MIDDLEWARE

Design et Autonomisation d'un kart

Projet

KART

Gwendal Priser

Paul-Antoine Le Tolguenec

Mamadou Dembele

Quentin Brateau

Jules Berhault

April 4, 2020

Contents

List of Figures

List of Tables

1 Introduction

2 Hardware

2.1 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 1

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 pragrams easily. There is also some usefull community shared tools like *rqt* or *key_telop* we will use in this project.

2.2 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 2.

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.

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

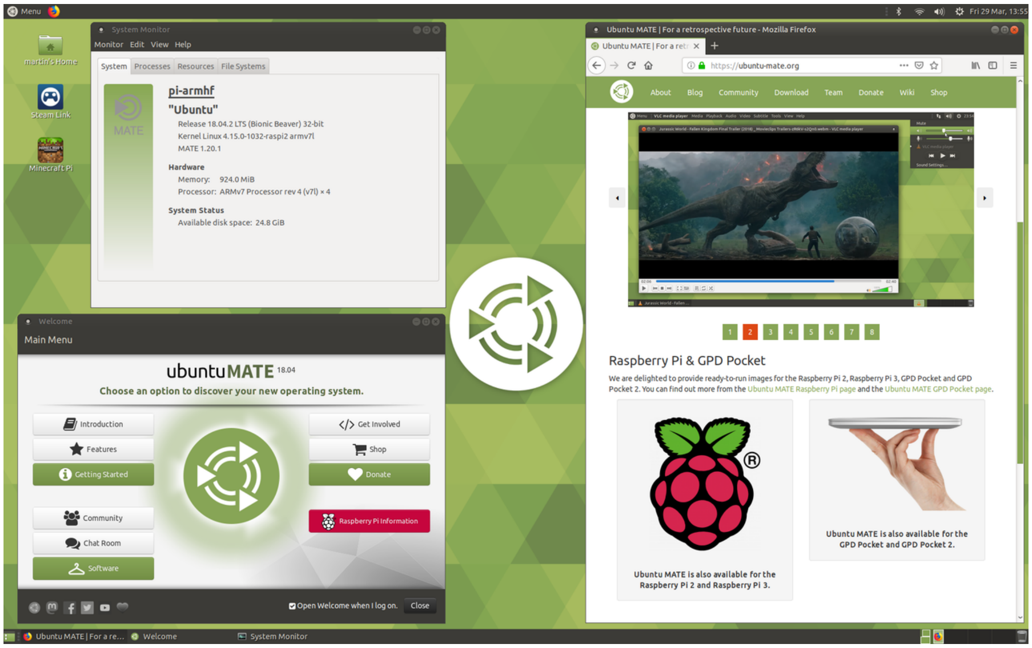

Figure 1: Ubuntu Mate screen

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

Figure 2: The Sensors list available on our GitHub

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.

2.3 Configuration

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

2.3.1 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.

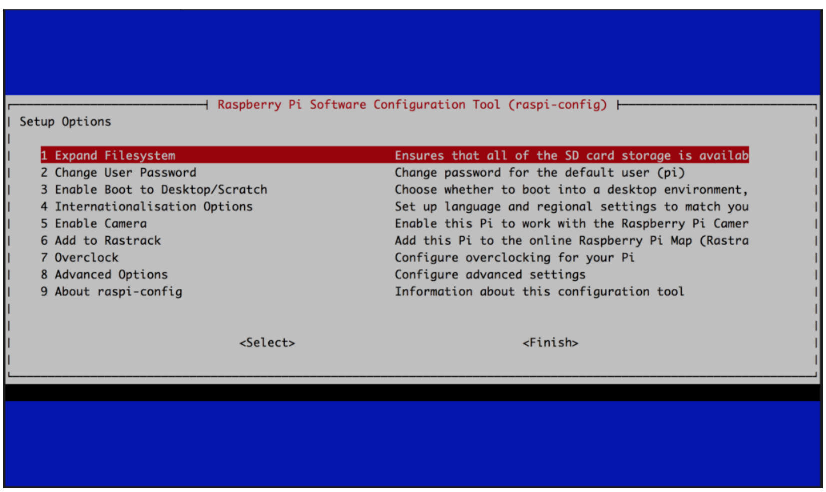

Figure 3: raspi-config utiliy on the Raspberry Pi

2.3.2 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 : `https://disconnected.systems/blog/pi-zero-w-rover-setup/#moving-the-robot`, 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.

3 Mechanical Architecture

3.1 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.

3.1.1 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.

3.2 Front Wheel Steering

bla bla bla bla

3.3 Front Camera Support

bla bla bla bla

3.4 Component Hosting Box

bla bla bla bla

3.5 Expectations and Reality

bla bla

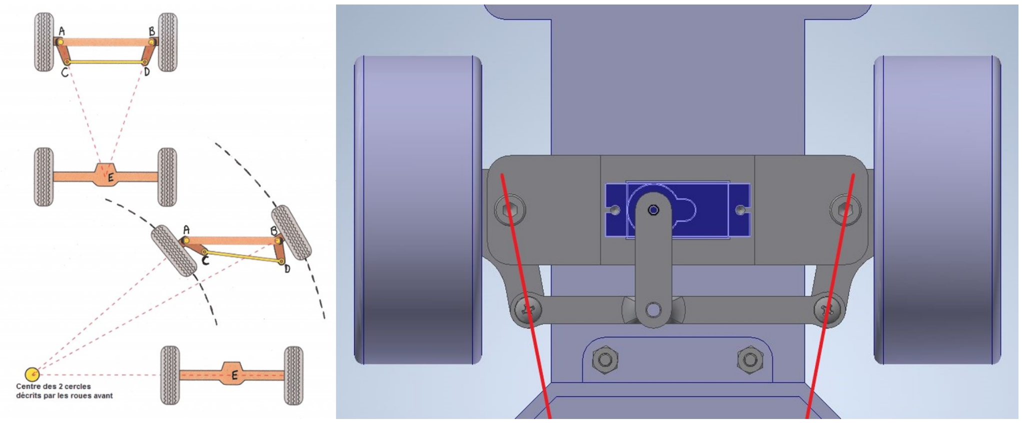

Figure 4: Steering system

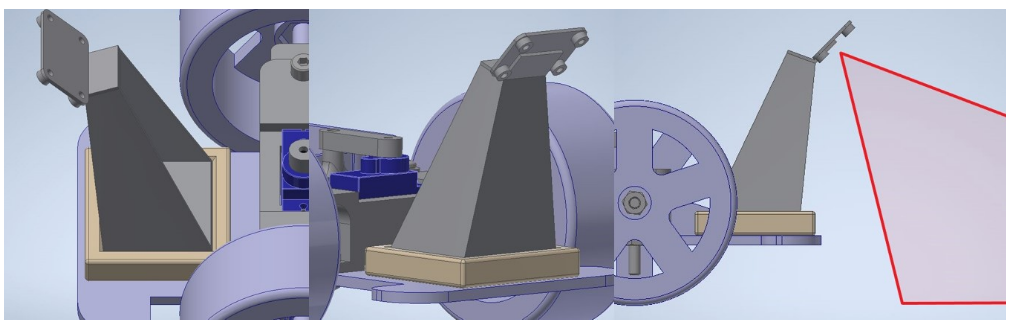

Figure 5: Front camera support

4 Image Processing

4.1 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.

4.2 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.

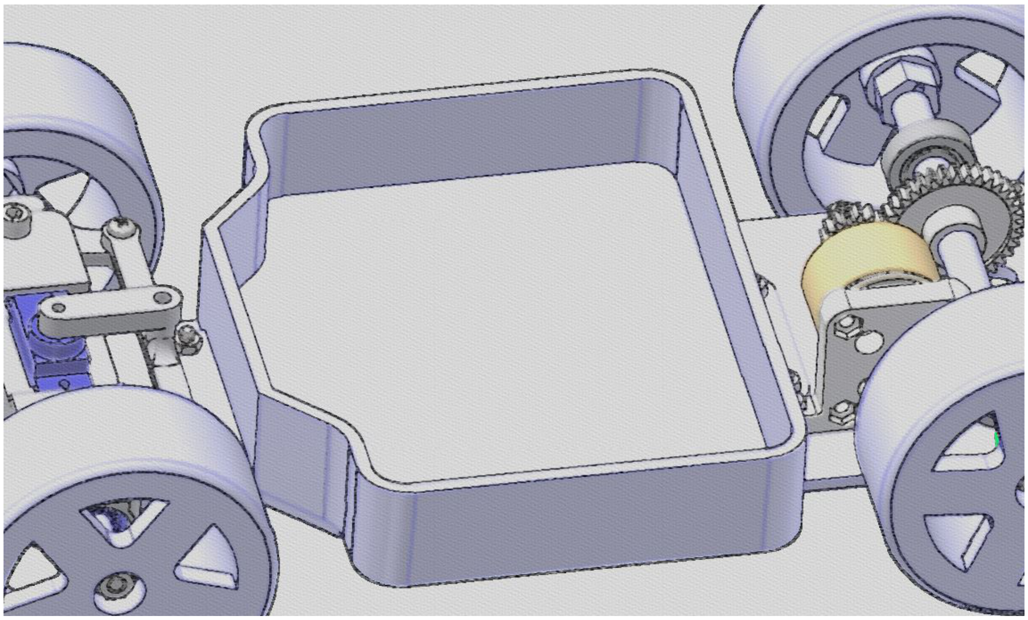

Figure 6: Central component box

4.3 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.

4.4 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.

4.5 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

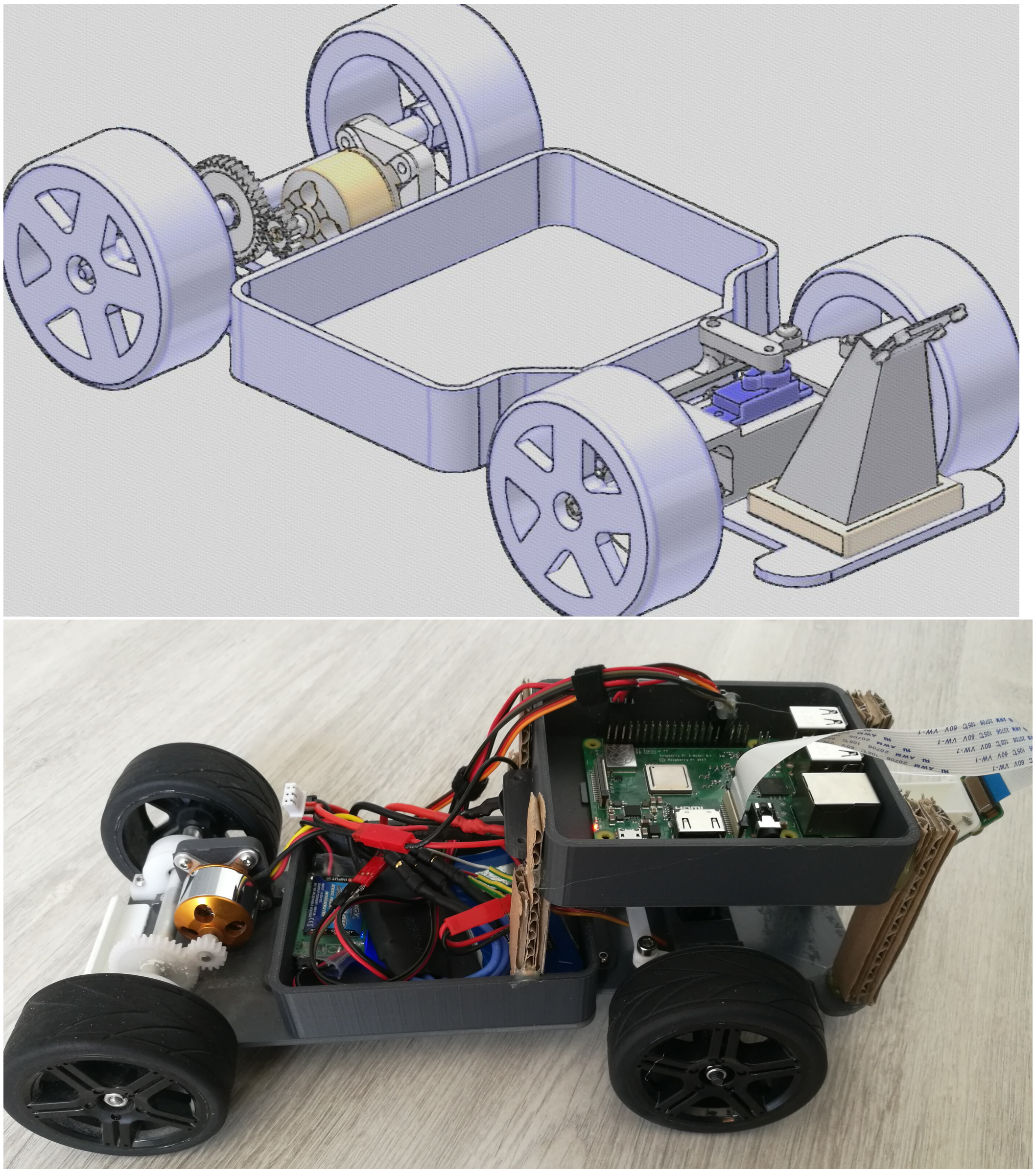

Figure 7: Expextations and reality

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

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

Figure 8: collecting data

and the coordinates of the pixel.

5 Discussions et Conclusion

6 Improvements

6.1 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.

6.2 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.

6.3 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