2019 Summer Internship Dalhousie University, Halifax, Nova Scotia, Canada

---

Robust Deep Learning Tracking Robot

---

written by ARNAUD KLIPFEL, Robotics specialization at ENSTA-BRETAGNE
supervised by Dr. THOMAS TRAPPENBERG, DALHOUSIE UNIVERSITY

Contents

Ackowledgement

This internship was a very enriching experience in my life. Undoubtedly, the concepts and tools I have come to learn will guide me in my future research career. I want to deeply thank Dr. Trappenberg for his astute and lasting support in my work.

Abstract

The concept of tracking robot can bring trailblazing and game-changing applications either in the industry, defense or at home. More practically, robots could be seen as the future coworkers of humans. In that context, robots will have at some point to be able to not only follow instructions, but assist for instance workers in a factory, and so be able to follow their track. One can imagine a robot that could carry tools, follow any worker, and hand out the right when asked.

Few trackers following general targets, not just human ones, have been realized using deep learning techniques, and even fewer using the middleware *ROS*. As a matter of fact, this project aimed to realize a versatile *ROS* architecture for a tracking robot using a stereo camera and deep learning algorithms to spot a general given target.

The project was divided into several main stakes. First of all, the hardware of the tracker had to be wisely selected to meet the requirements of the tracking process. Secondly, the tracking algorithm had to be chosen and assessed. The *ROS* architecture had then to be imagined and built. Finally, the thorniest part was to integrate each part on the robot.

At the end, a flexible and general *ROS* architecture was implemented and tested in diverse conditions. The realized architecture could then inspire future work, and be the basis of some improvements thanks to the flexibility of *ROS*.

Introduction

THIS document is the report of the project on which I worked during my summer internship in 2019 at Dalhousie University. Additional codes and documentations could be found on my personal *github*. The main goal was to build a tracking robot with the middleware ROS and with deep learning techniques.

The part 1 on the facing page contextualize the project, presents the goals and the strategy. The part 2 on page 9 outlines several projects or existing techniques that were relevant for the project in any way. The part 3 on page 11 explains how the hardware of the tracker was selected. The part 4 on page 17 justifies the choices made regarding the tracking algorithm and the *ROS* architecture. The part 5 on page 21 analyzes the results of the integration of each single part on the robot, that is to say the hardware and the software. Finally, the part 6 on page 24 gives the procedure and some pieces of advice in order to replicate the latest blueprint of the tracker.

Keywords

stereo-vision, tracking, *ROS*, mechatronics, mobile robotics, RC car, ground robot, deep learning.

Part 1

Project Contextualization

This part gives a first and general glimpse into what was at stake in the realization of the tracking robot. The relevance of the project is justified, the requirements and goals of the project are defined, and the global strategy of development is presented.

1.1 Project Specification

1.1.1 Motivations

The main goal of this project was to build a *Deep Learning Tracking Robot*. Many tracking algorithms have been implemented in the previous years [GOTB19], few have been tested on robots, and few have been interfaced with the adaptive, versatile, and now standard robotic Middleware *ROS*, which stands for *Robot Operating System*. In addition, even fewer have ventured combining *ROS* and *Deep Learning* techniques. [VJZ14; Bar; Wan+]

Why *ROS* precisely? As defined by Anis Kouba in [Kou16], *ROS* is an « open-source *middleware* » that provides a robust and reliable framework to build robots. *ROS* is a voguish tool to create robots with advanced functionalities so much so that it has become the standard to develop robots. *ROS2* is already in part operational. In this project though, it has been decided that the first version of *ROS* will be used since *ROS2* is still in heavy development and *ROS* is better documented than its counterpart. Besides, some sensors used in this project, such as the *ZED camera*, see 3.1.1 on page 11, have packages that are still in beta version, hence less reliable than the ones of *ROS*.

Why is *Deep Learning* suitable for robotic applications? In general, machine learning, which comprises *deep learning*, is really adapted for critical algorithms, such as embedded codes on robots, since the performances are much higher than with traditional implementations. By traditional implementations, one could for instance consider iterative algorithms where for a particular task all processes are hand-implemented. The conventional solution that comes to mind regarding tracking is an *RGB* tracking, that is to say an algorithm that tracks the color in a frame or image by applying color filters. *Machine learning* enables much finer and more robust implementations since the *model*, which can be regarded as the applied processes, can evolve by itself through diverse methods such as training.

1.1.2 Requirements

In order to realize the tracking robot, several requirements had to be met in this project. First of all, the robot had to reuse an old *Race Car* platform comprising the chassis, the wheels . . . all the mechanics. The platform is the *H1 model* of the *monster Car* [Car]. The brain of the robot, that is to say the embedded computer had to be the *Jetson Nano* designed by *Nvidia* [MSV19]. By and large, to build the tracker the material of the laboratory had to be used as much as possible, and the equipment bought had to remain affordable.

1.2 Strategy

1.2.1 Different periods

Developing a robot is a critical task, and has to be conducted thoroughly. In this regard, *ROS* builds a framework for developing robots in a more organized way. For each task to implement, the work-flow has always been the following: research, simulation, isolated tests, integration.

Before even implementing something and integrating it, one should research all the possibilities that are offered to them. Then one of the easier or maybe most sustainable options could be chosen. Yet, even the tests, simulation, and integration has to be thought and foreseen beforehand.

Simulating robots, without depending on the hardware is a needed step. A specific hardware has always some specificities that could hide some issues and unpredictable behaviors. As a matter of fact, simulating algorithms that will be integrated in the robot afterward in a controlled environment prevents any hardware based issue to shadow our understanding of the basic implementation of the *software*.

Once the simulation is in place, it is then recommended to test the simulation as much as possible in order to unveil some detrimental singularities. In the case of the hardware, each component has to be tested independently before integration. Especially, some driver issues are sometimes really hard to pinpoint.

The final step is to integrate the work in the robot, and to test it to see if the robot behaves as expected or if there is any regression.

A robot is not just a piece of software, it is a system where the interaction of the software and the hardware is by definition the future behavior of the robot.

1.2.2 Different aspects

The development of a robot encapsulates different areas or types of work. In the construction of the tracker, principally two aspects have been tackled, the hardware, and the software.

In the first place, the hardware has to be well-chosen. The motor had to replaced, the *wifi* access had to be added, batteries had to be chosen to fit the requirements of the tracking robot.

In the second place, the sofware has to be implemented. Precisely, the tracking algorithm had to be chosen, and the *ROS* architecture had to be designed.

For the entire project, the philosophy was to try to have a working first prototype as soon as possible. In a nutshell, having a hardware and software which are compatible, and then refining the existing platform. Indeed, *ROS* provides here another crucial boon, for it enables us to have an evolutive architecture where each part can be replaced easily without undermining the overhaul process. For instance, once an architecture is working, the tracking technique could be easily replaced.

Throughout this report, which presents the results of the project, three questions will be answered:

- How was the hardware selected? You can find the answers to this question in the part 3 on page 11.
- How was the tracking algorithm implemented? Which belongs more to the software. You can find the answers to this question in the section 4.1 on page 17.
- How was the *ROS* architecture designed? Which belongs more to the software. You can find the answers to this question in the section 4.2 on page 18.

Part 2

State of the art

In order to build the tracker, the first step was to choose the hardware of the robot. Secondly, the *ROS* architecture had to be decided and thirdly the tracking algorithm had to be implemented. This part presents a brief overview of different existing projects and solutions that could have been selected throughout the project. All the projects presented and listed here will then be compared and the choices made will be underpinned in the parts 3 on page 11 and 4 on page 17.

2.1 Building an electric car

Several projects have aimed to realize an electric car robot, sometimes autonomous. Having in mind these projects can give some piece of advice on how it is common to design such systems.

Some projects have used the on-board computer *Jetson Nano*. One can name the following robots : *Jetbot* [Nvib], *Kaya* [Nvie]. These robots are both designed by *Nvidia*, the designers of the *Jetson* board series, comprising the Jetson Nano, Xavier, TX1, and TX2. It is notably interesting to take as example their hardware selection since they use the same embedded computer the tracker use.

Other *RC-cars*, which stands for *Remote Control Car*, use other types of embedded computers. However, the hardware architecture with regard to *RC-cars* is almost always the same. With either the *DeepRacer* of *Amazon* [Ama], or the *Pi-Car raspberry* powered car [Sun], or *Ghost* car [Dan], or the *Roscar* [nai], or the *F1tenth* car [F1T], or the *Donkey* car [Diy], the architecture follows some fundamental principles.

The selection of the hardware is discussed further in the part 3 on page 11.

2.2 Tracking

Multiple ways of tracking a target exist [Mal17], in this project the most common one was used : a *visual tracking*. Visual tracking is basically based on the processing of images or frames taken by a camera.

More precisely, the mission of the tracker implemented in this project is to follow a specific object in a frame given by a camera, to be able to learn the object in that process, and even to be able to recover the target after any occlusion. Looking at the deep learning models and algorithms which have been developed in the recent years, some could be more labeled as detection algorithms, do not really learn the specificities of the target, and even track multiple targets. For instance, the network *Yolo* [Bje16] is rather designed for multi-tracking and not to follow a specific target.

Considering deep learning models that are able to track a specific target, the database *GOT-10k* has produced a ranking of the current most effective techniques [GOTB19].

In order to automate the robot, a detection algorithm could precede the tracking.

2.3 Using ROS

Developing with *ROS* gives an incredible advantage to any roboticist. In fact, some *ROS packages* which are totally open-source already provide cutting-edge algorithms such as sensor fusion by Kalman filtering, Simultaneous Localization And Mapping or shortly *SLAM*. Unfortunately, applied to specific object tracking few *ROS* packages were developed, and the existing ones do not use deep learning techniques at all.

Nevertheless, some *ROS interfaces* were implemented for *ROS*, which can allow one to use a tracking algorithm compatible with the *ROS interface*, also often named *bridge* [AS18]. The main problem is mainly that the implemented programs are not up-to-date which renders the integration in *ROS* sometimes almost impossible due to software dependencies.

Apart from that, several projects, almost all shared on *github*, implement tracking solutions. However, they either do not use deep learning, or are not supported by *ROS* anymore. The book *ROS Robotics Projects* presents a well-documented tracking project, though no deep learning techniques are used [Jos17]. Yet, the *ROS* architecture inspired some of the realization of the own architecture of this project.

A published paper tackled a robot for person following [CST]. Yet, the tracker in this project is not just intended to track a person, but more generally any object.

Part 3

Hardware

FIRST of all, before even thinking about the tracking process, and the behavior of the robot, the existing platform had to be fixed, and if needed adapted to the current needs of the project. This part starts by presenting the latest architecture of the tracking robot, an then breaks this architecture apart by giving an overview of each challenge which had to be tackled.

3.1 Hardware architecture

3.1.1 Overview

It is crucial to wisely chose the hardware of the robot in the first place, since the software could be seen then as an exploitation of the hardware of the robot. The initial hardware of the robot did comprise a servomotor, a *brushed DC motor*, see section 3.2 on page 14, and a motor controller, or *ESC*, which stands for *Electronic Speed Controller*.

This initial architecture had to be adapted for several reasons. The latest hardware architecture of the tracking robot is presented on figure 3.1 on the following page, which shows each component and how each interacts with each other.

Let's explain a bit what the crucial parts that constitute a tracking robot are. The target tracking process used in this project is based on image and depth inputs, but to be more precise, on a *stereo camera*. A *stereo camera* provides basically two types of information : depth and color. For this purpose the *ZED camera* of *Stereolabs* was used [ste]. The processing of the frames given by the camera is done on the embedded computer *Jetson Nano*. Regarding the actuators, a servomotor controls the bearing of the car, and a motor controls the speed of the car through an *ESC*. In order to communicate with the motor controller, or *ESC* a *PWM shield* was needed. Basically, a *PWM* signal, which stands for *Pulse Width Modulation* in this case, is a certain type of signal which are characterized by their duty cycle. This fundamental parameter is equivalent to the command sent to the *ESC*. At this end, the *PWM shield* PCA9685 of Adafruit was used [Ada]. Ultimately, since in the field of mobile robotics, robots tend to be mobile by definition, the tracking robot had to be powered on its own. To meet that constraint two batteries were added. From all that has been mentioned in this paragraph, only the initial servomotor was saved.

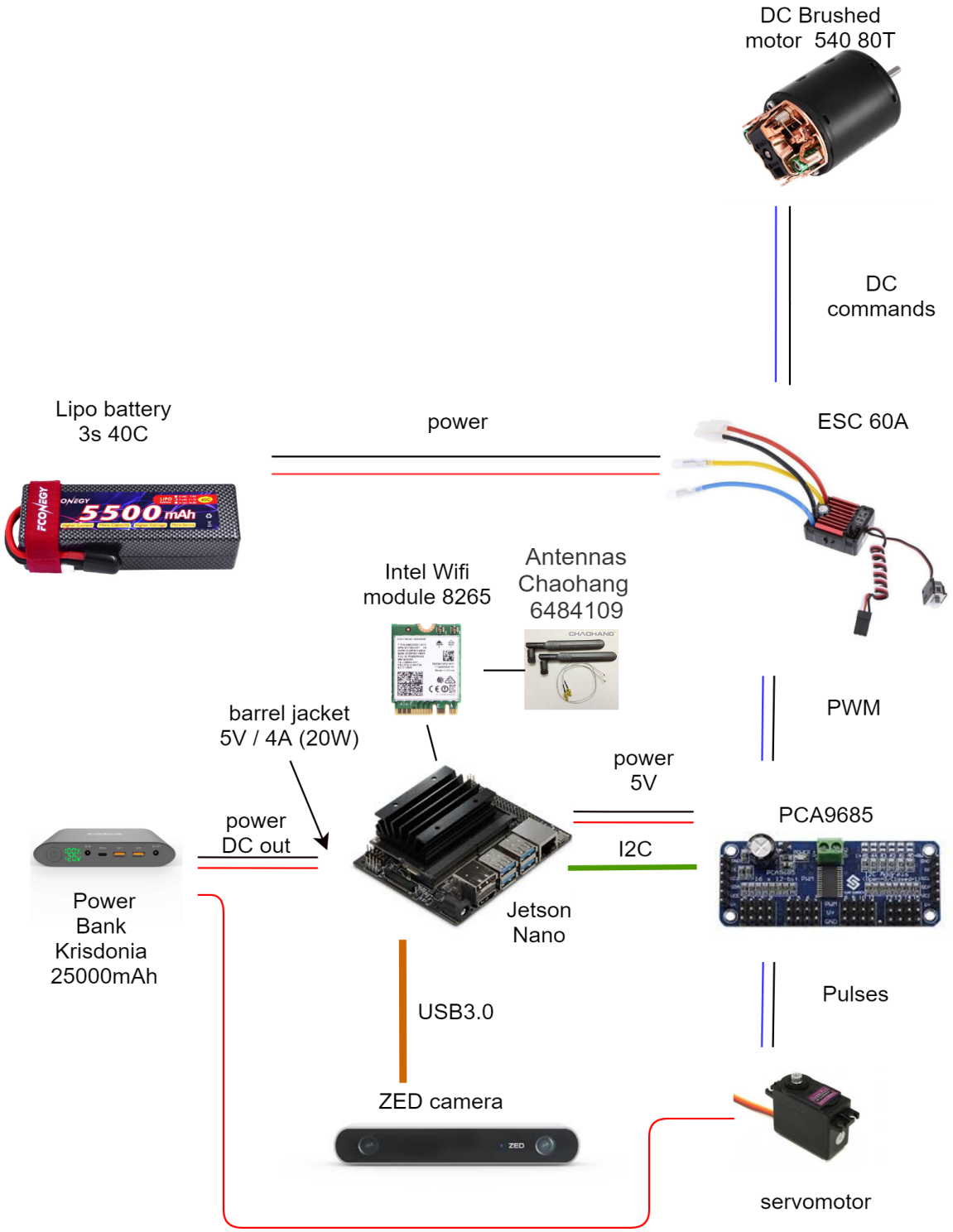


Figure 3.1 : Hardware architecture.

The figure 3.1 on the preceding page gives a high level understanding of the hardware organization. But how do the pins of the *Jetson Nano* and the *adafruit PWM shield* interact?

The figure 3.2 presents the *GPIO* layout of the *Jetson Nano*, and the figure 3.3 presents the pins layout of the *adafruit PWM shield*.

Jetson Nano J41 Header

http://www.neko.ne.jp/~freewing/

Pi GPIO#	Sysfs GPIO	Name	Pin	Pin	Name	Sysfs GPIO	Pi GPIO#
		3.3VDC *Power*	1	2	5.0VDC *Power*		
2		SDA1 *I2C Bus 1*	3	4	5.0VDC *Power*		
3		SCL1 *I2C Bus 1*	5	6	GND		
4	gpio216	AUDIO_MCLK	7	8	TXD0		14
		GND	9	10	RXD0		15
17	gpio50	UART2_RTS	11	12	DAP4_SCLK	gpio79	18
27	gpio14	SPI2_SCK	13	14	GND		
22	gpio194	LCD_TE	15	16	SPI2_CS1	gpio232	23
		3.3VDC *Power*	17	18	SPI2_CS0	gpio15	24
10	gpio16	SPI_MOSI	19	20	GND		
9	gpio17	SPI_MISO	21	22	SPI2_MISO	gpio13	25
11	gpio18	SPI1_SCK	23	24	SPI1_CS0	gpio19	8
		GND	25	26	SPI1_CS1	gpio20	7
(0)		ID_SDA *I2C Bus 0*	27	28	ID_SCL *I2C Bus 0*		(1)
5	gpio149	CAM_AF_EN	29	30	GND		
6	gpio200	GPIO_PZ0	31	32	LCD_BL_PWM	gpio168	12
13	gpio38	GPIO_PE6	33	34	GND		
19	gpio76	DAP4_FS	35	36	UART2_CTS	gpio51	16
26	gpio12	SPI2_MOSI	37	38	DAP4_DIN	gpio77	20
		GND	39	40	DAP4_DOUT	gpio78	21

JetsonHacks
https://www.jetsonhacks.com/nvidia-jetson-nano-j41-header-pinout/

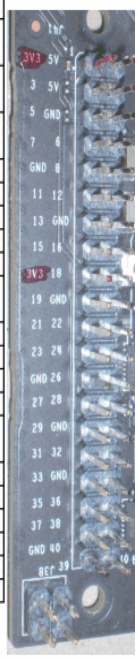


Figure 3.2 : *GPIO* pins of the *Jetson Nano*.

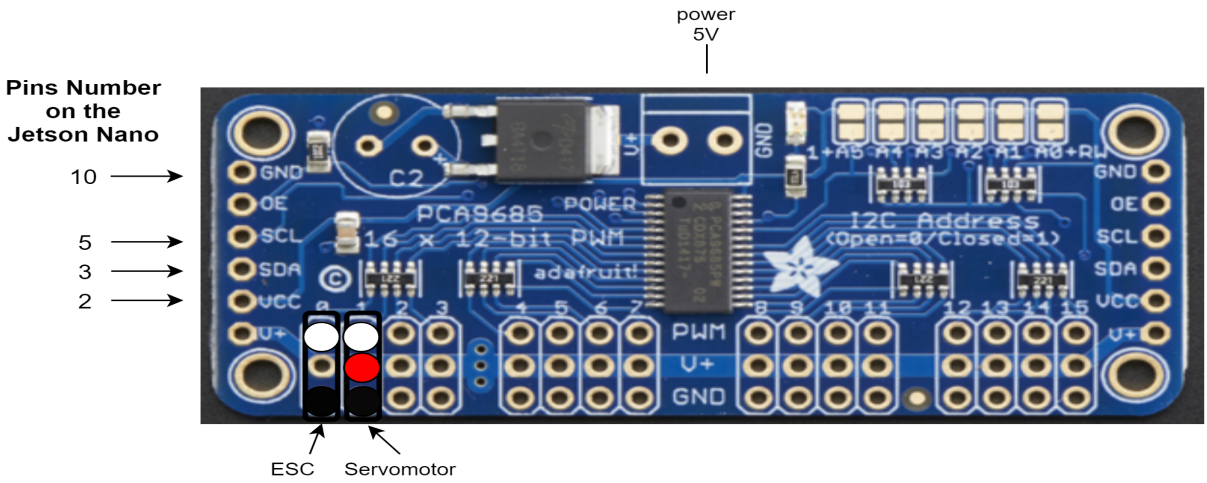


Figure 3.3 : Pins layout of the *adafruit PWM shield*.

3.1.2 Components list

The following list contains the links to each product bought in the hardware architecture presented figure 3.1 on the facing page:

- Brushed DC motor 540 80T.
- ESC 60A.
- Krisdonia power bank.

- Intel AC 8625.
- Chaohang Antennas.
- LiPo battery.

3.2 Low speed control

3.2.1 Motor

After having fixed the existing *RC car*, the speed was unfortunately too high to meet the likely processing speed of the embedded computer. The next step was then to find a way to slow down the robot.

The first constraint was to avoid changing the mechanics as much as possible. Changing the gear box, without buying a new *ESC* and a new motor could have been possible. Yet, it would have required to alter the entire mechanics of the *RC car* platform. Thus, this option was not considered.

How is it possible then not to modify the mechanics and at the same time to decrease the speed? If the mechanics was to be kept identical, that is to say that the power chain should remain untouched, the motor has to remain of the same dimensions. This is why as new motor, a *brushed DC motor* was in part chosen. Economically talking, a *brushed* motor is far cheaper in comparison to its counterpart the *brushless* motor. Yet, the speed has still not been decreased.

To understand how it is actually possible to slow down a *DC motor* it is essential to fathom the existing relation between the torque applied to the charge and the *RPM*, meaning Revolutions Per Minutes, or in other words the angular speed of the shaft of the motor. Basically, the angular speed is inversely proportional to the torque. Thus, in order to decrease the angular speed, a *brushed DC motor* with more torque had to be chosen. [McC18; Sca15]

3.2.2 Motor Controller

A *DC motor* without a controller is not very useful. At this stage, a controller had to be selected in order to set with commands the angular speed of the motor. Numerous strategies exist to regulate the speed of a *DC motor*. For instance, it is possible to use specific motor controllers [F1T], transistors, or *ESCs*. From all the projects introduced in the state or the art part 2.1 on page 9, a majority uses *ESCs*.

Yet, the race car of the *MIT*, Massachusetts Institute of Technology, employs *VESC*, which stands for *Vedder Electronic Speed Controller* [MIT]. Such *ESCs* were designed to provide more flexibility in the control of the low speeds. However, their price were far too high to be considered in this project.

All in all, a new brushed motor *ESC* was bought.

ESCs were first designed for *RC cars*, that is to say for remote control in general. This is why, the commands that should be given to them as inputs are signals transmitted by remote controller, *PWM signals*. In order to generate such signals the *GPIO* pins or outputs of the *Jetson Nano*, which stands for *General Purpose Inputs Outputs*, provide some relevant functionalities. The jetson nano has two channels that can generate *PWM* signals. Yet, the *PWM shield* PCA9685 of Adafruit is more reliable, since its function is exclusively to generate such signals. The selected solution was to use the *GPIO* pins to generate *I2C* signals to communicate with the *PWM shield*. *I2C* is a two-wire communication protocol that allows to address several devices or give several orders at the same time to external devices from a microcontroller. The *PWM shield* then outputs the right *PWM signals*.[Hac15; Nvid; Hacb]

3.3 Power consumption

3.3.1 How many?

The first question that should came to mind is not which type of battery should be bought, or even how much power the hardware requires, but how many batteries the robot needs. It is common in robotic applications and more extensively in any system to split the power supply into two parts.[Jet]

Actuators, that is to say for instance motors or servomotors, tend to require far more power than the embedded computer. Besides, should the actuators draw too much current, the embedded computer will switch off. It often happens that the actuators may induce current peaks. Thus, as a matter of safety, it is crucial to isolate the power source of the actuators from the one of the embedded computer. This explains the choice of two separate batteries.

By and large, the takeway is that the embedded computer must not, in any circumstance, be switched off, since the hardware will still apply the last commands stored in the more detrimental cases. It is highly recommended to use a self-powered *USB-Hub* to plug any subsequent device to the embedded computer, which may draw too much current from the embedded computer [Jet]. That advice was not applied in this project, and could be seen as a point of improvement.

All these choices were further underpinned by the projects presented in 2.1 on page 9.

3.3.2 Which ones?

On the one hand, regarding the motor, which is the actuator that draws the more power among the hardware of the robot, an independent *Li-Po* battery has been chosen. This choice was further promoted, for *Li-Po* batteries, in comparison to *NiMH* and *Ni-Cd* batteries, tend to be more efficient and start to gradually supersede them.

On the other hand, regarding the power source of the embedded computer, with the normal power input through *micro-usb* the supplied power reached a maximum of 10 Watts, and the jetson nano used to be turned off. As a matter of fact, after some tests, it appears that it was due to an under-powering of the *Jetson Nano*, although the *Jetson Nano* was not connected to any other device. Multiple ways exist to power the *Jetson Nano*. Powering the board through the *Power Jack*, which gives maximum 20 Watts, was the chosen solution. Hence, the battery *Krisdonia 25000mAh* was bought to meet that requirement [Kri]. In addition, it enables to plug more external devices to the *Jetson Nano* and fasten its overall performances.

The servomotor is still not powered. Servomotors are commanded via *PWM signals*. Those ones are generated with the *PWM shield*, also used for the communication with the *ESC*. The *PWM shield* needs to power the servomotor though. This is done by using the *V+* external power input of the *Adafruit shield*. Since the *Krisdonia 25000mAh* is powerfull enough with a *DC ouput* for the *Jetson Nano* and two other independent *USB power outputs* with a $5V/3A$, the servomotor is powered through the *PWM shield* with the *Krisdonia 25000mAh* battery. [Hac19; Nvic; eLi]

3.4 Remote control and processing

When the tracking robot is in mission, it is essential that at each time the information gathered and processed by the robot can be displayed and monitored by a human agent. For instance, it is useful to store the data on a remote computer, to display some curves, or simply to command the robot remotely when the tracker encounters some hurdles. Communication can be established via *SSH*, or Secure Shell, in a terminal environment. Yet, the robot needs to be able to connect to a network, and unfortunately, the *Jetson Nano* does not provide any way to do that. The most common way to achieve that is by using a *WiFi Module*. A huge variety of such devices can be find on the market, the question is then which one of those is the most suitable for the targeted application.

Another aspect of the decision should take into account that it would be more efficient and sustainable to be able to process the frames of the camera on a remote computer more powerful

than the *Jetston Nano*. However, to send frames via *WiFi* the *WiFi Module* must possess a relevant bandwidth and a faster than average communication speed. For this reason the module *8265NGW* of Intel was chosen, which delivers up to 867Mbps [Int].

3.5 Performances of the embedded computer

The performances of the embedded computer are crucial in any robotic application. At first the *Jetson Nano* was too slow to do anything else than acquiring the image of the *ZED camera*. In order to increase those performances a bigger *SD card* of about 64 GB had to be added.

Improving the processing speed of the *Jetson Nano* could also be done by artificially increasing the *RAM*, or Random Access Memory, of the board, which comes with a limited 4 GB of *RAM*. [Haca]

3.6 Unit Tests

Before trying to integrate anything, that is to say to use the hardware inside a *ROS* framework or to merge hardware and tracking, the hardware was tested independently.

Especially, regarding the actuators, being the servomotor and motor, a low-level library was written in python, *hardware.py*, which is presented in the appendix A on page 26. It belongs to the software of course, but since the low-level library is aimed to directly drive the hardware, this section can be considered inside the hardware part.

In fact, this library enables to create a *python object* for each piece of hardware. Each object provides some high level methods, which can then be used in a more complex *ROS* architecture. Parameters were also set in order to transfer a low-level command to a high-level command. As expected, the speed was sufficiently decreased for the tracking application.

Part 4

Software

The hardware selection has been discussed in the part 3 on page 11. Once the hardware architecture is defined and tested, the software part must be implemented in order to merge them together. The software comprises mainly two challenges : the realization of the tracking and the conception of the *ROS* architecture.

4.1 Tracking

4.1.1 Overview

As explained in 3.1.1 on page 11 and 2.2 on page 9, in this project the tracking algorithm is implemented with a deep neural network and takes as input the color frames of the *ZED camera*. The goal is to have a tracking algorithm able to follow a specific target in a sequence of frames.

The approach to the problem was to find a tracking algorithm or model which is pretrained. Basically, starting with the easiest solution and then, if needed, refining it. Another idea was also to have the model running in a controlled environment and then, once tested, interfacing it with *ROS*.

4.1.2 Goturn

Among the numerous and variegated tracking solutions presented in [GOTB19] the pretrained model *GOTURN* was chosen.

First of all, only the tracking algorithms that were implemented in *python* were selected for compatibilities issues. *GOTURN* was also the best choice regarding the ease of programming. It is implemented inside the *OpenCV* library, which is one of the most widely used computer-vision library worldwide, and which renders it far easier to integrate the tracking model in the *ROS* architecture afterward. [DH]

GOTURN is also sustainable and flexible, for it is possible to acquire the untrained model for a more specific application.[use]

4.1.3 ROS interface

Making the *GOTURN* tracker work inside the *ROS* framework was not that easy, although it was easier than the other available solutions. Running the *GOTURN* function provided in the *OpenCV* library needed to have a version of *OpenCV* higher than 3.4.2 [Mal18]. However, since *ROS* relies on *Python 2.7*, it only comes with an older version. I was not able to uninstall this library since it was a dependency for other *ROS* packages. The workaround was to create a *Python Environment* using *virtualenv* on *Ubuntu*, and then to specify this *Python* interpreter of this particular environment in the code of the tracker using the *shebang*, the first line of code where the interpreter is specified.

Once the right version of *OpenCV* was installed, and the *Python* interpreter suitably indicated, the *GOTURN* function needed to be interfaced with the *ROS* environment. *ROS* uses its own type of data, named *messages*. For instance an image or frame is an *Image.msg* message

[ROSa], which totally differs from the way *OpenCV* represents images. The bone of contention here was to be able to convert the *ROS* datatype into the *OpenCV* one. This is called *Interfacing*. Fortunately, the package *cv_bridge* provides that functionality [ROSb].

For sending frames between two devices connected to the same network the *ROS* package *image_transport* can be used [Lam]. The *ZED camera* already provides compressed image channels, using the *image_transport* package.

The code was then adapted to satisfy *ROS* functionalities.

4.1.4 Unit Tests

The tracking algorithm was first tested outside the *ROS* framework. Actually, the frame rate was around 50 *FPS*[1], half of what was advertised in the original paper [DH]. Yet, this may be due to the *OpenCV* implementation, which is not the initial one. These performances seem to suffice for the realization of the tracker though.

The next step was to interface this implementation with *ROS*. The tests comprised a static image, and the webcam of the computer. The performances were unchanged.

These tests also demonstrated that the *OpenCV GOTURN* model was really accurate at tracking humans. Yet, when it came to everything else, the model was lost easily. The assumption here may be that the model had been more trained on human targets. Regarding the reliability of the tracking given the nature of the target, the performances could be increased by retraining the model [use].

4.2 ROS

4.2.1 ROS Basics

ROS is a robotic *middleware*. Basically, it handles the interactions between the processes running on a robot.

In the *ROS* world, a process, for instance a program which sends commands to a motor or reads the outputs of a sensor, is named a *Node*. In that way, *ROS* abstractly depicts a robot as a graph, which comprises nodes which interact between each other.

How can the nodes interact? Each node *publishes* information on a channel named *topic*, which could be seen as a file which could be red by other nodes. In the *ROS* language, the node that writes the information is the *publisher* and the node that reads is the *subscriber*. This way of communicating is called *Asynchronous* since each node can read any given *topic* at any time.

Services enables to communicate in a *synchronous* way, that is to say that a *client* node will send a request to another node, the *server*. The *server* will then respond to that request.

What was just presented was the fundamentals of *ROS*, for further knowledge on the *ROS* philosophy please check these books : [Jos17; Kou16; Mah+18; FH17] .

4.2.2 Overview

First of all let's first explain the principles enforced by the *ROS* architecture presented figure 4.1 on the next page. The tracking robot has two independent commands as follows :

$$\begin{cases} \dot{x} = u_1 cos(\theta), \\ \dot{y} = u_1 sin(\theta), \\ \dot{\theta} = u_1 tan(u_2), \\ \dot{\delta} = \dot{u_2} \end{cases} \tag{4.1}$$

In these normalized state equations, inspired by [PLM], hypothesizing that the robot evolves on a plan, which is an acceptable approximation in the tests and applications conducted in this project, u_1 is the first command and is the speed of the robot which can be controlled

[1] Frames per second.

through the *ESC*, u_2 is the second command and is the bearing angle, which can be set via the servomotor. (x, y) are the coordinates of the robot on the plan. θ is the heading, and δ is the bearing angle. As the state equations show it, making the robot translate is done through u_1, and making the robot rotating is done through u_1 and u_2.

The sensor used for tracking, that is to say the *ZED* stereo camera gives basically the depth and the color information. The concept is to use the color information to control the bearing and to to use the depth information to control the speed of the robot. For each type of control the frames of the camera are processed, and then a command is generated.

A similar strategy was also put in place in another paper [CST].

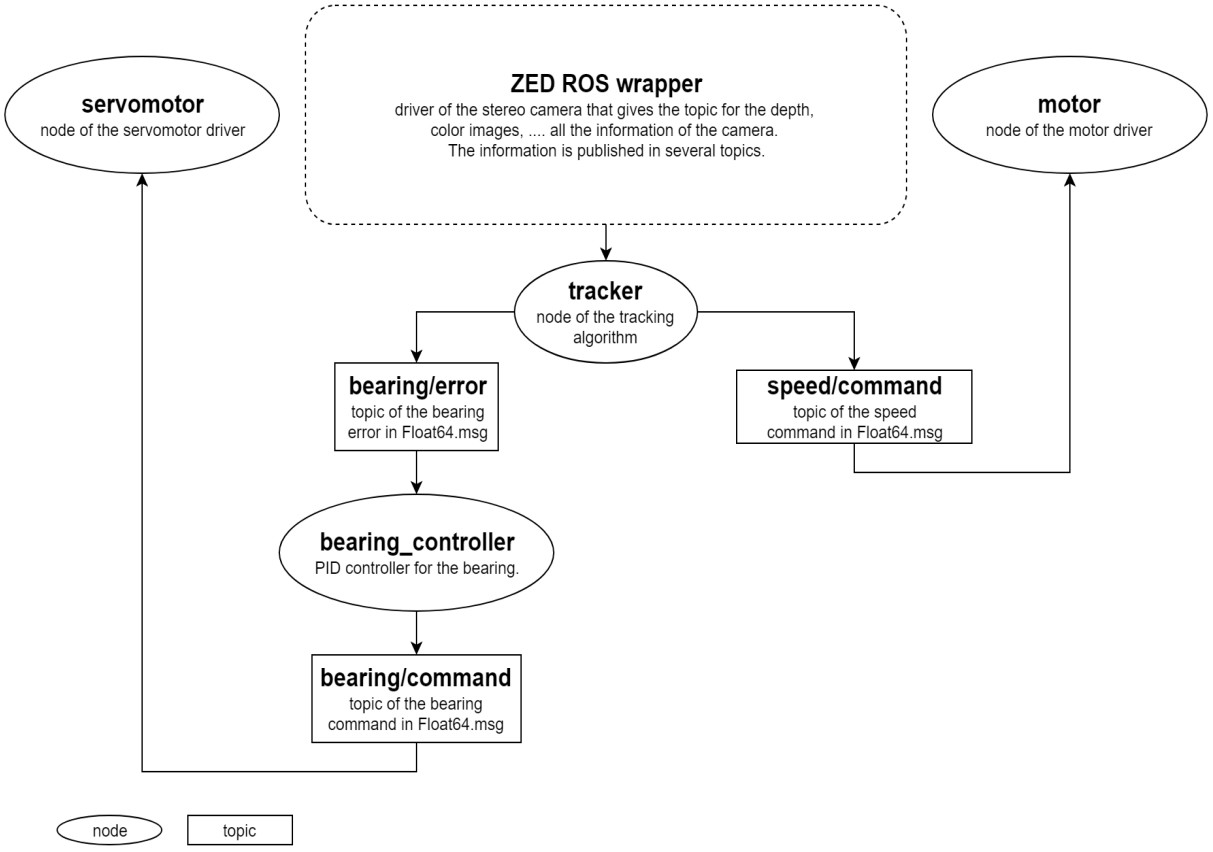


Figure 4.1 : *ROS* architecture.

One way to implement the *ROS* architecture figure 4.1 was to run only the nodes of the drivers on the *Jetson Nano* since each driver needs to run on the hardware, which enables to do any other processing on a remote computer for more computer power. The alternative is to run all the nodes presented on figure 4.1 on the *Jetson Nano* and use a remote computer only for introspection purpose.

4.2.3 Bearing regulation

To control the bearing of the robot the *RGB* images of the camera are used. The *GOTURN* algorithm processes an image and outputs the location of the target in the image by computing a bounding box. An error is then calculated by comparing the bounding box to the entire image. This error has the following expression:

$$e = \frac{m - m_{bb}}{X} \tag{4.2}$$

where m is the middle of the image regarding the horizontal axis or the x-axis. m_{bb} is the middle of the bounding box along the horizontal axis in the frame of the image. X is the width of the image, which is used as a normalization. A *PID*, or *proportional integrate derivative* controller

will tend to horizontally center the bounding box in the image, which will entail that the robot will be directed toward the target.

4.2.4 Speed regulation

In order to regulate the speed of the robot, the depth information of the stereo camera is processed. Once the bounding box is computed by the tracker, only the depth in the area of the bounding box is considered. The idea is then to regulate the speed regarding the object in the bounding box, which is the closest to the robot. Neither an error needs to be computed, nor a *PID* controller needs to be implemented. The command is directly the depth.

In practice, the command is not only the raw depth. The matrix of the depth in the bounding box, represented as D_{bb} in (4.3), is filtered by a function or kernel f which takes the median of the N lowest depth values in D_{bb}. Through that process the noise in the image is filtered and an equivalent depth value is computed for the closest obstacle inside the bouding box. The command implementing the dynamic following of the target for the *ESC* is as follows:

$$\begin{cases} u_1 = f(D_{bb}) \times K + \bar{u_1}, \\ u_{1,min},\ if\ f(D_{bb}) \leq d_{min}, \\ u_{1,max},\ if\ f(D_{bb}) \geq d_{max} \end{cases} \tag{4.3}$$

The speed will gradually increase if the target rolls away. If the target is too far, the speed will saturate, and reversely, if the target is too close, the robot will stop. The parameters in the command law have to be determined experimentally.

In case a potentially dangerous obstacle is not comprised in the bounding box, the tracker won't see it with the previous technique. Thus, a lidar sensor could be added on the front of the robot to detect any obstacle in case of emergency. Alternatively, the entire depth matrix could be filtered and processed to detect other obstacles outside the bounding box and stop the robot if needed.

4.2.5 Unit Tests

In order to test the *ROS* architecture, for the local part, without *Wifi* communication, everything was launched on a personal computer, *Asus G703VI*. The stereo camera was simulated, and the commands were not applied. The tests showed that the architecture worked as expected with a processing rate of 30 *FPS*, far more than acceptable. The next step was the integration discussed in the part 5 on the facing page.

Part 5

Integration

This part gives an analysis of the latest results obtained in this project. Basically, the part tackles the integration of the software in the hardware. The technique applied was to gradually remove each aspect of the simulation environment, adding more and more real constraints at each stride.

5.1 Integration of the ZED camera and the *Wifi* communication in the ROS architecture

5.1.1 Conditions

The goal of this integration test was to determine whether, while plugged to the wall-plug, that is to say without any power-bank or batterie, the tracking system was functioning correctly with a remote processing via *Wifi*.

The computer *Asus G703VI* was used as a remote computer and the *Jetson Nano* was used for the embedded computer on the robot. The actuators had not been integrated yet. Besides, *the ZED camera* was used to test whether the tracking process was working with a real stereo camera.

5.1.2 Analysis

The tracking rate obtained was about 5-6 *FPS* with a low resolution *VGA* and 4 *FPS* with a *720p* quality. The discrepancy between the 30 *FPS* obtained in simulation, as discussed in 4.2.5 on the preceding page, and the tracking rate in this test could be surprising at first. Yet, it is totally understandable considering that the frames of the *ZED camera* are compressed, and then sent to the remote computer where they are processed for tracking purpose, and finally the commands are sent to the robot. The commands were not applied to the hardware though.

Such a frame rate for the overhaul processing is though acceptable, and the latencies are not perceivable. The performances could be enhanced if the processing was done just on the *Jetson Nano*. It would however be more complicated to run very greedy deep learning models.

5.2 Integration of the remote control

5.2.1 Conditions

After having tested that the *ROS* architecture was functioning with a real stereo camera and through *Wifi* communication, the next step was to test whether the use of batteries was not undermining the process thanks to power issues. Several batteries were added : one for the motor, *Li-Po 3s*, and one for the embedded computer and the servomotor, however not the one ordered which had not arrived yet.

5.2.2 Analysis

The network established between the remote station and the robot was breaking down frequently. It appeared that the *Wifi stick* used, the *usb wifi dongle EP-AC1607*, since the ordered one had not been delivered at that time, had faulty drivers, which backfired when the *Jetson Nano* was underpowered.

The test demonstrated that powering the *Jetson Nano* through micro-usb was not sufficient. The ordered battery *Krisdonia 25000mAh* should fix the issue.

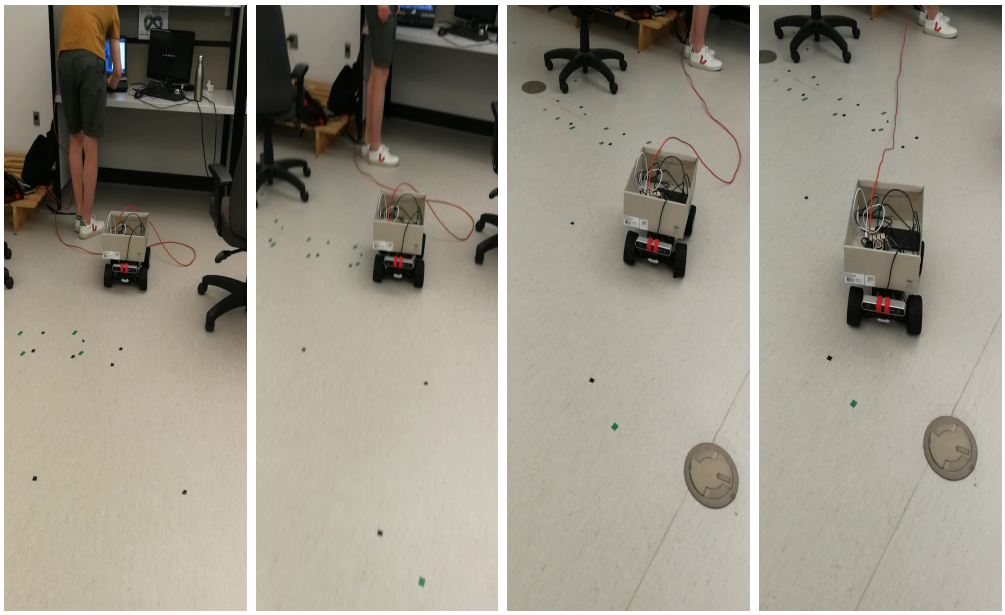
5.3 Integration of the actuators

5.3.1 Conditions

The goal of this test was to add the actuators to the *ROS* architecture and finally see the robot move. To avoid the network shutdowns experienced in the latter test, the communication between the two computers was handled via *Ethernet*, and the batteries were used.

5.3.2 Analysis

No network shutdowns were experienced, and the tracking rate was identical to the results obtained in the test presented in 5.1 on the preceding page. The test showed that the *Wifi stick* had in fact a faulty driver, and that power was also an issue to solve. The components of the hardware architecture in 3.1.1 on page 11 were chosen to meet these requirements.



(a) *(b)* *(c)* *(d)*

Figure 5.1 : Pictures of the tracker following the camera.

5.4 Integration of the tracking processing on the embedded computer

5.4.1 Conditions

After having noticed in the previous tests that the Wifi stick used had a faulty driver, the Wifi module ordered and presented in the *ROS* architecture figure 3.1 on page 12 did not arrive at that time, the idea was then to try to use the network communication as less as possible. In addition, sending the frames to the remote station for processing, hugely impacts the tracking rate, which directly depends on the bandwidth of the *Wifi* dongle or module. The whole goal

was to determine whether by processing the frames on the *Jetson Nano* the tracking rate was better or not.

To be as close as possible to the conditions of the test presented in 5.3 on the preceding page, an *Ethernet* communication was used to keep the basic *SSH* protocol between the robot and the remote station. But, here, unlike the test presented in 5.3 on the facing page, the frames had not been sent to the remote station for processing. Apart from that, the hardware was not added, only the performances of the tracking algorithm *Goturn* had been tested.

5.4.2 Analysis

The performances were about 1 *FPS* with the *5W* power mode and about 3 *FPS* with the *10W* power mode, which remain still far below the performances obtained when the frames were processed by the remote station.

Yet, unlike the network communication which is slowing down the tracking processing and which is hard to speed up, the *Goturn* algorithm could be sent to the *Maxwell GPU* of the *Jetson Nano* for better performances. This is called *GPU acceleration*. The performances were obtained here for an execution on the *two-core CPU* of the *Jetson Nano*. Using *GPU* acceleration should bring far better performances [Ste; eli].

However, sending the model to the *GPU* is not trivial. Since *Goturn* is an *OpenCV* function, it is impossible to use conventional tools, such as *PyCuda, tensorflow-gpu, PyTorch* to send the code to the *GPU*. The previous tools would require to hard-code the *Goturn* network. [Nvia]

On the backend, *OpenCV* uses *OpenCL* for *GPU* acceleration, and on the *Jetson Nano* *Cuda*, which is suited for *Nvidia*'s systems uses *OpenCL*, and so *OpenCV* acceleration. The basic *GPU* acceleration of *OpenCV* should be operating. It is possible to add some flags, while building the *OpenCV* library from source to activate the *GPU* acceleration.[Mal; For; Jun; eli]

Apart from hard-coding the tracking algorithm, another solution would be to use *C++* as a programming language. The idea is then to code a *C++* wrapper to execute in the back-end a python code. *C++* gives room for more flexibility, and the *Jetson Nano* was primarily designed to use *C++*. These workarounds have not been tested and present promising results regarding the performances.

The power supply of the *Jetson Nano* influences the frame rate. Once the battery *Krisdonia 25000mAh* is delivered, it would then be possible to double the power supply and, as a consequence, to increase the processing speed.

Part 6

Setup

This part presents the material needed and the necessary conditions to replicate the results[1] of this project. Should you need some explanations on the project, please refer to parts 1 on page 7, 2 on page 9, 3 on page 11, and 4 on page 17.

A detailed github presents the setup and instructions to follow in order to replicate the first prototype of the tracker developed during this project.

[1] cf. 5 on page 21.

Conclusion

The tracker robot was able to track a specified target, human targets were followed more accurately. The *ROS* architecture that has been implemented is flexible and could be used for further improvements.

Regarding the performances, in comparison to a person following robot [CST] tracking at a rate of 20 *FPS*, the tracker implemented in this project tracks at a maximum of 6 *FPS*. The performances could be improved though. The person following robot was implemented in *C++* in comparison to the tracker in this project, which was implemented entirely in *Python*.

It is also important to mention that not all the components ordered, and presented on figure 3.1 on page 12, had arrived before the end of the project. Having the best components would have certainly improved the performances of the tracking robot. Especially, the tracker would need to be tested with the *Krisdonia 25000mAh* power bank and the *Intel 8265 Wifi module*.

Another point of refinement could be the training of the *GOTURN* model to make it more accurate, the realization of a personal deep learning network, or the test of another one.

For any further implementation, and especially if realizing a personal deep learning model is the goal, switching to *ROS2* could be a wise move, since *ROS2* is coded in *Python 3*, and would make it easier to use the common deep learning libraries out of the box, such as *PyTorch* or *Tensorflow*. For any deep learning implementation with the *Jetson Nano*, it is recommended to read through the *Nvidia* tutorials first[Nvif].

To sum up, the latest tested prototype, as presented in the github, did not implement the speed control. The idea was to obtain first acceptable performances for the tracking algorithm, which is the juggernaut of the robot. Several things should be tested for any motivation to continue this project:

- Transpose the *ROS* architecture in *ROS2*: it would then be easier for any *Python* support.
- Implementing in *C++*: the performances of *C++* compared to those of *Python* are unequivocal. The *Jetson Nano* was designed to be more suitable for *C++* programming. It would be more flexible for *GPU* acceleration.
- Hard-code the tracking model in *PyTorch, tensorflow*, so that it would be possible to use *GPU acceleration* on the *Jetson Nano*, cf. integration presented in 5.4 on page 22.
- Test the *Wifi* module and the *Krisdonia 25000mAh* of the *ROS* architecture presented in 3.1 on page 12, when delivered.
- Test the speed control as presented in figure 4.1 on page 19 and in 4.2.3 on page 19.

Part A

Hardware Library

Listing A.1: Hardware library : file *hardware.py*

```
#!/usr/bin/env python3

import board
import adafruit_pca9685
import busio
import time
import os

class Servomotor:

        def __init__(self):
                print("This may take a few seconds . . . ")
                i2c = busio.I2C(board.SCL, board.SDA)
                self.pwm = adafruit_pca9685.PCA9685(i2c)
                self.pwm.frequency = 50
                # values
                self.leftMax = 100
                self.rightMax = 0
                self.straight = 50
                self.angle = None
                self.set_bearing(self.straight)
                print("Servomotor initialization SUCCESS")

        def test(self):
                accuracy = 10
                for angle in range(0,100 + accuracy,accuracy):
                        self.set_bearing(angle)
                        time.sleep(1)
                self.set_bearing(self.straight)

        def terminal_test(self):
                value = input("Angle [0-100]: ")
                while value != 'q':
                        self.set_bearing(float(value))
                        value = input("Angle [0-100]: ")
                        print("value entered : " + value)

        def set_bearing(self,angle):
                self.pwm.channels[1].duty_cycle = int(3932+ angle*2620/100)
                self.angle = angle

class Motor:

        def __init__(self):
                print("This may take a few seconds . . . ")
                i2c = busio.I2C(board.SCL, board.SDA)
                self.pwm = adafruit_pca9685.PCA9685(i2c)
                self.pwm.frequency = 50
```

```
49                  # values
50                  self.off = 50
51                  self.forwardMin = 68
52                  self.speed = None
53                  # motor setup
54                  self.setup()
55                  print("Motor initialization SUCCESS")
56
57          def stop(self):
58                  self.set_speed(self.off)
59
60          def setup(self):
61                  self.set_speed(50)
62                  time.sleep(1)
63
64          def test(self):
65                  for speed in range(0,100,10):
66                          self.set_speed(speed)
67                          time.sleep(1)
68                  self.set_speed(self.forwardMin)
69                  time.sleep(4)
70                  self.stop()
71
72          def terminal_test(self):
73                  value = input("Speed [0-100]: ")
74                  while value != 'q':
75                          self.set_speed(float(value))
76                          value = input("Speed [0-100]: ")
77                          print("value entered : " + value)
78
79          def set_speed(self,speed):
80                  self.pwm.channels[0].duty_cycle = int(3932+ speed*2620/100)
81                  self.speed = speed
82
83  if __name__ == "__main__":
84          print("test")
85          b = Servomotor()
86          m = Motor()
87          b.test()
88          m.terminal_test()
```

Part B

ROS

The code presented in this section is the code of the first prototype of the tracker, you can find it in this github too.

B.1 On the robot

Listing B.1: Driver : file *motor.py*

```
#!/usr/bin/env python3

"""
Node of the motor driver.
"""

import rospy
from hardware import * # library for the servomotor and the motor.


def main():
    # Node initialization.
    rospy.init_node('motor', log_level=rospy.DEBUG)
    rospy.loginfo("Motor node initialized.")
    # Process.
    global motor
    motor = Motor()
    #motor.terminal_test()  # just for test.
    motor.set_speed(motor.forwardMin)
    #rospy.spin()

if __name__ == "__main__":
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

Listing B.2: Driver : file *servomotor.py*

```
#!/usr/bin/env python3

"""
Node of the servomotor driver.
"""

import rospy
from std_msgs.msg import Float64
from hardware import * # library for the servomotor and the motor.

def callback(msg):
        # set the new command for the servomotor.
        rospy.loginfo("[" + rospy.get_caller_id() + "] Command applied : " + str(msg.data))
```

```
        servomotor.set_bearing(msg.data)

def main():
    # Node initialization.
    rospy.init_node('servomotor', log_level=rospy.DEBUG)
    rospy.loginfo("Servomotor node initialized.")
    # Process.
    global servomotor
    servomotor = Servomotor()
    # Subscriber.
    sub = rospy.Subscriber("bearing/command",Float64, callback)
    # Spinning.
    rospy.spin()

if __name__ == "__main__":
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

Listing B.3: Launch file for the robot : file *robot.launch*

```
<launch>
        <!-- ZED camera driver -->
        <include file="$(find zed_wrapper)/launch/zed.launch">
        <!-- Motor driver -->
        <node pkg="robot_tracker" name="motor" type="motor.py" output="screen">
        </node>
        <!-- Servomotor driver -->
        <node pkg="robot_tracker" name="servomotor" type="servomotor.py" output="screen">
        </node>
</launch>
```

B.2 On the remote computer

Listing B.4: Tracker : file *tracker.py*

```
#!/home/arnaud/opencv_py_env/bin/python2.7
"""

Node that reads the frames published by
the zed camera on a remote computer
converts it to an openCV format and
processes it.

/!\ Launch this node with a terminal open in the folder
where this file is located. (The implementation of
Goturn needs it)

"""
# ROS libraries
import rospy
from cv_bridge import CvBridge
# Other libraries
import cv2, sys, os
OPENCV_VERSION = cv2.__version__
# message
from sensor_msgs.msg import Image
from sensor_msgs.msg import CompressedImage
import threading
from std_msgs.msg import Float64

##############################################################################################

class GoturnNode:

    def __init__(self):
        self.tracker = cv2.TrackerGOTURN_create()
        self.init_flag = False  # False, the tracker was not initialized.
        self.check_model()

    def check_model(self):
        # root dir.
        ROOT_DIR = os.path.dirname(os.path.abspath(__file__)) # since isfile depends on the terminal
        # location.
        rospy.loginfo("Root directory : %s", ROOT_DIR)
        if not (os.path.isfile(ROOT_DIR + '/goturn.caffemodel') and os.path.isfile(ROOT_DIR + '/goturn.prototxt')):
            errorMsg = " Could not find GOTURN model in current directory.\n Please ensure goturn.caffemodel and goturn.prototxt are in th
            rospy.logfatal(errorMsg)
            sys.exit()
        # Checks if the terminal launching the tracker is launched in the folder where
        # the models are located.
        if not (os.path.isfile('goturn.caffemodel') and os.path.isfile('goturn.prototxt')):
```

```
47              rospy.logfatal(" The terminal has to be launched where the models are located.")
48              sys.exit()
49
50      def track(self, img):
51          """
52          Tracking process, outputs the bounding box.
53          The initialization requires to launch the node (terminal)
54          in the derictory where the models are.
55          """
56          #rospy.loginfo("FRAME NUMBER %s", self.identifier)
57          if self.init_flag == False:
58              # initialization.
59              #bbox = (276, 23, 86, 320)
60              bbox = cv2.selectROI(img, False)  # some issues, it freezes after.
61              flag = self.tracker.init(img, bbox)
62              if not(flag):
63                  rospy.logfatal("Cannot initialize the tracker .... ")
64                  sys.exit()
65              else:
66                  rospy.loginfo("initialization SUCCESS.")
67              self.init_flag = True  # initialization done.
68          # tracking process : initialization already done.
69          else:
70              rospy.loginfo("Goturn processing at time %s.", rospy.get_time())
71              # Application of teh goturn algorithm.
72              flag, bbox = self.tracker.update(img)
73              if flag: # success.
74                  rospy.loginfo("New bounding box : %s", bbox)
75                  p1 = (int(bbox[0]), int(bbox[1]))
76                  p2 = (int(bbox[0] + bbox[2]), int(bbox[1] + bbox[3]))
77                  cv2.rectangle(img, p1, p2, (255, 0, 0), 2, 1)
78                  cv2.imshow("Tracking", img)
79                  cv2.waitKey(1)
80                  # Errors computation.
81                  # bearing error.
82                  width = img.shape[1]
83                  bb_mid_x = bbox[0] + bbox[2]/2
84                  img_mid_x = width/2
85                  e_bearing = (img_mid_x - bb_mid_x)/width
86                  # Publishing errors.
87                  pub_err_bearing.publish(e_bearing)
88              else:
89                  rospy.logwarn("FAILURE IN THE TRACKING .... ")
90
91
92      def process(self,msg):
93          # conversion to opencv.
94          img_cv2 = conversion(msg)
95          #display(img_cv2)
96          # goturn processing
97          self.track(img_cv2)
98
99  ##########################################################################################################
100
101 def conversion(msg):
102     """
103     Convertion to OpenCV standards.
104     """
105     rospy.loginfo("Conversion at %s.. ", rospy.get_time())
106     # Image processing.
107     #encoding = msg.encoding
108     encoding = 'bgr8'
109     #rospy.loginfo("NEW CALLBACK : Image processed at time %s, with encoding type %s.", rospy.get_time(), encoding)
110     # convert sensor_msgs/Image to OpenCV Image
111     bdg = CvBridge()
112     # convertion of the image into openCV format.
113     # conservation of the same encoding type.
114     #img_cv2 = bdg.imgmsg_to_cv2(msg, encoding)
115     img_cv2 = bdg.compressed_imgmsg_to_cv2(msg, encoding)
116     return img_cv2
117
118 def display(img):
119     cv2.imshow("Recovered image", img)
120     cv2.waitKey(1)  # waits.
121
122 def check_dependencies():
123     # TODO : forcing the opencv version.
124     pass
125
126 ##########################################################################################################
127
128 def main():
129     # Node initialization.
130     rospy.init_node('tracker', log_level=rospy.DEBUG, anonymous = True)
131     rospy.loginfo("Subscriber starts.")
132     # Publishing.
133     global pub_err_bearing
134     pub_err_bearing = rospy.Publisher("bearing/error", Float64, queue_size = 1)
135     # Initialization of the error to avoid any unexpected behavior of the
136     # hardware.
137     pub_err_bearing.publish(0)
138     # Subscriptions.
139     #buffer_size = 2**30 # maximum possible buffer size
140     #rospy.loginfo("Subscriber's buffer sizer : %s", buffer_size)
141     #sub = rospy.Subscriber("initial_image",Image,callback, queue_size = 1, buff_size = buffer_size)
142     # tracker initialization.
143     global node  # makes the node structure available for each method.
144     node = GoturnNode()
145     # spinning loop for the frame.
146     while not rospy.is_shutdown():
147         t_start = rospy.get_rostime()
148         msg = rospy.wait_for_message("/zed/zed_node/rgb/image_rect_color/compressed",CompressedImage)  # gets a new message / frame.
149         rospy.loginfo("Frame timestamp :  %s.", msg.header.stamp)
```

```
150         rospy.loginfo("Frame processing starts at %s.", rospy.get_rostime())
151         node.process(msg)  # frame processing.
152         t_end = rospy.get_rostime()
153         FPS = 1/(t_end-t_start).to_sec()
154         rospy.loginfo("FPS : %s", FPS)
155
156 if __name__ == "__main__":
157     try:
158         main()
159     except rospy.ROSInterruptException:
160         pass
```

Listing B.5: *PID* controller for the bearing : file *bearing_controller.py*

```
1  #!/home/arnaud/opencv_py_env/bin/python2.7
2  """
3  Node of the controller of the servomotor.
4  """
5  import rospy
6  from std_msgs.msg import Float64
7
8  # parameters of the servomotor control.
9  # u is equivalent to an angle (bearing).
10 u0 = 50    # straight.
11 umin = 0    # right
12 umax = 100    # left
13
14 # PID parameters
15 Kp = 100
16
17 def callback_bearing(msg):
18     rospy.loginfo(rospy.get_caller_id() + "I heard %s at %s", msg.data, rospy.get_rostime())
19     # command computation.
20     e = msg.data  # error regarding the bearing.
21     u = u0 + Kp*e
22     # Saturation.
23     if u > umax:
24         u = umax
25     elif u < umin:
26         u = umin
27     # Publishing.
28     pub.publish(u)
29     rospy.loginfo(rospy.get_caller_id() + " Command : %s , at %s", u, rospy.get_rostime())
30
31 def main():
32     # Node initialization.
33     rospy.init_node('bearing_controller', log_level=rospy.DEBUG, anonymous = True)
34     rospy.loginfo("bearing_controller starts.")
35     # Subscription.
36     rospy.Subscriber("bearing/error", Float64, callback_bearing)
37     # Publisher
38     global pub
39     pub = rospy.Publisher("bearing/command", Float64, queue_size = 1) # bearing command.
40     # Publishing.
41     # TODO : commands
42     # Spinning.
43     rospy.spin()
44
45 if __name__ == "__main__":
46     try:
47         main()
48     except rospy.ROSInterruptException:
49         pass
```

Listing B.6: Launch file for the remote station : file *remote.launch*

```
1 <launch>
2         <node pkg="remote_tracker" name="tracker" type="tracker.py" output="screen">
3         </node>
4         <node pkg="remote_tracker" name="bearing_controller" type="bearing_controller.py" output="screen">
5         </node>
6 </launch>
```

Bibliography

[Ada] Adafruit. *Adafruit 16-Channel 12-bit PWM/Servo Driver - I2C interface - PCA9685*. https://www.adafruit.com/product/815 (cit. on p. 11).

[Ama] Amazon. *Amazon Deep Racer*. https://aws.amazon.com/deepracer/ (cit. on p. 9).

[AS18] Martin Jagersanda Abhineet Singha. "Modular Tracking Framework: A unified approach to registration based tracking". In: *Computer Vision and Image Understanding, Elsevier* (2018) (cit. on p. 10).

[Bar] Nikodem Bartnik. *Object Tracking Robot*. link to project. (Cit. on p. 7).

[Bje16] Marko Bjelonic. *YOLO ROS: Real-Time Object Detection for ROS*. https://github.com/leggedrobotics/darknet_ros. 2016–2018 (cit. on p. 9).

[Car] Monster Cars. *Instructions manual*. link. H1 model. (cit. on p. 7).

[CST] Bao Xin Chen, Raghavender Sahdev, and John K. Tsotsos. *Integrating Stereo Vision with a CNN Tracker for a Person-Following Robot*. York University, Canada (cit. on pp. 10, 19, 25).

[Dan] Steven Daniluk. *Ghost II : controlling an RC car with a computer*. https://medium.com/hackernoon/ghost-ii-controlling-an-rc-car-with-a-computer-b1d1849d9e43 (cit. on p. 9).

[DH] Silvio Savarese David Held Sebastian Thrun. *Learning to Track at 100 FPS with Deep Regression Networks*. paper in pdf. (Cit. on pp. 17, 18).

[Diy] DiyRoboCars. *Donkey Car*. https://docs.donkeycar.com/ (cit. on p. 9).

[eli] elinux. *Installing OpenCV*. link to article. (Cit. on p. 23).

[eLi] eLinux.org. *Jetson Nano documentation*. link (cit. on p. 15).

[F1T] University Of Pennsylvania F1Tenth. *F1Tenth race car*. http://f1tenth.org/build.html (cit. on pp. 9, 14).

[FH17] Carol Fairchild and Dr. Thomas L. Harman. *ROS Robotics By Example*. Packt, 2017 (cit. on p. 18).

[For] Nvidia Developer Forum. *OpenCV, CUDA, Python with Jetson Nano*. link to FAQ. (Cit. on p. 23).

[GOTB19] GOT 10k : Generic Object Tracking Benchmark. *Leaderboard : Up-to-date generic object tracking performance*. http://got-10k.aitestunion.com/leaderboard. 2019 (cit. on pp. 7, 9, 17).

[Haca] Jetson Hacks. *Jetson Nano – Use More Memory!* link. (Cit. on p. 16).

[Hacb] Jetson Hacks. *NVIDIA Jetson Nano J41 Header Pinout*. https://www.jetsonhacks.com/nvidia-jetson-nano-j41-header-pinout/ (cit. on p. 14).

[Hac15] Jetson Hacks. *PWM Servo Driver Board – NVIDIA Jetson TK1*. https://www.jetsonhacks.com/2015/10/14/pwm-servo-driver-board-nvidia-jetson-tk1/. 2015 (cit. on p. 14).

[Hac19] Jetson Hacks. *Jetson Nano – Use More Power!* https://www.jetsonhacks.com/2019/04/10/jetson-nano-use-more-power/. 2019 (cit. on p. 15).

[Int] Intel. *Intel Dual Band Wireless AC - 8265.* pdf documentation. (Cit. on p. 16).

[Jet] JetsonHacks. *RACECAR/J Initial Assembly.* https://www.jetsonhacks.com/2018/01/28/racecar-j-initial-assembly/ (cit. on p. 15).

[Jos17] Lentin Joseph. *ROS Robotics Projects.* Chapter 2. Packt, 2017 (cit. on pp. 10, 18).

[Jun] JK Jung. *Installing OpenCV 3.4.6 on Jetson Nano.* link to article. (Cit. on p. 23).

[Kou16] Anis Koubaa. *Robot Operating System : the complete reference (volume 1 and 2).* Springer, 2016 (cit. on pp. 7, 18).

[Kri] Krisdonia. *Krisdonia Portable Laptop Charger (TSA-Approved) 92.5Wh/25000mAh Travel Laptop Power Bank External Battery Bank for HP, Dell, Lenovo and Other laptops.* amazon.com (cit. on p. 15).

[Lam] Victor Lamoine. *Documentation of the package image_transport.* link. (Cit. on p. 18).

[Mah+18] Anil Mahtani et al. *ROS Programming: Building Powerful Robots: Design, build and simulate complex robots using the Robot Operating System.* Packt, 2018 (cit. on p. 18).

[Mal] Satya Mallick. *OpenCV Transparent API.* link to article. (Cit. on p. 23).

[Mal17] Satya Mallick. *Object Tracking using OpenCV (C++/Python).* link to course on learnOpenCV. 2017 (cit. on p. 9).

[Mal18] Satya Mallick. *GOTURN : Deep Learning based Object Tracking.* link to course on learnOpenCV. 2018 (cit. on p. 17).

[McC18] Gordon McComb. *Robot Builder's Bonanza, 5th Edition.* McGraw-Hill Education, 2018 (cit. on p. 14).

[MIT] MIT. *MIT racecar.* github project & official documentation (cit. on p. 14).

[MSV19] Janakiram MSV. *NVIDIA Brings Affordable GPU to the Edge with Jetson Nano.* https://thenewstack.io/nvidia-brings-affordable-gpu-to-the-edge-with-jetson-nano/. 2019 (cit. on p. 7).

[nai] Github repository by the user naisy. *ROSCAR.* https://github.com/naisy/roscar (cit. on p. 9).

[Nvia] Nvidia. *GPU Accelerated Computing with Python.* link to article. (Cit. on p. 23).

[Nvib] Nvidia. *Jetbot : an educational robot.* https://github.com/NVIDIA-AI-IOT/jetbot/wiki (cit. on p. 9).

[Nvic] Nvidia. *Jetson Nano Developer Kit Guide.* pdf file. (Cit. on p. 15).

[Nvid] Nvidia. *Jetson.GPIO - Linux for Tegra.* https://github.com/NVIDIA/jetson-gpio. github hosted documentation. (cit. on p. 14).

[Nvie] Nvidia. *Kaya.* https://docs.nvidia.com/isaac/isaac/apps/tutorials/doc/assemble_kaya.html (cit. on p. 9).

[Nvif] Nvidia. *Two Days to a Demo.* link to tutorial. (Cit. on p. 25).

[PLM] Romain Pepy, Alain Lambert, and Hugues Mounier. *Path Planning using a Dynamic Vehicle Model.* Université Paris Sud XI, France (cit. on p. 18).

[ROSa] ROS. *sensor_msgs/Image Message.* link. (Cit. on p. 18).

[ROSb] ROS. *cv_brdige package.* link to documentation. (Cit. on p. 18).

[Sca15] Matthew Scarpino. *Motors for Makers: A Guide to Steppers, Servos, and Other Electrical Machines.* Que Pub, 2015 (cit. on p. 14).

[Ste] Sam Sterckval. *Google Coral Edge TPU vs NVIDIA Jetson Nano: A quick deep dive into EdgeAI performance.* link to article. (Cit. on p. 23).

[ste] stereloabs. *ZED camera documentation.* https://www.stereolabs.com/docs/getting-started/ (cit. on p. 11).

[Sun] SunFounder. *Raspberry Pi Smart Robot.* https://www.sunfounder.com/picar-s-kit.html (cit. on p. 9).

[use] nrupatunga github user. *Goturn-Py.* link. (Cit. on pp. 17, 18).

[VJZ14] D. M. Vo, L. Jiang, and A. Zell. "Real time person detection and tracking by mobile robots using RGB-D images". In: *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*. 2014, pp. 689–694. DOI: 10.1109/ROBIO.2014.7090411 (cit. on p. 7).

[Wan+] Mengmeng Wang et al. *Real-time 3D Human Tracking for Mobile Robots with Multisensors.* link to paper. (Cit. on p. 7).