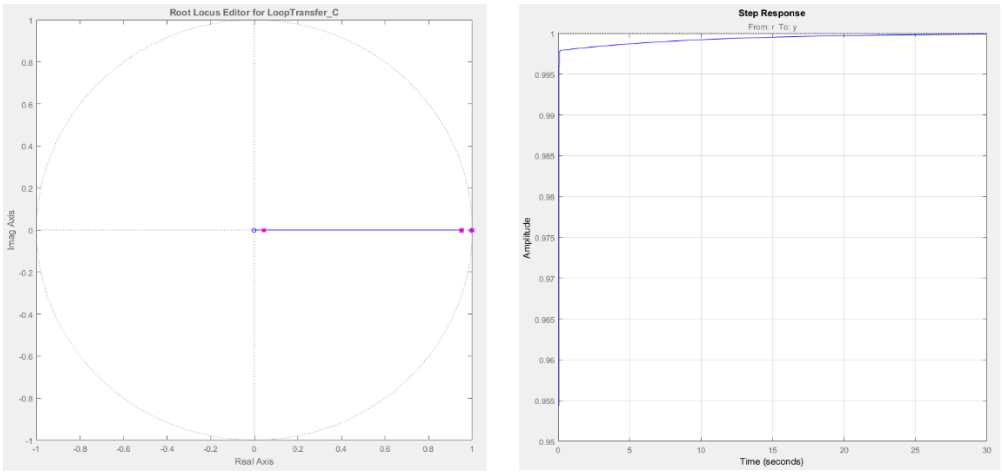


Figure 90. Root locus (a) and step response (b) for $D = 0s$

By comparing both root locus presented above, it becomes clear the delay indeed adds poles to the control system. Not only does this constrain stability, but also it increases system analysis' complexity, and tuning is restricted due the fact the system might become unstable. When delay is negligible, one has more flexibility while tuning the system, which is typically stable: zeros and poles are within the unit circle (Figure 90a). In terms of response, an oscillatory behavior followed by an exponential envelope is observed for $D = 0.5s$ (Figure 89b), which results in a disappointingly slow response – resembling the effects of lowering the sampling frequency of the system. Finally, by tuning a (proportional) compensator for no delay, it is possible to significantly decrease the settling time of its step response ($Gain = 200$ in Figure 90b). For the case in which delay is present, on the other hand, tuning is more constrained, with the system becoming unstable for $Gain \geq 1.9$ ($Gain = 1.8$ in Figure 89b). Overall, this initial analysis shows the delay considered is significant, and negatively impacts system design, stability and response.

In order to further investigate how the delay impacts the system, the aforementioned model is further refined and presented in Figure 91 below:

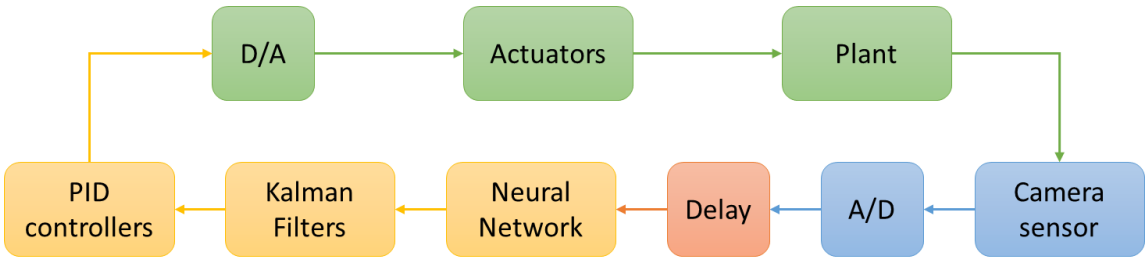


Figure 91. Model used for analyzing the delay impact

Moreover, the model above was implemented in MATLAB and simulated while taking into account the following:

- Both coordinates are considered. Hence, two Kalman filters and two PID controllers are instantiated;
- Image dimensions are respected (320x240);
- The target remains static;

- Whenever possible, noises are added and correspond to values used during the implementation of the project;
- PID controllers use the same implementation as the real setup, including the gains;
- Kalman filters use the same implementation as the real setup;
- The plant makes use of the Pre-Kalman filter equations, which is possible due the PID controllers output RPM and the angular velocity can be derived as well;
- Kalman filters, PID controllers and Neural Network run at $20\ Hz\ \pm 10\%$;
- An initial position is set for the target, and for every iteration a new position is computed based on the previous one, augmented by the pixel speeds derived by the Pre-Kalman equations multiplied by the sampling time ($0.5s\ \pm 10\%$). The new position corresponds to the neural network output;
- The delay corresponds to buffering the neural network's output in a FIFO manner. Moreover, it is parametrized in the implementation in order to be easily modifiable;
- Preferably, the references for the PID controllers must be changed a couple of times.

The model presented in Figure 91 was implemented considering all aforementioned points, and the results for an initial position $(x_0, y_0) = (220, 100)$, and a delay $D = 0.5\ s$ are presented in Figure 89 below (for 1000 iterations):

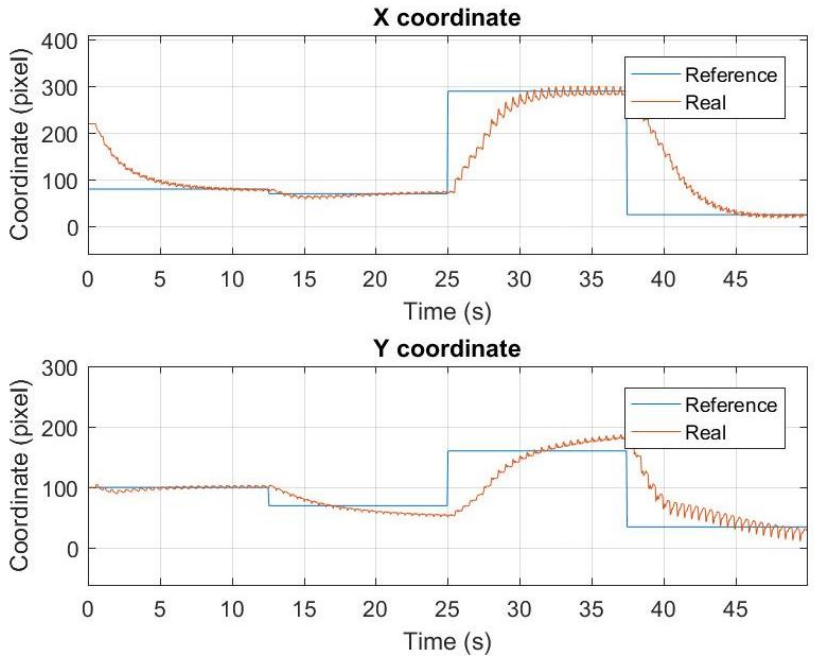


Figure 92. Model results for $D = 0.5s$

Notice that the references were changed four (4) times in total, and correspond to the following sequence for the X coordinate: 80, 70, 290, 25. The Y coordinate, on the other hand, follows the sequence: 100, 70, 160, 35. More importantly, the observed settling time to an output value equal to 95% of the reference value corresponds to about 5 seconds. Besides, the oscillation observed on the "real" coordinate values (more visible for $t \geq 25s$) is mainly due the sensor standard deviation (i.e. neural network deviation) being included in the new position computation for every iteration. Finally, it is relevant to be aware of the fact the PID controllers use the gains obtained for the practical setup and were not tuned for this simulation setup, which is not exactly the same as the real application.

The same procedure was then applied for a delay $D = 0s$, which yielded the following results:

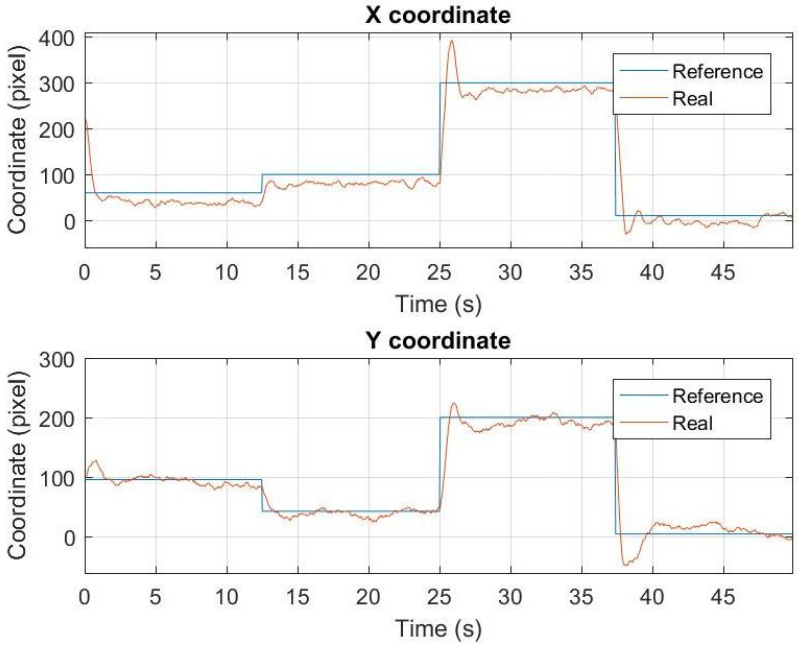


Figure 93. Model results for $D = 0s$

In comparison to the case where $D = 0.5s$, the results are far superior even for non-tuned gains of the PID controllers. Furthermore, by comparing Figures 92 and 93, it becomes evident the delay hugely impacts the performance of the system. Although not visible in Figure 93 due the limited number of iterations, the system eventually converges to the reference values, but it is clear the settling time to an output value equal to 95% of the reference value is reduced and is equivalent to approximately 2 seconds. Instead of analyzing the control for X and Y coordinates separately, a 2D plot is provided to observe the behavior, as shown in Figure 94 below:

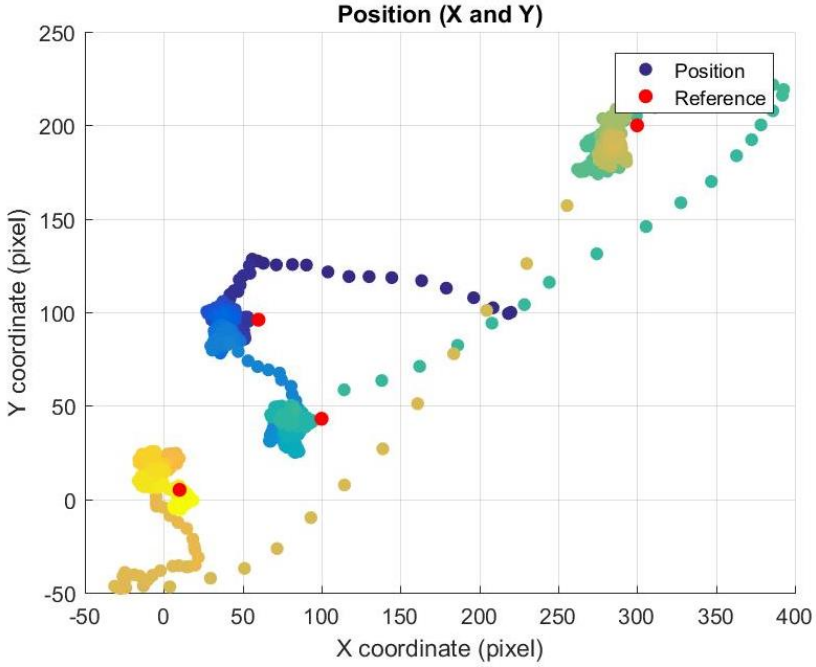


Figure 94. Position results of the model for $D = 0s$

The position ranges from dark blue to bright yellow in Figure 94 above, with the former corresponding to the initial position, meanwhile the references are shown in red. Notice the model reacts rapidly and once it is close to the reference, oscillates around it mainly due the neural network's deviation. Tuning the gains for the PID controllers would theoretically improves even further the response, but such procedure was not performed for the simulation setup.

In summary, not only does the stream delay remarkably decreases the responsiveness of the system, but it also slows it down and restrains stability. Even though the analyzed models are approximations of the real system, the obtained results are comparable to the observed behavior in practice. By simply decreasing the delay, the settling time was reduced even for the same gains of the PID controllers, hence replacing the interface used for processing the camera stream is extremely beneficial. However, due to the current implementation constraints, it is not possible to reduce the stream delay, because the neural network executes on a PC.

4.2 Real-time analysis

Ideally, the tracking system implementation should be analyzed prior to deployment through a dataflow graph, derived from its respective task graph. However, the current implementation is rather complex in terms of communication, runs in a Linux operating system and utilizes both synchronous and asynchronous events. Hence, in order to simplify the current architecture but keep it close to the real implementation, the following points were made:

- Stream delay is considered;
- The neural network executes at 20 Hz;
- Sensor data is provided at 140 Hz;
- No handler tasks (NN Handler or MegaPi Handler) are considered;
- There is no Coordinator task: a "merger" is used instead;
- Worst-case execution times are unknown;
- Behavior of the Kalman filter is approximated (i.e. prediction and update steps are not segregated);
- Arbitration effects are not considered;
- Scheduling policy of the operating system is not contemplated.

Considering the aforementioned points, the Single Rate Dataflow (SRDF) graph in Figure 95 was derived for this system. Actor D corresponds to the delay, NN to the neural network, KFx and KFy to the Kalman filters, S to the sensor data retrieval, PKF to the Pre-Kalman filter, ePID and bPID to the PID controllers and M to the merging procedure. The latter essentially combines the RPMs and forwards it to the robotic system. ρ and δ correspond to the worst-case execution time of each actor (i.e. task) and number of tokens (i.e. size of the message queue) between actors, respectively.

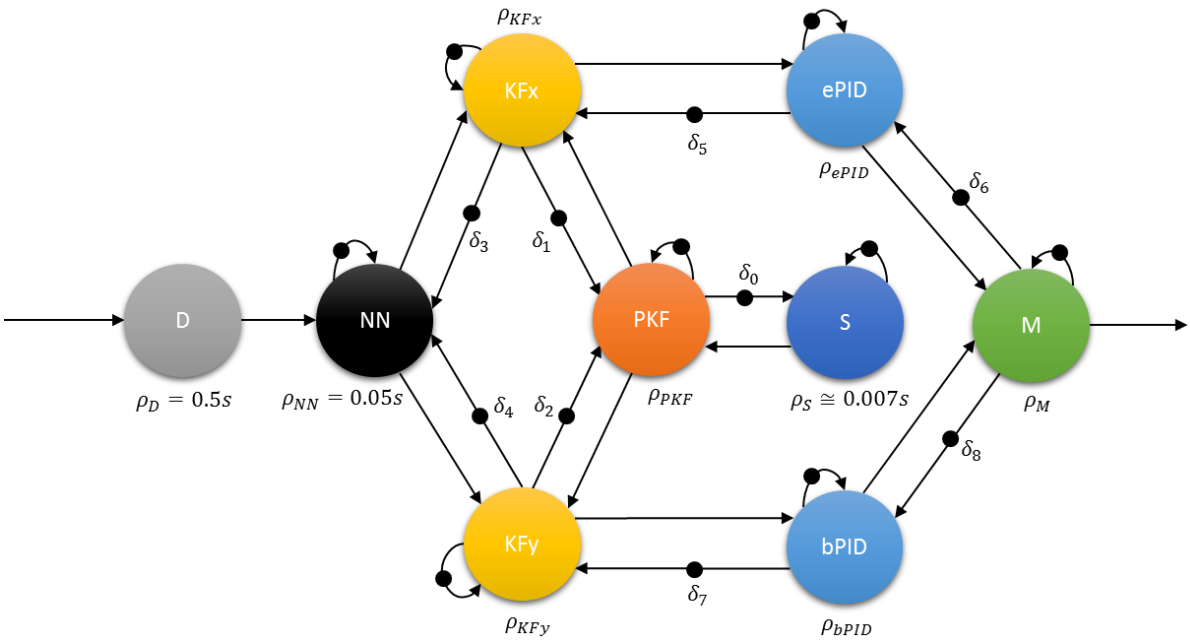


Figure 95. Tracking system SRDF graph model

Note the model does not completely matches the implementation. Communication and behavior of Pre-Kalman filter and Kalman filters are the most notable cases, due the fact the prediction step of the Kalman filters is performed whenever data is received from the Pre-Kalman filter – which is not captured in the current model. A multi-rate dataflow graph (MRDF) is more suitable for describing this behavior, alongside partitioning the Kalman filter into two (2) actors instead of one, with each actor comparable to the prediction and update steps, respectively. Even with such modifications, token consumption is dependent on worst-case execution times of both neural network and Pre-Kalman filter.

Although ρ_{NN}, ρ_D, and ρ_S are considered the maximum worst-case execution times of the neural network, delay, and sensor actors, respectively, are not accurate due the fact they do not consider the actual execution time of the tasks. Instead, they are solely based on their current sampling rate (for the neural network and sensor cases). Moreover, the execution time of each actor cannot be easily nor accurately derived because this application currently runs on a Linux machine, thus a non-determinist multi-thread environment, being influenced by arbitrary delays introduced by other applications on the PC (workload), in addition to the influence of the scheduling policy (arbitration). Both effects, however, can be included in the model, with the former using a two parameter (σ, ρ) characterization and the latter using a latency-rate dataflow model [52, 53]. Moreover, both characterizations can be combined and, in simple terms, a task is represented by two actors instead of one.

Even for such model, however, it is possible to ponder about the major bottlenecks in terms of throughput and latency. The delay currently dictates the latter, due the fact $\rho_D \gg \rho_{others}$: the execution time of other actors are much greater than the delay itself. The throughput, on the other hand, is constrained by the neural network actor (NN) which presents the greatest execution time after the delay, and triggers the core actors. Thus, delay and neural network are the bottlenecks and must be addressed in order to increase both throughput and latency of the system.

Table 9. WCET characterization

Task	Pre-KF	KF	PID	Merger	Total	
					Time (ms)	Frequency (Hz)
Execution Time (ms)	5.7	3.2	0.2	1.0	**10.1**	**99.0**

Finally, the WCET characterization presented in Table 9 above was derived in practice for the core tasks: Pre-KF, KF, PID and Merger (i.e. Coordinator). Both neural network and sensor data retrieval from the robotic setup were assumed periodic. The former is essentially what triggers all other tasks, thus the estimated execution time for a single activation can be estimated by simply adding the execution time of the core tasks, which results in 10.1 milliseconds, as shown in Table 9. Hence, the absolute maximum frequency the current implementation can (theoretically) achieve is analogous to 99.0 Hz, with the current bottleneck being the neural network itself running at 20 Hz.

Taking into account the presented shortcomings of the model, there is no point on pondering about its throughput nor latency. However, it is presented here alongside the rough WCET characterization for the core tasks in order to serve as a starting point for future work, requiring further refinement. More specifically, the Kalman filters and actors modelling overall, alongside fine grain precision on worst-case execution time of the tasks.

5. Conclusions and future work

The main objectives of this research comprised implementing, investigating, and evaluating a practical implementation of a system with tracking and sensor fusion techniques, for an embedded platform. A complete practical working setup was the major outcome of the graduation project described in this thesis, and based on results, the research questions were answered.

The uncomplicated and fast complementary filter was used to fuse encoder and IMU data, which greatly increased the ego-motion sampling frequency from 50 to 140 Hz, in addition to increasing its accuracy. The estimation of the ego-motion impact, in terms of pixel movement, was implemented through approximation functions that are only dependent on the derived speed and gyroscope data, and proved to be fairly accurate without the need for more complex procedures. The standard Kalman filter implemented considers the pixel speeds derived with the ego-motion data, and proved to handle false-negatives, target occlusions and decrease the position deviation in a reliable manner meanwhile dealing with several uncertainties. However, due the fact that a pure software solution was used for the tracking system and a 0.5 seconds delay was introduced by the wireless transmission of the video stream, the complete implementation has a noticeably slow responsive behavior. Ultimately, responsiveness of the system is strictly related to the execution time of the neural network, in addition to the delay present: both must be reduced in order to enhance performance.

During development computational complexity was always considered and reduced whenever possible. Firmware running on the robotic system, for instance, only uses floating-point divisions for the speed control loop (PID controller), and double precision is avoided at all costs. Moreover, the Pre-Kalman filter module only requires sine and cosine functions to properly compute the pixel speeds based on the ego-motion data, and in the worst-case scenario might be bypassed at a cost of decreasing the performance of the Kalman filter. A specific library was developed for the latter, which includes dynamic timing computation (if desired), uses float precision, few variables (thus does not require lots of memory) and does not implement matrix operations (i.e. inversion) in order to further reduce execution time of the implementation. The PID controllers were also implemented in a specific library, and are similar in terms of precision and memory usage in comparison to the Kalman filter. Both instances of the Kalman filters and PID controllers can be instantiated and run in parallel, as there is no data dependency, hence they can be directly mapped to processing elements in a FPGA.

Although the final implementation executes on a PC, this gives room to exploring jitter and non-deterministic behavior, and does not invalidate the research; in practice, most industrial applications still make use of PCs. Moreover, the whole application is implemented such that it can be easily ported to an embedded Linux. The necessary modifications correspond to the usage of a retrieval method of the neural network output, modification of the serial port used for communicating with the robotic system, and tuning the PID controllers once more in case their sampling frequency is changed. Finally, both systems must be correctly connected and interfaced to properly communicate. If a bare-metal implementation is desired, however, modifications are rather complex. Both software and hardware implementations followed guidelines during development and can be easily modified to fulfill future requirements. This is possible because development considered possible functionalities addition in forthcoming versions, likewise understandability such that others can continue with the results of this graduation project.

Despite the final setup performed reasonably good given the constraints imposed by mainly the tracking system implementation, some parts are not ideal and can be improved. Reviewing the project arduously, the following aspects can be enhanced:

- Complementary filter: The filter can be better tuned, however a reliable way of accurately determining the speed of the robot must be used in order to do so;
- Speed estimation: Pitch correction can be included, while considering roll and implementing mechanisms to prevent accumulating speed when the robot is rolling or pitching;
- Communication between tracking and robotic system: Instead of using an USB cable to connect both systems, a direct (serial) connection would be a far superior implementation;
- Real-time implementation: The neural network should be ported to a FPGA, alongside the other software components being totally (or partially) ported to an embedded Linux (or bare-metal). Not only would it hugely decrease the current delay introduced by the stream, but also mitigate the bottleneck of the neural network and remove the need for extra applications, thus increasing the system performance. Finally, multiple-object detection and localization, alongside power optimization are relevant topics, and might be addressed;
- Pre-Kalman filter: The number of points considered to estimate the base speed and angle functions can be further augmented to increase the precision of this module. Moreover, alternatives to estimating the ego-motion impact might be explored;
- Kalman filter: A variation of the standard Kalman filter or even a completely different method might perform better, and could be explored in the future. Moreover, the current design can be augmented with other techniques and improved with further testing;
- Pixel control: Re-tuning and greater sampling frequency might be addressed. Furthermore, steering the robot can be performed in a different manner, or completely modified to a distance control approach. Finally, the control can be optimized for delay and uncertainties;
- Additional modules: The robotic system may include extra modules, such as a compass for determining the current heading, thus positioning is possible. Moreover, a Bluetooth can replace the USB connection with the PC, although its protocol stack might introduce delays. Additionally, the camera can be mounted on top of a servo motor, in order to follow the target by moving the camera, although this complicates the design and invalidate the current ego-motion impact estimation. Finally, an ultrasonic sensor can be placed in front of the robotic system to derive the distance to other objects;
- Communication protocol: The protocol does not implement any type of Cyclic Redundancy Check (CRC) nor ACK/NACK procedures, which may compromise the communication in the long term. Hence, it is advisable to improve such protocol, especially when relevant data is sent or received, alongside increasing the serial baud rate, if supported by the tracking system;
- Neural network and Kalman filter combination: Instead of filtering out the output of the neural network, the Kalman filter might be combined with the neural network itself. However, a more complex network architecture and training set are required in this case;
- Real-Time analysis: RT tools were not used to analyze the tracking system software, hence future work should determine the theoretical bounds on throughput and latency.

It is important to keep in mind that a system can always be improved. Moreover, a wide variety of alternative designs might be derived from the current realization.

6. Bibliography

[1] Kim, Tae-Il, et al. "Vision system for mobile robots for tracking moving targets, based on robot motion and stereo vision information." System Integration (SII), 2011 IEEE/SICE International Symposium on. IEEE, 2011.

[2] Available at: http://store.digilentinc.com/zybo-zynq-7000-arm-fpga-soc-trainer-board/. Accessed in: November 6th, 2017.

[3] Available at: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed in: November 6th, 2017.

[4] Babaians, Edwin, et al. "Skeleton and visual tracking fusion for human following task of service robots." Robotics and Mechatronics (ICROM), 2015 3rd RSI International Conference on. IEEE, 2015.

[5] Wang, Howard, and Sing Kiong Nguang. "Video target tracking based on fusion state estimation." Technology Management and Emerging Technologies (ISTMET), 2014 International Symposium on. IEEE, 2014.

[6] Franken, Dietrich, and Andreas Hupper. "Unified tracking and fusion for airborne collision avoidance using log-polar coordinates." Information Fusion (FUSION), 2012 15th International Conference on. IEEE, 2012.

[7] Crowley, James L., and Yves Demazeau. "Principles and techniques for sensor data fusion." Signal processing 32.1-2 (1993): 5-27.

[8] Castanedo, Federico. "A review of data fusion techniques." The Scientific World Journal 2013 (2013).

[9] Shaikh, Muhammad Muneeb, et al. "Mobile robot vision tracking system using unscented Kalman filter." System Integration (SII), 2011 IEEE/SICE International Symposium on. IEEE, 2011.

[10] Panahandeh, Ghazaleh, Magnus Jansson, and Seth Hutchinson. "IMU-camera data fusion: Horizontal plane observation with explicit outlier rejection." Indoor Positioning and Indoor Navigation (IPIN), 2013 International Conference on. IEEE, 2013.

[11] Antonello, Riccardo, et al. "IMU-aided image stabilization and tracking in a HSM-driven camera positioning unit." Industrial Electronics (ISIE), 2013 IEEE International Symposium on. IEEE, 2013.

[12] Kim, Tae-Il, et al. "Vision system for mobile robots for tracking moving targets, based on robot motion and stereo vision information." System Integration (SII), 2011 IEEE/SICE International Symposium on. IEEE, 2011.

[13] Vaidehi, V., et al. "Neural network aided Kalman filtering for multitarget tracking applications." Computers & Electrical Engineering 27.2 (2001): 217-228.

[14] Yu, Zhi-Jun, et al. "Neural network aided unscented kalman filter for maneuvering target tracking in distributed acoustic sensor networks." Computing: Theory and Applications, 2007. ICCTA'07. International Conference on. IEEE, 2007.

[15] Available at: https://makeblockshop.eu/products/makeblock-ultimate-robot-kit-v2. Accessed in: November 6th, 2017.

[16] Available at: http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf. Accessed in: November 6th, 2017.

[17] Available at: https://github.com/Makeblock-official/Makeblock-Libraries/archive/master.zip. Accessed in: November 6th, 2017.

[18] Available at: https://www.kiwi-electronics.nl/optical-encoder-motor-25-9v-86rpm?lang=en. Accessed in: November 6th, 2017.

[19] Available at: https://store.invensense.com/datasheets/invensense/MPU-6050_DataSheet_V3%204.pdf. Accessed in: November 6th, 2017.
[20] Higgins, Walter T. "A comparison of complementary and Kalman filtering." IEEE Transactions on Aerospace and Electronic Systems 3 (1975): 321-325.
[21] Available at: http://robottini.altervista.org/kalman-filter-vs-complementary-filter
[22] Quoc, Dung Duong, Jinwei Sun, and Lei Luo. "Complementary Filter Performance Enhancement through Filter Gain." International Journal of Signal Processing, Image Processing and Pattern Recognition 8.7 (2015): 97-110.
[23] Min, Hyung Gi, and Eun Tae Jeung. "Complementary filter design for angle estimation using mems accelerometer and gyroscope." Department of Control and Instrumentation, Changwon National University, Changwon, Korea (2015): 641-773.
[24] Yoo, Tae Suk, et al. "Gain-scheduled complementary filter design for a MEMS based attitude and heading reference system." Sensors11.4 (2011): 3816-3830.
[25] Lai, Ying-Chih, Shau-Shiun Jan, and Fei-Bin Hsiao. "Development of a low-cost attitude and heading reference system using a three-axis rotating platform." Sensors 10.4 (2010): 2472-2491.
[26] Seifert, Kurt, and Oscar Camacho. "Implementing positioning algorithms using accelerometers." Freescale Semiconductor (2007): 1-13.
[27] Olson, Edwin. "A primer on odometry and motor control." (2004): 1-15.
[28] Available at: http://www.robotnav.com/position-estimation/. Accessed in: November 6th, 2017.
[29] Available at: http://ttuadvancedrobotics.wikidot.com/odometry. Accessed in: November 6th, 2017.
[30] Available at: http://www.seattlerobotics.org/encoder/200610/Article3/IMU%20Odometry,%20by%20David%20Anderson.htm. Accessed in: November 6th, 2017.
[31] Available at: http://faculty.salina.k-state.edu/tim/robotics_sg/Control/kinematics/odometry.html. Accessed in: November 6th, 2017.
[32] Yamauchi, Genki, Daiki Suzuki, and Keiji Nagatani. "Online slip parameter estimation for tracked vehicle odometry on loose slope." Safety, Security, and Rescue Robotics (SSRR), 2016 IEEE International Symposium on. IEEE, 2016.
[33] Khanniche, M. S., and Yi Feng Guo. "A microcontroller-based real-time speed measurement for motor drive systems." Journal of microcomputer applications 18.1 (1995): 39-53.
[34] Available at: https://gopro.com/help/HERO4-Black. Accessed in: November 6th, 2017.
[35] Available at: https://www.ffmpeg.org/. Accessed in: November 6th, 2017.
[36] Lee, Chang-Ryeol, Ju Hong Yoon, and Kuk-Jin Yoon. "Robust calibration of an ultralow-cost inertial measurement unit and a camera: Handling of severe system uncertainty." Robotics and Automation (ICRA), 2014 IEEE International Conference on. IEEE, 2014.
[37] Fang, Wei, Lianyu Zheng, and Huanjun Deng. "A motion tracking method by combining the IMU and camera in mobile devices." Sensing Technology (ICST), 2016 10th International Conference on. IEEE, 2016.
[38] Faion, Florian, et al. "Camera-and IMU-based pose tracking for augmented reality." Multisensor Fusion and Integration for Intelligent Systems (MFI), 2016 IEEE International Conference on. IEEE, 2016.
[39] Lee, Yongseok, et al. "Camera-GPS-IMU sensor fusion for autonomous flying." Ubiquitous and Future Networks (ICUFN), 2016 Eighth International Conference on. IEEE, 2016.

[40] Li, Mingyang, and Anastasios I. Mourikis. "3-D motion estimation and online temporal calibration for camera-IMU systems." Robotics and Automation (ICRA), 2013 IEEE International Conference on. IEEE, 2013.
[41] Li, Mingyang, and Anastasios I. Mourikis. "Online temporal calibration for camera–IMU systems: Theory and algorithms." The International Journal of Robotics Research 33.7 (2014): 947-964.
[42] Heeger, David J. "Notes on motion estimation." (1996).
[43] Forsyth, David, and Jean Ponce. Computer vision: a modern approach. Upper Saddle River, NJ; London: Prentice Hall, 2011.
[44] Available at: https://nl.mathworks.com/matlabcentral/fileexchange/34765-polyfitn. Accessed in: November 6th, 2017.
[45] Hakim, V. S. Implementation and Analysis of Real-time Object Tracking on the Starburst MPSoC. MS thesis. University of Twente, 2015.
[46] Labbe, R. R. "Kalman and bayesian filters in python." (2015).
[47] Simon, Dan. Optimal state estimation: Kalman, H infinity, and nonlinear approaches. John Wiley & Sons, 2006.
[48] Bar-Shalom, Y., Xiao-Rong L., and Thiagalingam Kirubarajan. Estimation with Applications to Tracking and Navigation. New York: Wiley, 2001.
[49] E. Brookner, Tracking and Kalman Filtering Made Easy, John Wiley & Sons, New York, 1998.
[50] Available at: https://www.tensorflow.org/. Accessed in: November 6th, 2017.
[51] Aström, Karl Johan, and Richard M. Murray. Feedback systems: an introduction for scientists and engineers. Princeton university press, 2010.
[52] Wiggers, Maarten H., Marco JG Bekooij, and Gerard JM Smit. "Modelling run-time arbitration by latency-rate servers in dataflow graphs." Proceedingsof the 10th international workshop on Software & compilers for embedded systems. ACM, 2007.
[53] Wiggers, Maarten H., Marco JG Bekooij, and Gerard JM Smit. "Monotonicity and run-time scheduling." Proceedings of the seventh ACM international conference on Embedded software. ACM, 2009.

7. Appendices

Appendix A: PID Controller

A PID controller is simple and intuitive to implement, reliable, and widely used for Single-Input Single-Output (SISO) systems. Not only does it not require a theoretical model, but can be easily tuned in practice even if the sensors or actuators are changed. However, timing must be respected, thus an accurate timer which has enough granularity with respect to the sampling frequency has to be used. In total, four (4) PID controllers were used in this project, two (2) for the low-level speed control (RPM control) of the encoder motors, and another (2) for the high-level control (speed and direction control). All of them were implemented according to the following formula [A.1]:

$$u_n = u_n + K_p * [e_n - e_{n-1}] + \frac{K_p * T_s}{T_i} * e_n + \frac{K_p * T_d}{T_s} * [e_n - 2 * e_{n-1} + e_{n-2}]$$

With K_p, T_i and T_d being the proportional, integral and derivative gains, T_s the sampling period in seconds, and e_n and u_n the error and absolute control value at iteration n. More specifically, $e_n = Reference_n - Sensor\ Value_n$.

The PID controller takes care of decreasing the error to a minimum, but it requires tuning and a sampling time must be defined. It is a common technique to implement the PID controller as depicted in Figure 96 below, where the *Actuate* phase is done with respect to the **previous iteration output**, instead of the current one. This is done to guarantee the exact same sampling time between iterations of the controller.

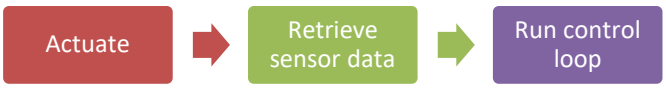


Figure 96. PID controller implementation

Additionally, the control loop is usually guarded with both in- and out-security layers, with both preventing invalid values to be processed or outputted by the controller. In summary, the PID coefficients are found in practice, following this sequence:

1. T_i is set to a high value, T_d and K_p to zero;
2. K_p is increased until the rise time is sufficient;
3. T_d is slowly increased to decrease overshoot (damping);
4. T_i is slowly increased until the results are satisfying and;
5. Steps 2-4 are repeated to improve the response.

[A.1] Available at http://www.kirp.chtf.stuba.sk/moodle/pluginfile.php/66882/mod_resource/content/0/tidsdiskret_pid_reg.pdf. Accessed in: November 5th, 2017.

Appendix B: Pre-Kalman Filter Equations

Within this appendix, a summary of the equations used for the Pre-Kalman filter module is presented. The combined pixel speeds are calculated with the following equations:

$$V_x = C_{trans}(RPM, x, y) * \sin\left(\theta_{trans}(x, y)\right) * signal_x(x) + C_{rot}\left(\omega_{yaw}, x, y\right) * \sin\left(\theta_{rot}(x, y)\right) * signal_{\omega_{yaw}}\left(\omega_{yaw}\right)$$

$$V_y = C_{trans}(RPM, x, y) * \cos\left(\theta_{trans}(x, y)\right) * signal_{RPM}(RPM) + C_{rot}\left(\omega_{yaw}, x, y\right) * \cos\left(\theta_{rot}(x, y)\right) * signal_{\omega_{yaw}}\left(\omega_{yaw}\right) * signal_x(x)$$

With $C_{trans}(RPM, x, y) = \gamma(RPM) * v_{trans}(x, y)$, and $C_{rot}\left(\omega_{yaw}, x, y\right) = \rho\left(\omega_{yaw}\right) * v_{rot}(x, y)$. Moreover, the signal functions are defined as follows:

$$signal_x(x) = \begin{cases} +1, & if\ x > W/2 \\ -1, & otherwise \end{cases}$$

$$signal_{RPM}(RPM) = \begin{cases} +1, & if\ RPM \geq 0 \\ -1, & otherwise \end{cases}$$

$$signal_{\omega_{yaw}}\left(\omega_{yaw}\right) = \begin{cases} +1, & if\ \omega_{yaw} \geq 0 \\ -1, & otherwise \end{cases}$$

B.1 Translation

The translation components are calculated with the following equations:

$$\gamma(RPM) = 0.1282 * |RPM| - 0.6138$$

$$v_{trans}(x, y) \cong -0.000012x^2 + 0.000166xy - 0.0545x - 0.00016y^2 + 0.09347y + 50.65$$

$$\theta_{trans}(x, y) \cong -0.000105x^2 + 0.000124xy - 0.092x + 0.00009y^2 - 0.1068y + 53.62$$

B.2 Rotation

The rotation components are calculated with the following equations:

$$\rho\left(\omega_{yaw}\right) = -0.0005 * \left|\omega_{yaw}\right|^2 + 0.1474 * \left|\omega_{yaw}\right| - 0.8033$$

$$v_{rot}(x, y) \cong 0.00005x^2 + 0.000002xy - 0.0496x - 0.00004y^2 - 0.0793y + 143.81$$

$$\theta_{rot}(x, y) \cong 0.00008x^2 + 0.0001xy + 0.0183x - 0.000012y^2 - 0.0238y + 80.461$$

Appendix C: Embedded Software

Snippet 1. Encoder motor (init, interrupts, usage)
Snippet 2. IMU module (init, usage)
Snippet 3. Pre-Kalman Filter (implementation)
Snippet 4. Kalman Filter's prediction & update steps
Snippet 5. GoPro Stream Handler
Snippet 6. Neural Network with TensorFlow
Snippet 7. *ffmpeg* invocation and parameters
Snippet 8. MAPP compilation options
Snippet 9. POSIX: Message queue creation and usage example
Snippet 10. POSIX: Thread creation example
Snippet 11. Float over serial example

Snippet 1. Encoder motor

This is the most relevant code in *megapi_fw.ino* file of the application running on the robotic system, used to initialize and drive the encoder motors.

```
1.  /* Code omitted */
2.
3.  // Motors + Encoders
4.  MeEncoderOnBoard encR(SLOT1);                          // Forward, speed > 0
5.  MeEncoderOnBoard encL(SLOT2);                          // Forward, speed < 0
6.
7.  /* INTERRUPTS */
8.  // Interrupts for encoders
9.  void ISR_encR(void)
10. {
11.   if(digitalRead(encR.getPortB()) == 0)
12.     encR.pulsePosMinus();
13.   else
14.     encR.pulsePosPlus();
15. }
16. void ISR_encL(void)
17. {
18.   if(digitalRead(encL.getPortB()) == 0)
19.     encL.pulsePosMinus();
20.   else
21.     encL.pulsePosPlus();
22. }
23.
24. void setup()
25. {
26.   /* Code omitted */
27.  // Motors + Encoders
28.   TCCR1A = _BV(WGM10);                      // 8kHz PWM
29.   TCCR1B = _BV(CS11) | _BV(WGM12);
30.   TCCR2A = _BV(WGM21) | _BV(WGM20);
31.   TCCR2B = _BV(CS21);
32.   encR.setPulse(8);                         // Set the pulse number of encoder code disc

33.   encL.setPulse(8);
34.   encR.setRatio(46.67);                     // Set ratio of encoder motor
35.   encL.setRatio(46.67);
36.   encR.setSpeedPid(1.7, 0.1, 0.0001);     // Set internal PID values for speed control
    (RPM)
37.   encL.setSpeedPid(1.7, 0.1, 0.0001);
```

```
38.   encR.setMotionMode(PID_MODE);           // Set to standard SPEED PID controller
39.   encL.setMotionMode(PID_MODE);
40.
41.   attachInterrupt(encR.getIntNum(), ISR_encR, RISING);  // Encoder interrupts
42.   attachInterrupt(encL.getIntNum(), ISR_encL, RISING);
43.   /* ... */
44. }
45.
46. void loop()
47. {
48.   // Drive
49.   drive();
50. }
51.
52. void drive()
53. {
54.   // Set speeds
55.   encR.runSpeed(+100);      // Move robot forwards @ 100 RPM
56.   encL.runSpeed(-100);
57.   // Update encoder values
58.   encR.loop();
59.   encL.loop();
60. }
```

Snippet 2. IMU module

This is the most relevant code in *megapi_fw.ino* file of the application running on the robotic system, used to initialize and retrieve data from the IMU (gyroscope and accelerometer) module.

```
1.  /* Code omitted */
2.
3.  // Gyro + acc
4.  MeGyro gyroacc(0, 0x68);                          // Gyro+Acc
5.  double pData[6];                                 // Pre-processed data
6.
7.  void setup()
8.  {
9.    /* Code omitted */
10.   // Gyro + Acc
11.   gyroacc.begin();                        // Calibrate already
12.
13.   /* ... */
14. }
15.
16. void loop()
17. {
18.   // Retrieve pre-processed data
19.   gyroacc.getProcessedData(&pData[0]);
20.
21.   /* process it ... */
22. }
```

Snippet 3. Pre-Kalman Filter

This is the most relevant code of the Pre-Kalman filter thread in *main.c* file of the main application.

```
1.  // Defines
2.  #define IMAGE_WIDTH         (320)               // Image dimensions
3.  #define IMAGE_HEIGHT        (240)
4.  #define MIN_RPM_PREKF       (5.0f)              // Minimum RPM for Pre-KF
5.  #define MIN_WYAW_PREKF      (3.0f)              // Minimum yaw for Pre-KF
6.  // Macros
7.  #define DEG_TO_RAD(x)       (float)(x*PI/180.0f)  // Degrees to radians conversion
8.
```

```
9.  /* Code omitted */
10. 
11. // Pre-KF thread
12. void *prekfThread(void *arg)
13. {
14.     /* Initialization omitted */
15. 
16.     /* Assuming data is received from Robotic system & Kalman Filters */
17.     // Check if data is valid
18.     if(x >= -20 && x < (IMAGE_WIDTH+20) &&
19.        y >= -20 && y < (IMAGE_HEIGHT+20) &&
20.        !isnan(sensors.speed) && !isinf(sensors.speed) &&
21.        !isnan(sensors.heading) && !isinf(sensors.heading) &&
22.        !isnan(sensors.pitch) && !isinf(sensors.pitch) &&
23.        !isnan(sensors.roll) && !isinf(sensors.roll) )
24.     {
25.         /* RPM & WYAW correction */
26.         // RPM
27.         if(sensors.speed < 0)
28.             auxRPM = -sensors.speed;
29.         else
30.             auxRPM = +sensors.speed;
31.         // wyaw
32.         if(sensors.heading < 0)
33.             auxWy = -sensors.heading;
34.         else
35.             auxWy = +sensors.heading;
36.         /* SYMMETRY & SIGNAL(X) */
37.         if(x > (IMAGE_WIDTH/2))
38.         {
39.             auxX = IMAGE_WIDTH - x;
40.             sig_x = +1;
41.         }
42.         else
43.         {
44.             sig_x = -1;
45.             auxX = x;
46.         }
47.         /* SIGNALS */
48.         // signal(RPM)
49.         if(sensors.speed >= 0)
50.             sig_rpm = +1;
51.         else
52.             sig_rpm = -1;
53.         // signal(yaw)
54.         if(sensors.heading >= 0)
55.             sig_wy = +1;
56.         else
57.             sig_wy = -1;
58.         /* SCALE FACTORS */
59.         // Translation
60.         if(auxRPM < MIN_RPM_PREKF)
61.             gama = 0;
62.         else
63.             gama = 0.1282f*auxRPM - 0.6138f;
64.         // Rotation
65.         if(auxWy < MIN_WYAW_PREKF)
66.             rho = 0;
67.         else
68.             rho = -0.0005f*auxWy*auxWy + 0.1474f*auxWy - 0.8033;
69.         /* SPEEDS */
70.         St = -
    0.000012f*auxX*auxX + 0.0001655f*auxX*y - 0.05445f*auxX       // Translation
71.              -0.0001632f*y*y + 0.0934647f*y + 50.6455;
72.         Sr = 0.000052f*auxX*auxX + 0.000002f*auxX*y - 0.04962f*auxX         // Rotation
```

```
            -0.000037f*y*y - 0.07933f*y + 143.8089f;
        Ct = gama*St;
        Cr = rho*Sr;
        /* ANGLES */
        Tt = -
0.000105f*auxX*auxX + 0.000124f*auxX*y - 0.0922076f*auxX      // Translation
             +0.000086f*y*y - 0.10679f*y + 53.616f;
        Tr = 0.000079f*auxX*auxX + 0.000101f*auxX*y + 0.0183f*auxX          //Rotation

             -0.000012f*y*y - 0.02381f*y + 80.4608f;
        if(Tr > 90)
            Tr = 90;
        // Convert to radians
        Tt = DEG_TO_RAD(Tt);
        Tr = DEG_TO_RAD(Tr);
        /* FINAL SPEEDS */
        // Compute Vx and Vy - finally :) - Divided by two due the image resizing
        Vx = (Ct*sin(Tt)*sig_x   + Cr*sin(Tr)*sig_wy)/2;
        Vy = (Ct*cos(Tt)*sig_rpm + Cr*cos(Tr)*sig_x*sig_wy)/2;
    }
    // Invalid coordinates, cannot estimate Vx and Vy...
    else
        Vx = Vy = 0;

    /* Pixel speeds are then forwarded to the KFs */
}
```

Snippet 4. Kalman Filter's prediction & update steps

This is the most relevant code in *kf.c* file, used for the predictio and update steps of the Kalman filter implementation.

```
// Local functions declaration
void kf_computeQ(kf_t *kf);
float kf_getDt(struct timespec *sT);

/* Code omitted */

//---------------------------------------------------------------------------
// Perform the prediction step
// kf        : Config structure
// vr        : Self-motion impact, in px/s
// useDt0    : Use period defined in kf_init(...)?
// return    : TRUE if successful, FALSE otherwise
bool kf_predict(kf_t *kf, float vr, bool useDt0)
{
    /* Prediction equations:
     * x^(k|k-1) = F*x^(k-1|k-1) + B*u
     * P(k|k-1) = alpha^2*F*P(k-1|k-1)*F' + Q
     */

    // Sanity check
    if(kf != NULL && kf->init && !kf->idle)
    {
        // Compute dT
        if(!useDt0 && !kf->first)
        {
            kf->dT = kf_getDt(&kf->sT);
            // Recompute Q based on dT
            kf_computeQ(kf);
        }
        // Else, everything already set...
        kf->first = false;       // Make sure it is not the first one anymore

        // Actual prediction
```

```
34.         kf->pos     = kf->pos + (kf->spd + vr)*kf->dT;
35.         kf->spd     = kf->spd;
36.         kf->P.P11   = kf->alpha2*(kf->P.P11 + kf->dT*(kf->P.P12 + kf->P.P21 + kf-
    >dT*kf->P.P22)) + kf->Q.Q11;
37.         kf->P.P12   = kf->alpha2*(kf->P.P12 + kf->dT*kf->P.P22) + kf->Q.Q12;
38.         kf->P.P21   = kf->alpha2*(kf->P.P21 + kf->dT*kf->P.P22) + kf->Q.Q21;
39.         kf->P.P22   = kf->alpha2*kf->P.P22 + kf->Q.Q22;
40.
41.         // DONE!
42.         return true;
43.      }
44.      else
45.         return false;
46. }
47.
48. //---------------------------------------------------------------------------
49. // Perform the update step
50. // kf        : Config structure
51. // sensorVal : Sensor value
52. // return    : TRUE if successful, FALSE otherwise
53. bool kf_update(kf_t *kf, float sensorVal)
54. {
55.     /* Update equations:
56.      * y(k) = z(k) - H*x^(k|k-1)
57.      * S(k) = H*P(k|k-1)*H' + R
58.      * K(k) = P(k|k-1)*H'*inv(S(k))
59.      * x^(k|k) = x^(k|k-1)+K(k)*y(k)
60.      * P(k|k) = (I-K(k)*H)*P(k|k-1)
61.      */
62.
63.     // Auxiliar variables
64.     float yk = 0;
65.     float p11 = 0;
66.     float p12 = 0;
67.
68.     // Sanity check
69.     if(kf != NULL && kf->init && !kf->idle)
70.     {
71.         yk = sensorVal - kf->pos;
72.         kf->S = kf->P.P11 + kf->senVar;
73.         kf->K.K1 = kf->P.P11/kf->S;
74.         kf->K.K2 = kf->P.P21/kf->S;
75.         kf->pos = kf->pos + kf->K.K1*yk;
76.         kf->spd = kf->spd + kf->K.K2*yk;
77.         // Copy to aux vars
78.         p11 = kf->P.P11; p12 = kf->P.P12;
79.         kf->P.P11 = p11*(1 - kf->K.K1);
80.         kf->P.P12 = p12*(1 - kf->K.K1);
81.         kf->P.P21 = kf->P.P21 - kf->K.K2*p11;
82.         kf->P.P22 = kf->P.P22 - kf->K.K2*p12;
83.
84.         // DONE!
85.         return true;
86.     }
87.     else
88.         return false;
89. }
90.
91. /* INTERNAL FUNCTIONS */
92. //---------------------------------------------------------------------------
93. // Compute Q (process noise covariance matrix) based on dT
94. void kf_computeQ(kf_t *kf)
95. {
96.     // Does not perform sanity because it is an internal function...
97.     float dT = kf->dT;
98.     float dT2 = dT*dT;
```

```
    float dT3 = dT2*dT;
    float dT4 = dT3*dT;

    // Compute elements
    kf->Q.Q11 = kf->proVar*dT4/4;
    kf->Q.Q12 = kf->proVar*dT3/2;
    kf->Q.Q21 = kf->Q.Q12;
    kf->Q.Q22 = kf->proVar*dT2;
}

//---------------------------------------------------------------------------
// Compute dT, in seconds
float kf_getDt(struct timespec *sT)
{
     // Variables
     float dT = 0;
     struct timespec eT;

     // Get time
     clock_gettime(CLOCK_REALTIME, &eT);
     // Compute dT, in seconds
     dT = (eT.tv_sec - sT->tv_sec) + (eT.tv_nsec - sT->tv_nsec)/1000000000.0f;
     // Overwrite initial time
     clock_gettime(CLOCK_REALTIME, sT);

     // Return
     return dT;
}
```

Snippet 5. GoPro Stream Handler

This is the most relevant code in *gopro.py* file (GPNN application), used to initialize the camera stream, launch the *ffmpeg* application and keep alive the stream.

```
# Libs
from time import sleep
import socket                                       # Keep alive is sent via UDP
import urllib.request                               # HTTP get for triggering stream
import subprocess                                   # Launch ffmpeg
import threading                                    # Keep Alive thread for GoPro

# Variables
# Threading
pipeSemaphore = threading.Semaphore(0)              # Semaphore for syncrhonization
threadFlag = True                                   # Flag for shutting down threads

# Code omitted #

# GoPro-specific command message
def get_command_msg(id):
    return "_GPHD_:%u:%u:%d:%1lf\n" % (0, 0, 2, 0)

# GoPro streaming thread
def gopro_stream():
    # GoPro-specific variables
    UDP_IP = "10.5.5.9"
    UDP_PORT = 8554
    KEEP_ALIVE_PERIOD = 2500                            # Period in seconds
    KEEP_ALIVE_CMD = 2                                  # Do not change this
    # Get keep alive cmd for GoPro
    MESSAGE = bytes(get_command_msg(KEEP_ALIVE_CMD), "utf-8")
    # Triggers streaming
    urllib.request.urlopen("http://10.5.5.9/gp/gpControl/execute?p1=gpStream&a1=proto_v
2&c1=restart").read()
    # Launch ffmpeg (stream preview or save to file only)
```

```
    subprocess.Popen("ffmpeg -loglevel panic -f:v mpegts -an -probesize 8192 -i rtp://10.5.5.9:8554 -vf scale=320:240 -r 20 -updatefirst 1 -y img.png", shell=True)
    print("GoPro> Streaming started...")
    # Release semaphore
    pipeSemaphore.release()     # NN
    pipeSemaphore.release()     # KF
    # Keep alive loop
    while threadFlag:
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
        sleep(KEEP_ALIVE_PERIOD/1000)
```

Snippet 6. Neural Network with TensorFlow

This is the most relevant code in *gopro.py* file (GPNN application), used to initialize the neural networks used, launch the *ffmpeg* application and keep alive the stream.

```
# Libs
from __future__ import print_function
from keras.models import load_model, Model          # Neural Network
import keras
import tensorflow as tf
import numpy as np                                  # For arrays
import cv2                                          # OpenCV
import os                                           # OS utilities
from os import remove
from time import sleep
import socket                                       # Keep alive is sent via UDP
import urllib.request                               # HTTP get for triggering stream
import subprocess                                   # Launch ffmpeg
import threading                                    # Keep Alive thread for GoPro
# 3rd party POSIX IPC
import posix_ipc

# Variables
# Threading
pipeSemaphore = threading.Semaphore(0)              # Semaphore for syncrhonization
threadFlag = True                                   # Flag for shutting down threads
# Message queue
MQOUT = "/gopro2mainappMQ"                          # Create in main app, MAXMSG=1, MSG SIZE=sizeof(int32_t), BLOCKING
mqout = posix_ipc.MessageQueue(MQOUT)
# Others
normImgs = np.zeros((1,240,320,3),np.uint8)         # Create image arrays
classImgs = np.zeros((1, 224, 224, 3), np.uint8)
# Load models
model = load_model('mobilenet_binary_1.h5', custom_objects={'relu6': keras.applications.mobilenet.relu6, 'DepthwiseConv2D': keras.applications.mobilenet.DepthwiseConv2D})
small_model = load_model('videoModel_improved.h5')              # Small model
# Copy only the convolutional layers to a new model for localization
conv_model = Model(inputs=small_model.input, outputs=small_model.layers[13].output)


# Code omitted #

# Neural network itself
def neural_network():
    # Run prediction on both models to prevent freezing (with fake images)
    prediction = model.predict(classImgs, 1, 0)
    auxLayer = conv_model.predict(normImgs, 1, 0)

    # Wait for Stream to start...
    pipeSemaphore.acquire()

    print("NN> Init done!")
```

```
    # Loop
    while threadFlag:
        start = time.time()
        # Tries reading image
        while True:
            normImg = cv2.imread('img.png', 1)
            if (normImg is not None):
                break
            else:
                time.sleep(0.0001)

        # Delete last image
        remove('img.png')
        # Resize image
        classImg = cv2.resize(normImg, (224, 224))
        # Re-arrange colors
        classImgs[0] = classImg[...,::-1]
        normImgs[0] = normImg[...,::-1]

        # Run prediction for big model
        prediction = model.predict(classImgs, 1, 0)

        xVal = -1
        yVal = -1
        # Check if a car was detected
        if (np.argmax(prediction)):
            # Car detected, run prediction for small model
            auxLayer = conv_model.predict(normImgs, 1, 0)
            auxLayer2 = auxLayer[0,:,:,0]
            coordinates = np.unravel_index(auxLayer2.argmax(), auxLayer2.shape)
            xVal = (coordinates[1]*8 + 10).tolist() # x
            yVal = (coordinates[0]*8 + 30).tolist() # y

            # Modify image
            cv2.circle(normImg, (xVal,yVal), 4, (0,255,0), -1)

        # Get KF output and add a purple circle
        try:
            xAux = xList.pop()
            yAux = yList.pop()
            cv2.circle(normImg, (xAux,yAux), 5, (0,255,255), -1)
        except IndexError:
            pass    # Do nth...

        # Show image
        cv2.imshow('Stream', normImg)
        cv2.waitKey(1)

        # Send data to Main App
        x = (xVal).to_bytes(2, byteorder='little', signed=True)
        y = (yVal).to_bytes(2, byteorder='little', signed=True)
        mqout.send(x + y)

            time.sleep(0.0001)      # Yield processor (100us)
```

Snippet 7. *ffmpeg* invocation and parameters

The *ffmpeg* (Stream Capture) application is launched by the GPNN, using the following bash command:

```
ffmpeg -loglevel panic -f:v mpegts -an -probesize 8192 -i rtp://10.5.5.9:8554 -
vf scale=320:240 -r 20 -updatefirst 1 -y img.png
```

The parameters are described below:

- -loglevel panic: (optional) display only panic messages;

- -f:v mpegts: camera's stream format;
- -an: (optional) disable audio from stream;
- -probesize 8192: used to reduce stream's latency;
- -i rtp://10.5.5.9:8554: indicate the specific IP/RTP port used by the camera;
- -vf scale=320:240: video filter for resizing the incoming frame to 320x240 resolution;
- -r 20: (soft-)limit the frame rate to 20 fps;
- -updatefirst 1: overwrites output image, if it already exists;
- -y: force [y]es option for any prompts;
- img.png: name of the output image file.

Snippet 8. MAPP compilation options

The main application is normally compiled (line 1) and run (line 2) with the following bash command:

```
1.  bash> gcc main.c kf.c pid.c megapi.c typedefs.c -pthread -lrt -lm
2.  bash> sudo ./a.out
```

However, the following additional flags can be used for compilation, either singularly or combined, if needed:

- -DDEBUG: Used for debugging, print out debugging messages during execution;
- -DEXECTIME: Used for estimating the WCET of all threads;
- -DBYPASS: Used during testing, bypass the Pre-Kalman filter computations ($V_x = V_y = 0$);
- -DFAKENN: Used during testing, launches a fake GPNN application;
- -DWEBCAM: Used during testing, launches a similar GPNN application which process a camera stream from a USB webcam.

Notice the only restriction is -DFAKENN and -DWEBCAM cannot be used together. For standard operation, use the compilation command without extra flags.

Snippet 9. POSIX: Message queue creation example

This is an example on how to create message queues when using POSIX in a Linux environment for the high-level control system.

```
1.  #include <mqueue.h>               // Message queues
2.
3.  // Message queue name
4.  char *mqName        = "/mq";
5.
6.  // Message queue object
7.  mqd_t mq;
8.
9.  bool create_mqueue(mqd_t *mqueue, char *name, long maxmsg, long msgsize, int flags)
10. {
11.     // Local attributes
12.     struct mq_attr attr;
13.
14.     // Sanity check
15.     if(mqueue != NULL && name != NULL && maxmsg != 0 && msgsize != 0)
16.     {
17.         // Set attributes
18.         attr.mq_maxmsg = maxmsg;        // Max capacity
19.         attr.mq_msgsize = msgsize;      // Msg size, in bytes
```

```
        attr.mq_flags = 0;              // Blocking
        umask(0);                       // Change permission mask

        // Create mqueue
        if( ((*mqueue) = mq_open(name, flags | O_CREAT, 0666, &attr)) < 0)
            return false;

        // Done!
        return true;
    }
    else
        return false;
}
void main()
{
    /* CREATE MQUEUES */
    // Create a non-blocking uint16_t message queue, size 2
    // For a blocking message queue, omit flag O_NONBLOCK
    if( !create_mqueue(&mq, mqName, 2, sizeof(uint16_t), O_RDWR | O_NONBLOCK) )
    {
        printf("MAIN> Could not create %s.\r\n", mqName);
        exit(-1);
    }
}
```

Snippet 10. POSIX: Thread creation example

This is an example on how to create threads when using POSIX in a Linux environment for the high-level control system.

```
#include <pthread.h>          // Posix threads
#include <sched.h>            // Tweaking priorities

/* THREADS */
// Thread function
void *thread(void *arg)
{ /* Do sth... */ }

bool setPriority(pthread_attr_t *attr, struct sched_param *param, int priority)
{
    // Sanity check
    if( attr != NULL && param != NULL &&
       (priority >= -20 && priority <= 19) )
    {
        // Init attributes + scheduling variables
        pthread_attr_init(attr);
        pthread_attr_getschedparam(attr, param);
        // Set priority
        param->sched_priority = priority;
        // Set new scheduling parameters
        pthread_attr_setschedparam(attr, param);

        // Done... user can create thread now
        return true;
    }
    else
        return false;
}
void main()
{
    // Threads
    pthread_t th;
    // Attributes - for priority tweak
    pthread_attr_t attr;
    struct sched_param sched;
```

```
36.
37.     /* CREATE THREADS */
38.     // Create thread with priority 0 - default
39.     setPriority(&attr, &sched, 0);
40.     if( pthread_create(&th, &attr, thread, NULL) != 0)
41.     {
42.         printf("Could not create thread.\r\n");
43.         exit(-1);
44.     }
45.
46.     // Shut down "main" thread
47.     pthread_exit(NULL);
48.     return 0;
49. }
```

Snippet 11. Float over serial example

This is an example on how to send and receive 32-bit floating point values over serial which handles bytes.

```
1.  // Union structure for float values
2.  typedef union u_tag
3.  {
4.      uint8_t     b[4];
5.      float32_t   fval;
6.  } u_tag;
7.
8.  void sendF(float32_t val)
9.  {
10.     // Create union
11.     u_tag tag;
12.     // Copy value to union
13.     tag.fval = val;
14.     // Send each byte, LSB
15.     send(tag.b[3]);
16.     send(tag.b[2]);
17.     send(tag.b[1]);
18.     send(tag.b[0]);
19. }
20.
21. void recvF(float32_t *val)
22. {
23.     // Create union
24.     u_tag tag;
25.     // Read and copy bytes, LSB
26.     recv(tag.b[3]);
27.     recv(tag.b[2]);
28.     recv(tag.b[1]);
29.     recv(tag.b[0]);
30.     // Copy to output variable
31.     (*val) = tag.fval;
32. }
33.
34. void main()
35. {
36.     // Variables
37.     float toSend = 4.21f;
38.     float toRecv = 0.0f;
39.
40.     // Send float value
41.     sendF(toSend);
42.     // Receive float value
43.     recvF(&toRecv);
44. }
```