MIDDLEWARE

Design et Autonomisation d'un kart

Projet

KART

Gwendal Priser

Paul-Antoine Le Tolguenec

Quentin Brateau

Jules Berhault

Mamadou Dembele

April 12, 2020

Contents

List of Figures

List of Tables

1 Introduction

In the middleware module of UV 4.1, we are asked to implement the ROS middleware to drive a small car built in the previous semester. This small project consists in driving around an athletics track in an autonomous way. To carry out this project we have various sensors such as the Inertielle control unit which estimates the heading of the car and its acceleration, the camera, the gps which will allow us to estimate the position of the small car. We also have an on-board computer, in this case a raspberry.

Repo GitHub

`https://github.com/gwendalp/kart`

2 Functionnal Architecture

2.1 Introduction

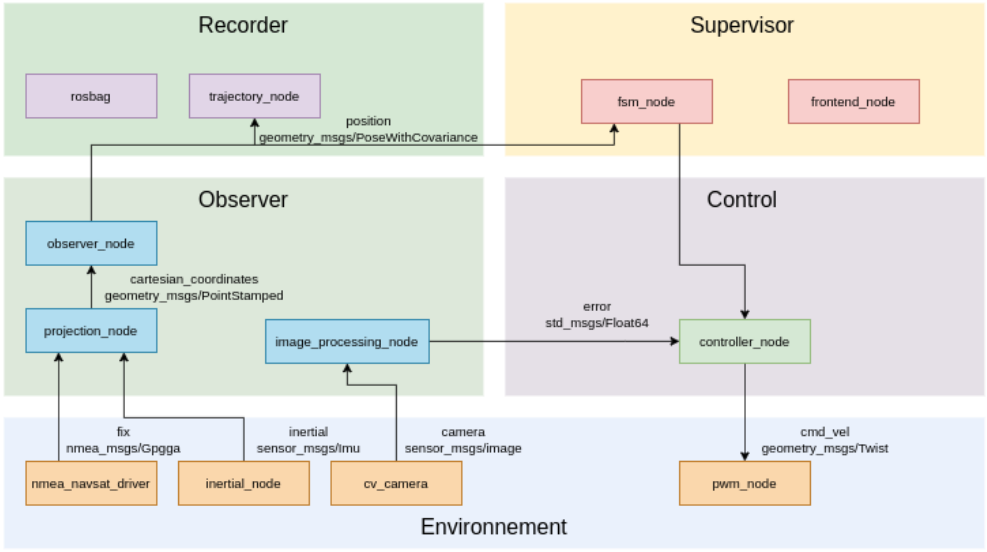

Figure 1: Functionnal Architecture of this project

This constitutes the functional specifications of the project and allows us to have criteria for the success of our project. **Figure ??** shows us the requirements diagram that follows the *SysML* standard for our system.

3 Implementation Architecture

3.1 Introduction

Divide and conquer. How better to describe Information Technology than by this concept? We therefore have to divide the code into different parts in order to better frame the problem and be able to solve it. In order to take control of the car, we decided to implement a control-observer architecture. This architecture divides the project into different parts and thus partitions the code. To implement this architecture, we can use a middleware like the *Robot Operating System* (ROS) which will allow us to simplify the communication between the different sectors through messages.

3.2 Overview

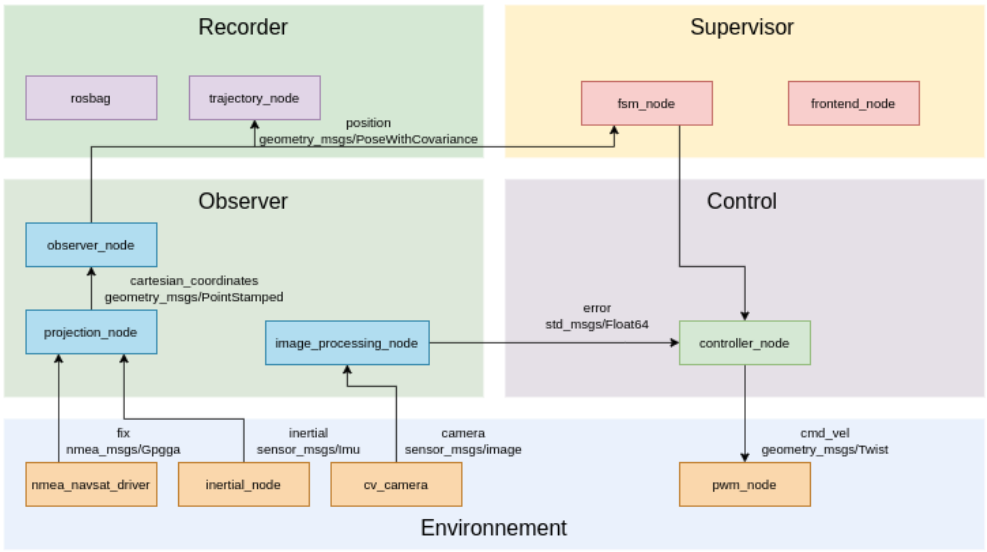

Figure 2: Implementation Architecture of this project

Figure ?? represents the implementation architecture of our project. In classical architectures, there are five main sections: Environment, Observer, Control, Recorder and Supervisor. They are then present in the architecture of our project.

Environment This section gathers all the low level nodes allowing the communication with the hardware. This entire section can be replaced by a simulation node that would publish the same messages in order to evaluate the rest of the functional architecture. In this project we have a global navigation satellite system (GNSS), an inertial unit and a camera. Finally we can control the speed of the rear engine and the position of the servomotor steering the car.

Observer This section is used to process data from the different sensors of the robot. In particular, it will be used here to perform image processing on the data from the camera. It will also allow to merge the data from the inertial unit and the GNSS in a Kalman filter to estimate more precisely the position of the car.

Control This section allow us to establish several control laws for our robot from the sensor data in order to drive our car's actuators. The Supervisor section will indicates the right control law on a case-by-case basis.

Recorder This section records the various robot data in order to be able to replay the data, particularly through rosbags, and it records the trajectory of the robot to feed the finite state machine.

Supervisor This section consists mainly of the finite state machine that controls the mission. It will determine the right command law to control the robot according to what the robot can perceive from its environment. There is also a front-end server that allows us to have a feedback, a graphical user interface to know the state of the robot.

Hardware

3.3 Introduction

3.4 Main Board

For this project, we decided to use a *Raspberry Pi 3B+* as main board. It will let us plug some sensors and control the motors of the car according to the wanted behavior we have programmed.

On this Raspberry Pi, we need to choose an Operating System. our choice was to use Ubuntu Mate because of its simplicity to install and its polyvalence. It will let us do everything we want, like plug sensors, code any program to control our car, ... A preview of Ubuntu Mate is shown on the ??

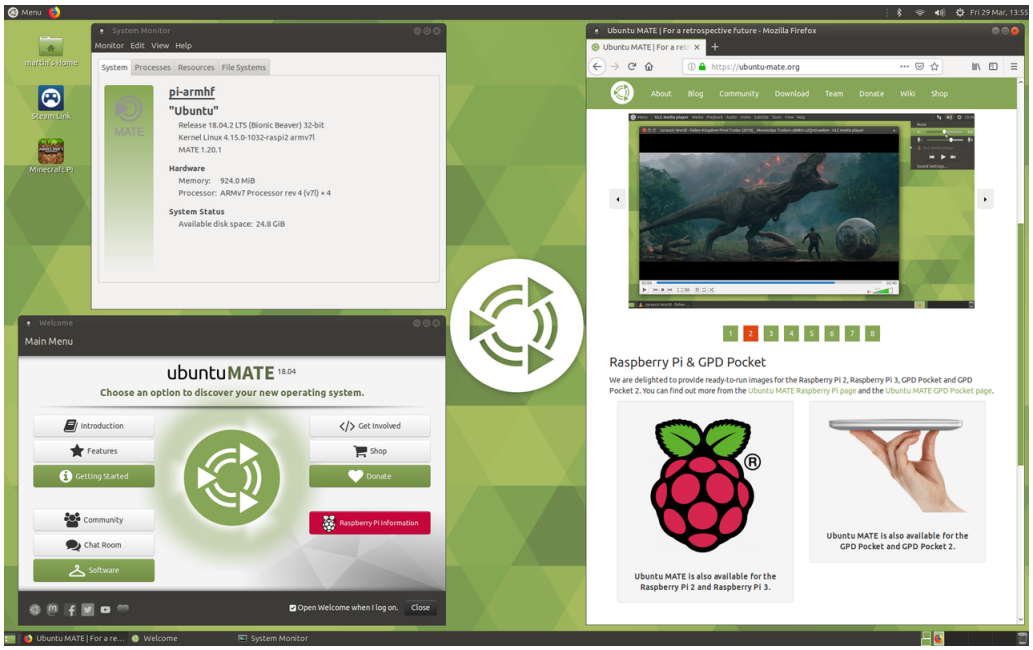

Figure 3: Ubuntu Mate screen

As an imposed figure for our project, we decided to use the *Robot Operating System* (ROS) as Middleware for this car. It's going to offer us some practical tools to code our programs easily. There is also some usefull community shared tools like *rqt* or *key_telop* we will use in this project. Then we decided to setup a graph node and to code these node in order to build our system as shown on the following graph node.

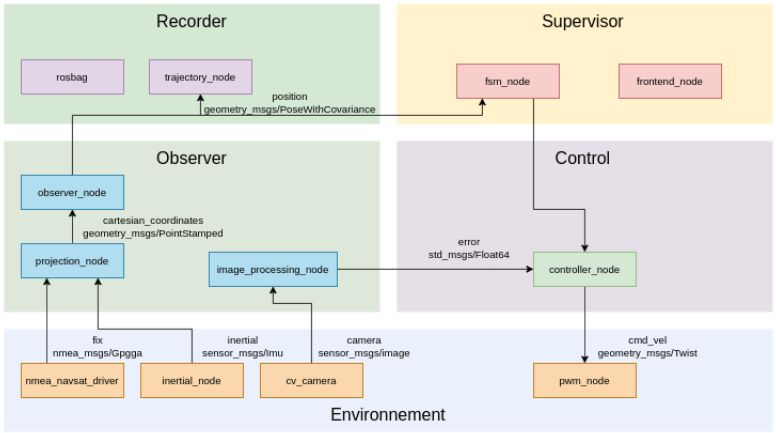

Figure 4: Graph node of the kart

3.5 Sensors

This section present the main hardware configuration of the car. For this project we have to choose which sensors we want in our car in the following list.

As it's shown we decided to choose neither the GNSS nor the Inertial Units, mainly because of their accuracy.

Actually, for our problem we found that an accuracy of 1 meter for the *GNSS* is too large because the car need to run in a 0.8 meter wide racing lane, following a line. This sensors is alos not able to know if the car position is correct.

Sensors	Used	Dope-Level
GNSS	❌	🐢💥👊💩
Inertial Unit	❌	🌈
Pi camera	✔️	🦄

Figure 5: The Sensors list available on our GitHub

For the *Inertial Unit*, we found that this sensor is too noisy to give us any usefull informations about the state of our car. For instance the consecutive integration of the acceleration in order to get the speed and the position of the car leads to an important drift effect on our data. So this sensor is not currently able to to give any correct informations about the state of the car. Moreover, the acceleration of the car could be quite good after filtering if we only needed it. In our case the only usefull information is to know the position of our car in relation to the line.

That's why we decided to focus our attention on the camera. Because we are using a *Raspberry Pi 3B+*, the camera we have choosen is the official camera which can be pluged on the dedicated port on the board. This sensor is perfectly suited to our problem, because with an appropriated image processing we will be able to detect the line and to correct the car trajectory.

3.6 Configuration

Now we will explain how we configured our sensors in our project, to let them comunicate with the software and with the car.

3.6.1 Pi Camera

The configuration of the camera on the raspberry pi is relatively simple. We used the *raspi-config* utility to configure the camera. That's how we set up the camera on the Raspberry Pi.

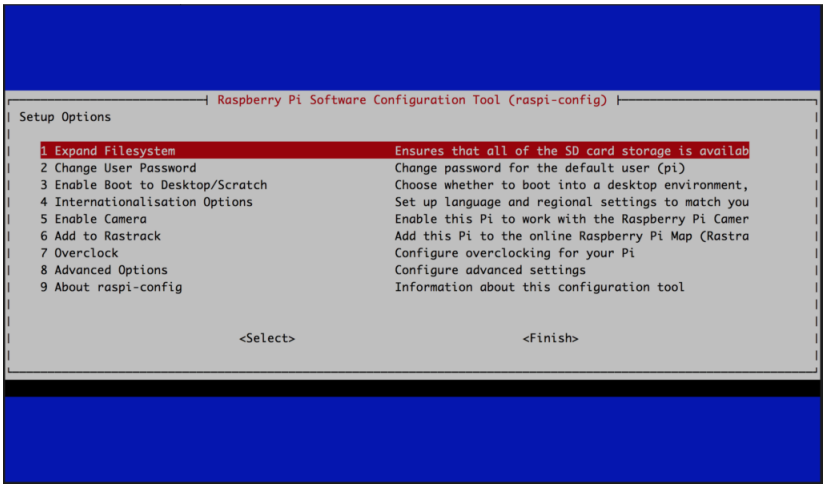

Figure 6: raspi-config utiliy on the Raspberry Pi

3.6.2 Hardware PWM

On Raspberry Pi board there is a lot of way to generate pwm signals. The most of the time, these methods are software based and so they are not accurate. With a lot of searches, we found a website who speak about the raspberry pi's hardware pwm signals. There is apparently an hardware pwm generator used by the bord to generate sounds. It's better to use hardware generated pwm, because if the processor has a slow down and the interrupt is not correctly handled, the pwm duty cycle will not be very accurate and the car will not be able for instance to follow a straight line, because the bearing of the car is controled by a servomotor with a pwm signal.

So we decided to use this tutorial : [**?**], which explain us how to setup pwm signals on the Raspberry Pi, and how to correctly configure the files to have a standard pwm signal which is generated. Then we need to give the rights to users for reading and writing in these files. All these bash command are in *init_pwm.sh*.

Then we have to add some automation. So we created a *crontab* rule. That will automatically create all the required files and allow the permissions to every users. We just have to write in the file *duty_cycle* a value between

1.000.000 and 2.000.000, and the Raspberry Pi will read and adjust pwm signals in real time.

Last but not least, we setup an autologin in order to open a session automatically when the Raspberry Pi boot. That's very usefull in order to launch our programms easily on boot and without any keyboard, mouse or monitor.

4 Mechanical Architecture

4.1 Reasons of an Overhaul

The original architecture of the vehicle was already a good basis for the realisation of a line tracking car. However, manual manoeuvring tests by remote control have revealed some driving faults. Misconduct that could be embarrassing in an autonomous driving mission. Given that our robot will certainly not be as adaptive as a human in its driving, it is interesting to ease the maneuvers and correct some mobility deficiencies.

4.1.1 Problems identified and changes to be expected

Some of the shortcomings noted are listed as follows:

- Loss of control due to high front wheel slippage during high speed turns.
- In case of sharp turn, locking of the inner wheel on the bend caused by the central component box.
- Lack of firmness of the front wheel guiding created by a large backlash.
- Uncontrolled spinning of the rear wheels when acceleration from a standstill or deceleration from high speeds.

And some others component have to be implemented to the initial structure in order to be an optimal support for autonomous driving such as:

- The installation of a camera to see and locate the line to follow.
- Hosting a Raspberry Pi card to manage the control computations.
- Keep space for the rest of the essential components such as the battery, ESC, cables, etc.

4.2 Front Wheel Steering

4.2.1 Front wheel slip

One of the major problems that needs to be corrected, a problem that could impact autonomous driving, is the significant slipping of the front wheels when the steering angle is too large at high speed.

To correct this defect, the best solution would be to improve road handling by increasing the car's grip. To do this, the wheels would have to be changed from smooth tires to studded wheels. However, considering the stock of parts available at ENSTA Bretagne, unless you make an order which would take some time, we did not have better wheels than the ones proposed.

4.2.2 Ackermann steering geometry

Nevertheless, we have come up with another solution inspired by normal cars. The principle of this solution comes from the geometry, especially the steering geometry of Ackermann.

The intention is to avoid the need for tyres to slip sideways when following the path around a curve. The geometrical solution to this is for all wheels to have their axles arranged as radii of circles with a common centre point. As the rear wheels are fixed, this centre point must be on a line extended from the rear axle. Intersecting the axes of the front wheels on this line as well requires that the inside front wheel be turned, when steering, through a greater angle than the outside wheel.

A simple approximation to perfect Ackermann steering geometry may be generated by moving the steering pivot points inward so as to lie on a line drawn between the steering kingpins and the centre of the rear axle.

We have therefore redesigned the pivot points to respect this geometry. We also took advantage of this to improve the bearing embedding system that connects the bearing to the wheel so that the steering wheels are held more firmly.

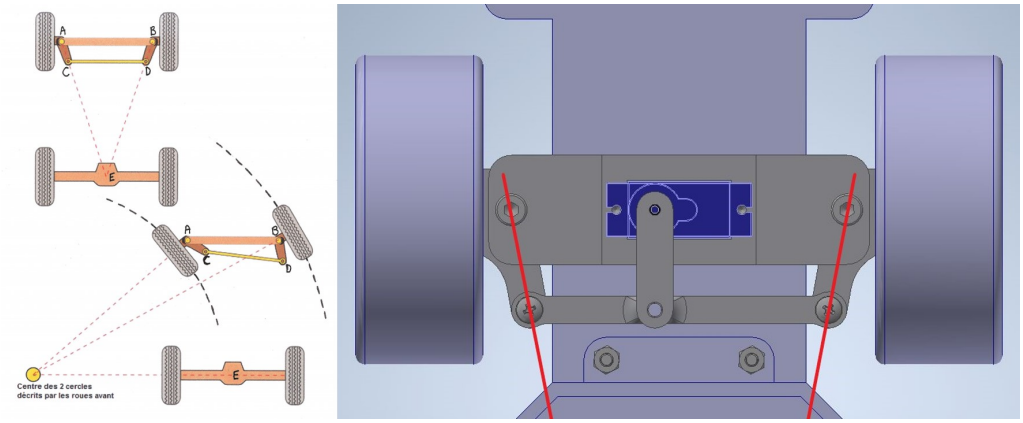

Figure 7: Steering system

4.3 Front Camera Support

4.3.1 Camera support structure

We then had to implement the camera that would allow our robot car to have eyes on the road.

In order to position it correctly, the camera had to be able to stand back enough on the road in front of it and the angle of view had to be horizontal enough to see far away but with a high enough incidence to identify the line without being hindered by the sun's reflections.

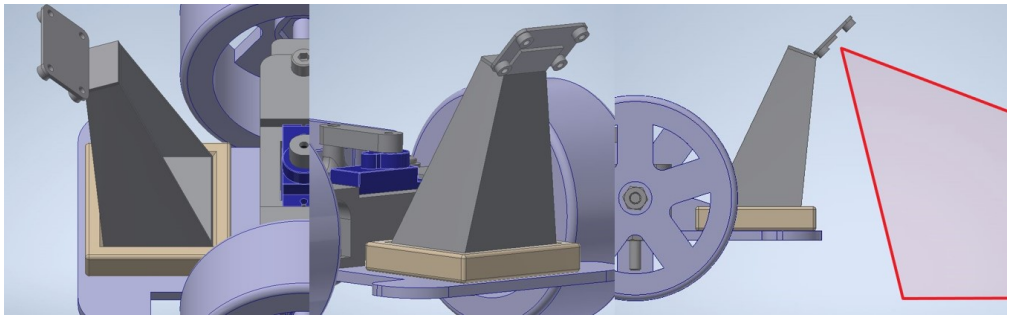

Figure 8: Front camera support structure

We have therefore designed a structure that raises the camera a few cen-

timetres above the chassis and tilts it at 45 degrees towards the ground.

4.3.2 Annoying vibrations

Our car will be driven in a natural environment and the ground will probably not be totally smooth but slightly bumpy and gravelly. The chassis will be subjected to many vibrations and at the same time, the camera to which it is connected will also be subjected to them, which can be quite annoying afterwards in the image processing.

We therefore had to find a solution to try to reduce these disturbances. One of the simple mechanical solutions is to dampen the vibrations in the connection between the chassis and the camera by using for example a silentblock. As a silentblock, a soft synthetic foam seems to be a good choice to counteract these vibrations. (See Figure 5 for more details)

4.4 Component Hosting Box

4.4.1 Additional space

In order to host the necessary additional electronics we needed to increase the space to host the components. We redesigned the box by increasing its width. In its new dimensions, the box offers twice as much space without unbalancing the overall structure.

4.4.2 Carved box

In the process of creating a new box, we took the opportunity to correct one of the problems caused by the front corners that blocked the steering wheels when they were fully steered. So we took the care to add two notches in the place of these corners to allow the wheels to move without hitting them. (See Figure 6 for more details)

4.5 Expectations and Reality

As you may probably know, there is a difference between expectations and the actual reality.

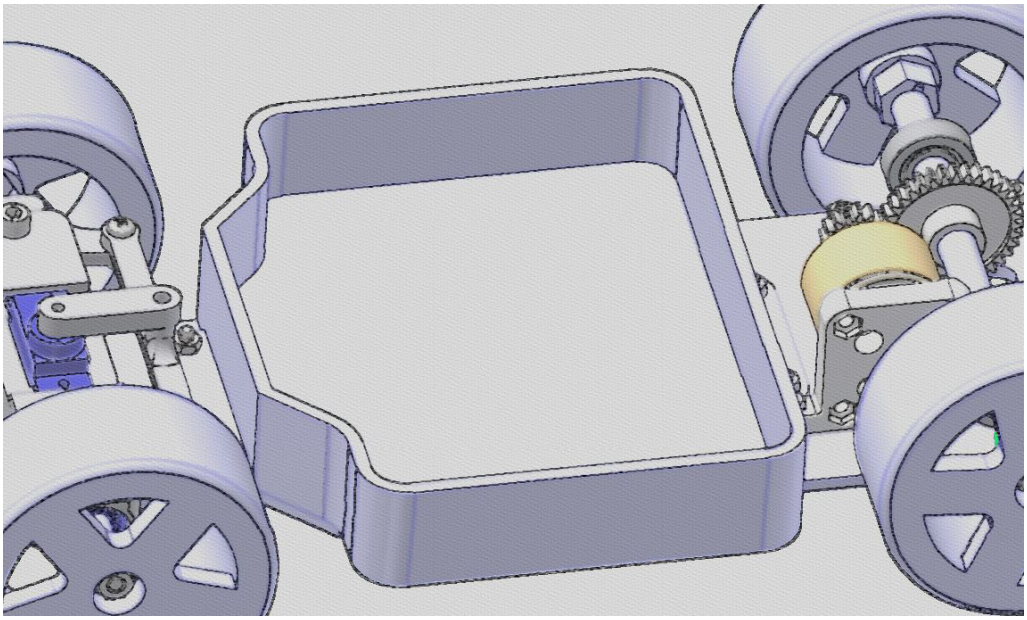

Figure 9: Central component box

The plans were good, but an unprecedented event came to modify them, as a consequence of the proliferation of covid-2019, the school had to close its doors. Deprived of manufacturing means and having only one prototype for the group, we had to stick to what was already done. With DIY, hot glue and a bit of common sense we were able to come up with a modest but acceptable structure.

In the final model, the proposed solutions include Ackermann's steering system and the system for embedding the bearings connected to the steering wheels. The front camera support had to be totally modified but the intentions are the same even though there is no damping system. And the housing box has remained the same but has been doubled on a second floor to contain the electronic components. By limiting the rotational movement of the steering wheels, we can ensure that the front wheels do not get stuck. (See Figure 7 for more details)

Although the final structure of our car does not have all the improvement imagined, it still in acceptable condition for autonomous driving in an environment like a traditional athletics track.

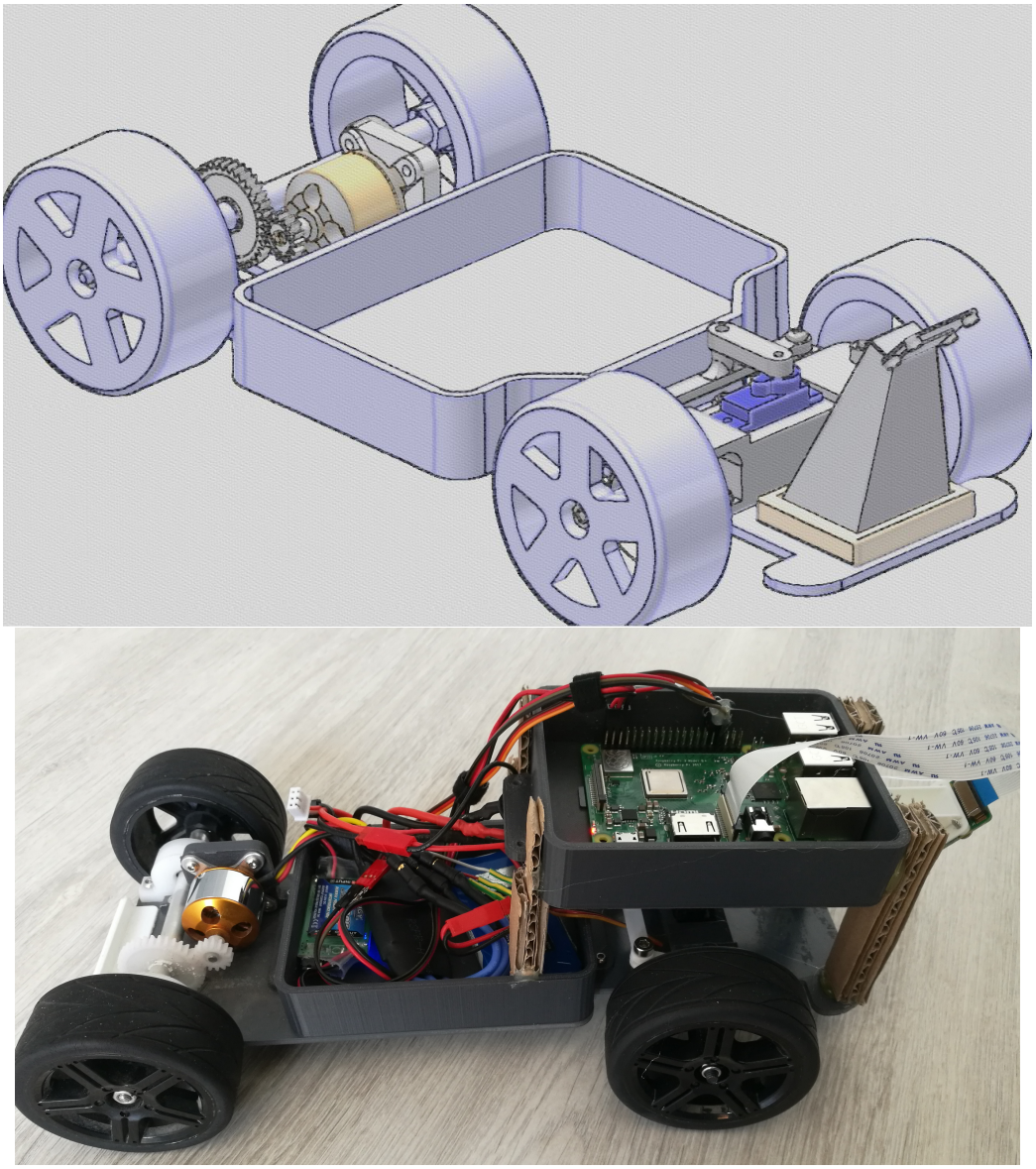

Figure 10: Expextations and reality

5 Image Processing

5.1 Data logging

For the first part of this image processing, we had to collect images. So first we went into the environment in which the robot was going to evolve.

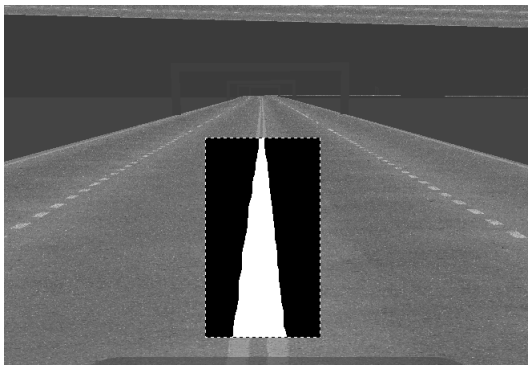

Figure 11: image processed

5.2 pre binarization treatment

In this part, we first had to perform a pre-binarization treatment in order to reduce post-binarization noise. So we used a Gaussian filter to blur the image.

5.3 binarization

Since the line we wanted to mark is white. An effective treatment is simply to switch to grey level. So for binarization, we switch the image to a grey level and threshold for a grey level that we have determined empirically.

5.4 post binarization treatment

In this part we performed a morphological treatment. There was still a lot of noise after binarization. So we made an opening. With a kernel in the shape of a rectangle (since it was the most efficient for this treatment).

At the end of this treatment we obtain a well defined line which crosses the screen.

5.5 find the center of the line

In this last part, the contours are marked using a gradient method. Then the contours are sorted from the smallest to the largest. We recover the largest contour. And we recover the coordinates of the barycentre of the contour. Then the error is the difference between the center of the image and the coordinates of the pixel.

6 VREP Simulation

6.1 Explaination

Based on the practical work made with Benoit Zerr, we got a realisitic and a track with a line in the middle of the track. The simulated in VREP car is equiped with a camera. Our VREP simulator is interfaced with ROS through the piece of LUA code : `rc_car_control_ros.lua` et `rc_car_onboard_cam.lua`.

The VREP camera send a through a topic `image`, a picture of the front of the car. Then the `camera_node.py` subscriber process the image with OpenCV and return the position of the center line in the windows, then it compute an error, we want the line in the middle of the windows. This error is sent through a topic `error`, the controller is suscribed to this topic, it generate a command thanks to a PD command law which is sent to VREP.

And now the car can loop forever, and achieve the perfect run in 1'15" ! 🚙 🏁 🕙

Here is the video link :

`https://www.youtube.com/watch?time_continue=10&v=_vIXo1TvG0w&feature=emb_logo` ⏯

Inspired from Benoit Zerr work : `https://www.ensta-bretagne.fr/zerr/dokuwiki/doku.php?id=vrep:create-rc-car-robot`

7 Localization

7.1 Problem in Localization

In this project we decided to realize the challenge by a visual servoing. That is to say a line tracking by the camera. This type of servo control only works if the robot has at each moment the line to follow in its field of vision. This kind of problem can happen when the robot misses the turn. If the robot ever loses sight of this line, it is obvious that the challenge will not be successful.

To compensate for this unforeseen situation, we have implemented a control law that will be used as soon as the robot loses the line of sight sound. This control law requires knowledge of the robot's state.

7.2 Robot status estimation

To compensate for this unforeseen situation, we have implemented a control law that will be used as soon as the robot loses the line of sight sound. This control law requires knowledge of the robot's state.

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \\ \dot{\delta} \end{pmatrix} = \begin{pmatrix} v\cos\delta\cos\theta \\ v\cos\delta\sin\theta \\ v\sin\delta \\ u_1 \\ u_2 \end{pmatrix}$$

Figure 12: State of yhe robot

Where (x,y) is the position of the robot, θ is its heading, v its speed and δ the angle of the front wheels. To be able to apply the Kalman filter we need a linear state representation of the robot. To make these equations linear, we have considered the angle θ of the robot as an input (since it is given by

the inertial unit). This removes equation 3. Then we linearized the equation into xhat (xhat being the new state to be determined). Xhat consists of x,y, θ, δ. The file available on our github shows the function of this Kalman filter.

8 Discussions et Conclusion

To conclude, we think that we already have a strong basis for this project, particularly with the ROS structure which is already setup. Then we have to perform test with the real system because the simulation is correctly working and a good proof of concept, but we lnow too that simulation and reality are always different. We have to do some improvements on our projetc too, like adding an mission controler in order to control the robot in some cases where we couldn't use the line following command, for instance when the camera is not seeing any lines, and we ha to add a GUI in order to show the kart state like his position, is speed and the circuit.

9 Improvements

9.1 Prerequisites

This part will be implemented in the future and are for now only some good ideas we have for this project. We are first focused on the achievement of the simple control of this car.

9.2 Mission

We have establish a simple command for our kart based on image processing. This command law is quite easy to setup and fully functionnal as we could see. However this command is efficient only if the camera is able to see a line. But actually the car could be unable to see a line, particularly when the car will accelerate in order to run a lap as quick as possible. Our strategy on this topic is to run a first lap very slow and to drop virtual gps tags on a map while the car is following the line. So after a lap we will get a whole map of the circuit and at any given times the car will have to determine if a line is available on the camera, and if the car could follow this line, else the car will have to reach this line again with the help of the gps tags. Then after the first lap, the car will be able to accelerate and for instance some

times lost the line because it will be able to reach again the circuit with the gps tags.

9.3 GUI

We found quit interresting to have a beautifull Graphical User Interface for our car. We are thinking about a frontend solution in order to show dynamical parameters such as acceleration and seep of the car, and a map to show the position with the classical circle to represent the incertainty of this position. Moreover it could be interresting to have the camera output available on this GUI. The easiest way to display a map and some gnss coordinates on python is to use the *folium* package based on the *OpenStreetMap* map. We are now working on a bind between *ROS*, *folium* and the web framework *Django* to sum up these informations on a state webpage. However, we don't own any GNSS or Inertial Units to prepare hardware nodes. That is also not our prioriy, but it can be a very usefull tool.

A Appendix