DALHOUSIE UNIVERSITY

ENSTA Bretagne

ENSTA Bretagne
2, rue François Verny
29806 BREST cedex
FRANCE
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr

report

CI2020
second year

July 30, 2019

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

Resume

Introduction

THIS document is the report of the project on which I worked during my summer internship in 2019 at Dalhousie University. The main goal was to build a tracking robot with the middleware ROS and with deep learning techniques applied to tracking.

The first part presents how to setup the robot.

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 robots have been implemented in the previous years, few have used the adaptive, versatile, and now standard robotic Middleware *ROS*[1]. In addition, even fewer have ventured combining *ROS* and *Deep Learning* techniques.

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*[2], 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, *AI* [3] 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[4] by applying color filters. *AI* 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] Which stands for *Robot Operating System*.

[2] See 3.1.1 on page 11.

[3] Artificial Intelligence, which comprises *Deep Learning*.

[4] image.

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 [5].

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 type 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 beforehand. 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 15.
- 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 16.

[5] Especially, some driver issues are sometimes really hard to pinpoint.

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 15.

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* [Nvia], *Kaya* [Nvid]. These robots are both designed by *Nvidia*, the designers of the *Jetson* board series[1]. It is notably interesting to take as example their hardware selection since they use the same embedded computer the tracker use.

Other *RC-cars*[2] 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, 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 [3], 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 an enumeration and ranking of the current most effective techniques [GOTB19].

For sure, in order to automate the robot, a detection algorithm could precede the tracking.

[1] Jetson Nano, Xavier, TX1, TX2 . . .
[2] *Remote Control Car.*
[3] Given by a camera for instance.

2.3 Using ROS

Developing with *ROS* gives also 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, *SLAM* [4]. 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.

[4] Simultaneous Localization And Mapping.

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 chose wisely 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*[1], 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 **??**, which shows each component and how each interacts with each other.

Let's explain a bit what are the crucial parts that constitute a tracking robot. 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.

3.1.2 Components list

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

[1]More is explained on that in the section 3.2.

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*[2], 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 by 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*[3] 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*[4] pins or outputs of the *Jetson Nano* 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*[5] signals to communicate with the *PWM shield*. The *PWM shield* then outputs the right *PWM signals*.[Hac15; Nvic; 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[6], the embedded computer will switch off. 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.

[2] Revolutions Per Minutes.

[3] Massachusetts Institute of Technology, USA.

[4] Which stands for *General Purpose Inputs Outputs*.

[5] Two-wire communication protocol that allows to address several devices or give several orders at the same time to external devices from a microcontroller.

[6] It often happens that the actuators may induce current peaks.

By and large, the takeway is that the embedded computer must not, in any circumstances, be switched off, since the hardware will still apply the last commands stored in the more detrimental cases. It is also 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 power supplied was maximum 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; Nvib; 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*[7] in a terminal environment. Yet, the robot needs to be able to be connected 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.

[7] Secure Shell.

Improving the processing speed of the *Jetson Nano* could also be done by artificially increasing the *RAM*[8] of the board, which comes with a limited 4 GB of *RAM*. The latter was not put in place, it is highly recommended in the future though. [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 20. 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.

[8] Random Access Memory.

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 then 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. 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 dependencies 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.

4.1.4 Unit Tests

4.2 ROS

4.2.1 ROS Basics

4.2.2 Overview

4.2.3 Speed regulation

4.2.4 Bearing regulation

4.2.5 Unit Tests

Part 5

Results : 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.

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 15.

6.1 Hardware setup

✗ Computers : Jetson Nano, Ubuntu 18.04, remote computer Ubuntu 18.04.

6.2 Software setup

[1] cf. 5 on the previous page.

Conclusion

Part A

Hardware Library

```python
#!/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
                # values
                self.off = 50
```

```python
            self.forwardMin = 68
            self.speed = None
            # motor setup
            self.setup()
            print("Motor initialization SUCCESS")

        def stop(self):
            self.set_speed(self.off)

        def setup(self):
            self.set_speed(50)
            time.sleep(1)

        def test(self):
            for speed in range(0,100,10):
                self.set_speed(speed)
                time.sleep(1)
            self.set_speed(self.forwardMin)
            time.sleep(4)
            self.stop()

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

        def set_speed(self,speed):
            self.pwm.channels[0].duty_cycle = int(3932+ speed*2620/100)
            self.speed = speed

if __name__ == "__main__":
        print("test")
        b = Servomotor()
        m = Motor()
        b.test()
        m.terminal_test()
```

Part B

ROS

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).

[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).

[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 p. 15).

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

[eLi] eLinux.org. *Jetson Nano documentation*. link (cit. on p. 13).

[F1T] University Of Pennsylvania F1Tenth. *F1Tenth race car*. http://f1tenth.org/build.html (cit. on pp. 9, 12).

[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. 9, 15).

[Haca] Jetson Hacks. *Jetson Nano – Use More Memory!* link. (Cit. on p. 14).

[Hacb] Jetson Hacks. *NVIDIA Jetson Nano J41 Header Pinout*. https://www.jetsonhacks.com/nvidia-jetson-nano-j41-header-pinout/ (cit. on p. 12).

[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. 12).

[Hac19] Jetson Hacks. *Jetson Nano – Use More Power!* https://www.jetsonhacks.com/2019/04/10/jetson-nano-use-more-power/. 2019 (cit. on p. 13).

[Int] Intel. *Intel Dual Band Wireless AC - 8265*. pdf documentation. (Cit. on p. 13).

[Jet] JetsonHacks. *RACECAR/J Initial Assembly*. https://www.jetsonhacks.com/2018/01/28/racecar-j-initial-assembly/ (cit. on pp. 12, 13).

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

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

[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. 13).

[McC18] Gordon McComb. *Robot Builder's Bonanza, 5th Edition*. McGraw-Hill Education, 2018 (cit. on p. 12).

[MIT] MIT. *MIT racecar*. github project & official documentation (cit. on p. 12).

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

[Nvib] Nvidia. *Jetson Nano Developer Kit Guide*. pdf file. (Cit. on p. 13).

[Nvic] Nvidia. *Jetson.GPIO - Linux for Tegra*. https://github.com/NVIDIA/jetson-gpio. github hosted documentation. (cit. on p. 12).

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

[Sca15] Matthew Scarpino. *Motors for Makers: A Guide to Steppers, Servos, and Other Electrical Machines*. Que Pub, 2015 (cit. on p. 12).

[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 p. 15).