Nicolas
Lavelaine de Maubeuge
EEL 5666C

Intelligent Machines design laboratory :

Carver², the **Car** **V**ision-**E**nabled **R**obot **R**acer
Final report

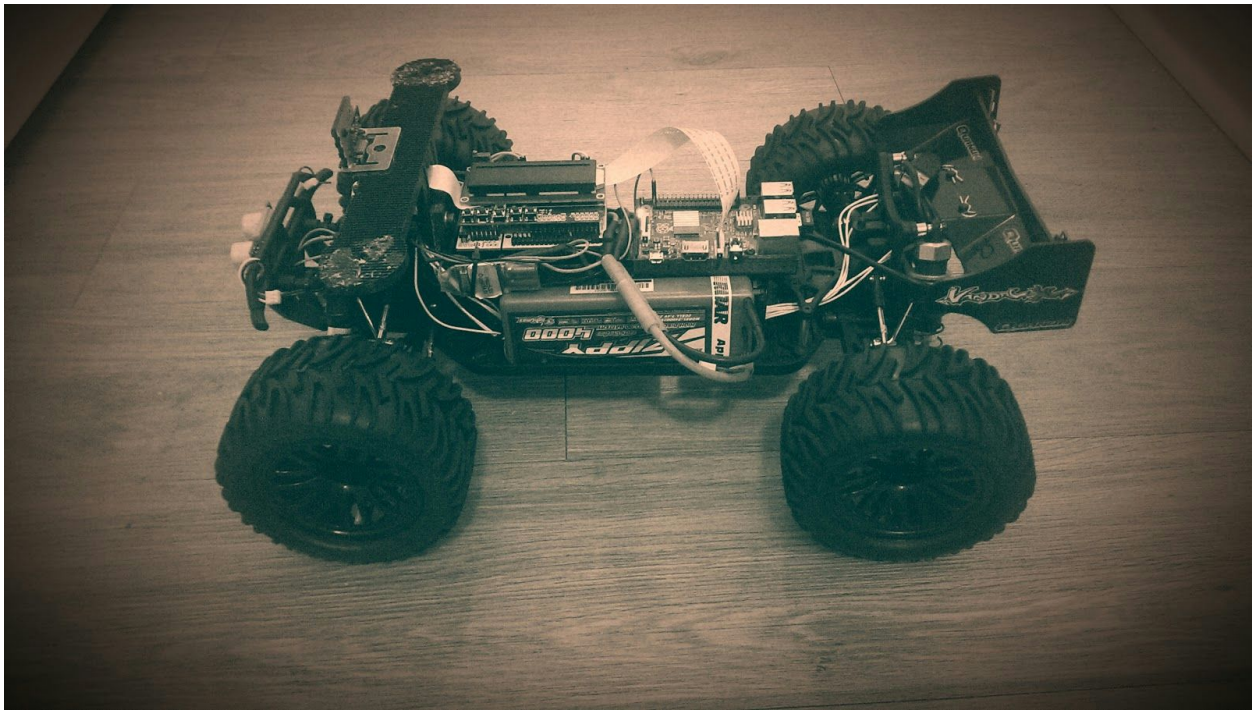


Table of contents :

Introduction

If the line-following robots and the associated competitions are well known among electrical engineers, the increasing computational power of embedded computers now permits a more modern kind of robotic racers : Beacon following robots using a vision system.
Carver², for **Car V**ision-**E**nabled **R**acer **R**obot is a proof of concept for such a discipline.

This type of Robot may have many applications. For instance, let us consider a large property. The owner may want to secure it from possible trespassers by using some kind of automated electronic countermeasure.

If the property is large, using cameras will be definitely very expensive, since one camera has a very limited field of view, the owner would have to use a large number of them.

A UAV could be a good alternative, but those systems are expensive and the battery life will not be more than an hour in the best case scenario. Flying indoor is also very difficult to achieve.

However, a robotic car, would be in this case the best technical solution : Its autonomy can reach several hours and moreover, it is inexpensive.

In order to be compatible with indoor patrolling, a track following system using computer vision and beacons may be used instead of a GPS system. An other system (PIR sensors, movement detection vision algorithms…) may be used to detect individuals inside of the restricted area and to trigger an alarm.

This formal report describes the work undertaken on **Carver** for IMDL during Fall 2015. It presents the robot at some point of the design where it performs reasonably well its main mission.

However, **Carver** still presents a lot of opportunities for improvements, as it will be stressed in this document.

I The mission :

Goal overview :

The principal goal of this robot is to **follow a track marked out by colored beacons** (or cones). This implies the following :

- Carver has to avoid the cones and try not to hit them while following the course

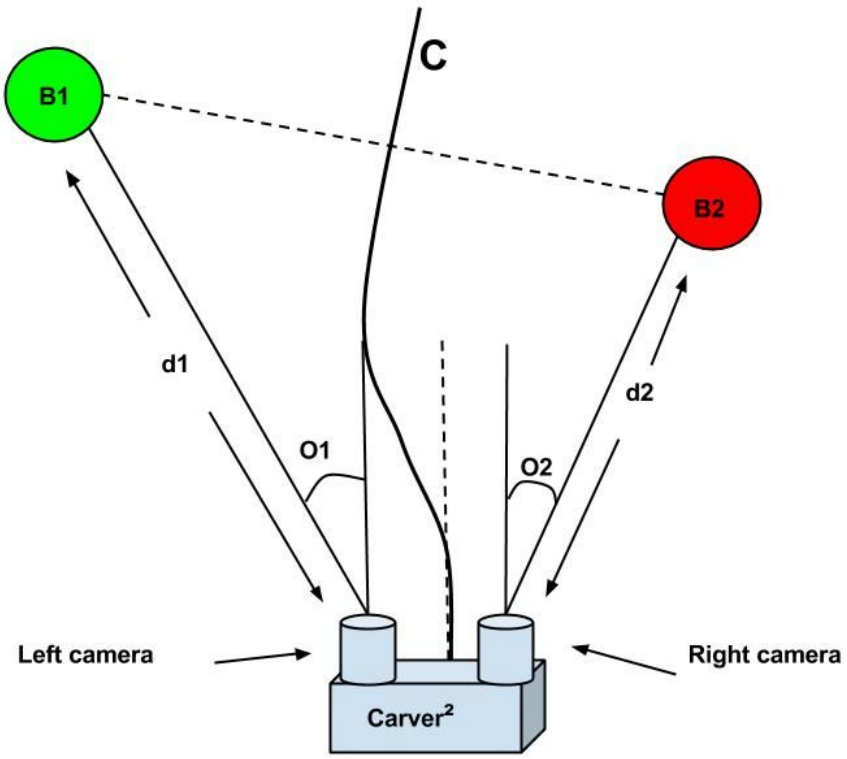
- Since Carver is a race car, it has to go as fast as possible. This means accelerating in straight line and braking for sharp turns.

- If it is possible, Carver will make use **obstacle avoidance** to ensure security. This goal could be hard to reach because the goal may trigger the obstacle avoidance. A workaround could be to make the robot stop as soon as it does not see the beacons. This will prevent the robot from running into a big obstacle like a wall.

Initial approach :

To reach this purpose, my first idea was to use couples of beacons with a predefined color sequence. Using such a sequence would allow the vision system not to mistake the left beacon with the right beacon and would also help in making the distinction between the closest beacon and the other beacons.

More precisely, the robot uses one left camera and one right camera. Each camera only tracks the beacons from one side (ie. the left camera is tracking **B1**).



Basic drawing of the geometry for two beacons

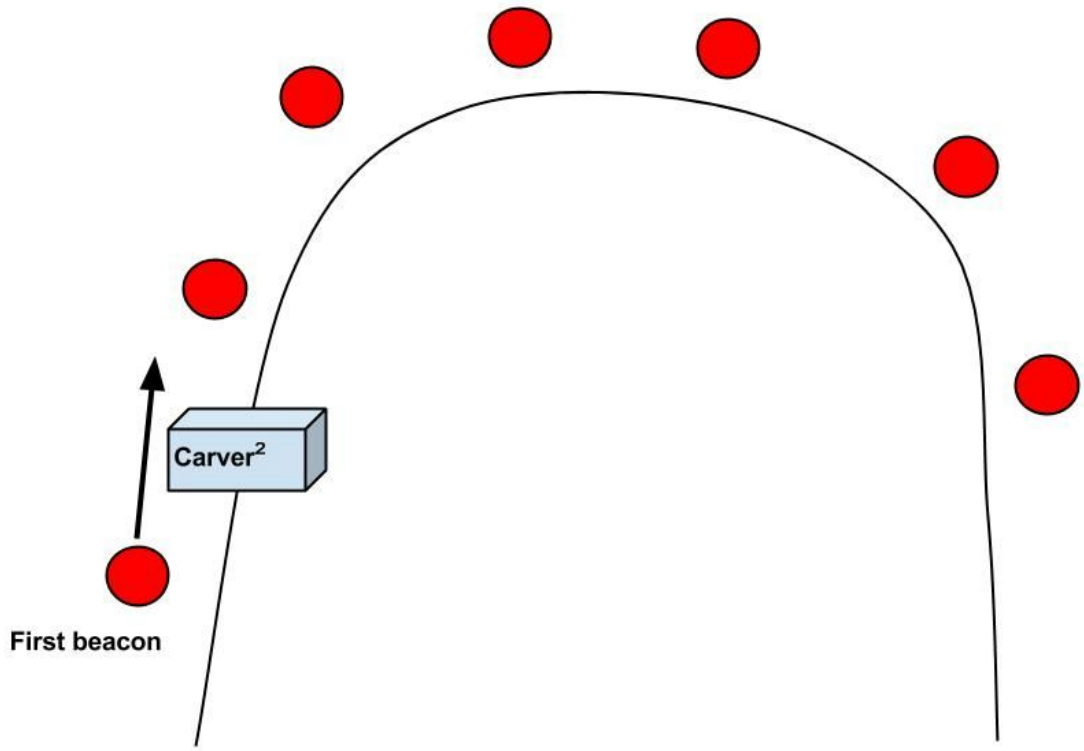
Given the distance d1 and d2 and the angles O1 and O2, I thought that it should be possible to design a control algorithm that would act on the steering wheels to guide the robot through the two beacons.

It turned out during the first experimentations of such a system that this approach did not work very well. This could be explain with mainly two reasons :

- The two cameras on gimbals vision systems did not perform as expected, this will be further explained in the special sensor part
- It is actually very difficult to keep the two beacons in the field of view of the camera at the same time. In fact, most cameras have a field of view close to 60 degrees which made the car loose the beacon as soon as it made a slight turn

Final approach :

Keeping in mind the mission, I decided to rethink my strategy. That is why I designed a simpler solution :



This is different from the previous approach in the following ways :

- Instead a using couples of beacons (cones), Carver will follow a **discrete curve marked out by red beacons**. This is a lot more robust than the previous approach since the probability of seeing one beacon is higher than the probability of seeing to beacons at the same time.

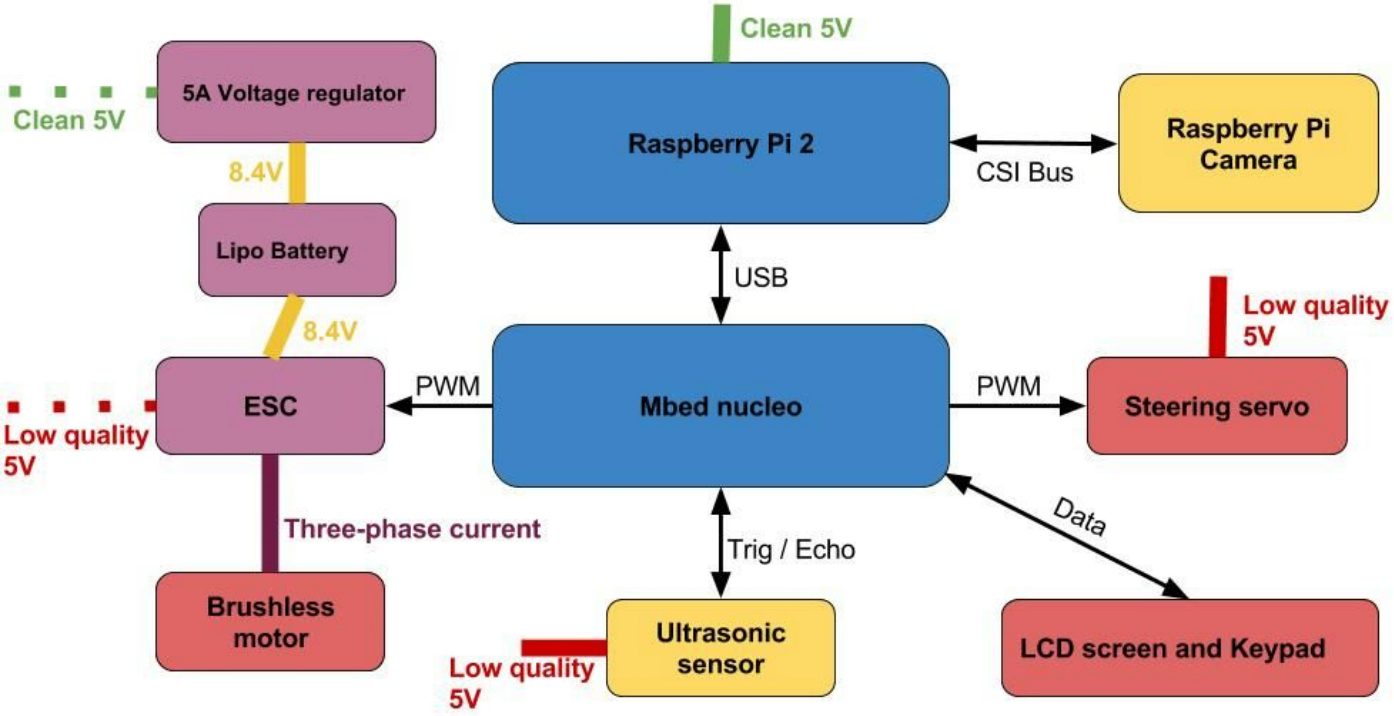
- Carver will pass at the right of the beacons instead of trying to pass at the center of a fictive gate.

- All beacons will be of the same color (RED). This will save a lot of **CPU time** since the vision program will only have to perform one color thresholding at a time instead of three

Experimentations showed that this approach was significantly better than the previous one by giving substantial results.

II System overview :

Hardware architecture :



The **power system** is composed of :

- A 2 cells, 4A Lithium polymer battery that powers a 5A voltage regulator and the Motor's ESC (electronic speed controller)
- A 5A voltage regulator that is a switching power supply. It creates a high quality 5V that will be use to power the "delicate" electronics of the robot
- A 45A ESC that creates a low quality 5V, mostly because it also supplies Carver's motor.

It empowers the following **actuators** :

- A 50A 1000kV brushless motor that allows Carver to move.
- A high speed PWM servo used for steering

The power chain composed of the ESC and the brushless motor is able to respond to both positive and negative setpoints, giving the ability to go **forward**, **reverse** and **brake** depending of the command.

The **computing** is based on :

- A Raspberry Pi 2 that runs the high level computations and is connected to its dedicated camera through a CSI bus.
- An Mbed Nucleo F411re, that takes advantage of its 100 MHz frequency to poll Carver's sensors and synthesize the PWMs for the actuators command.

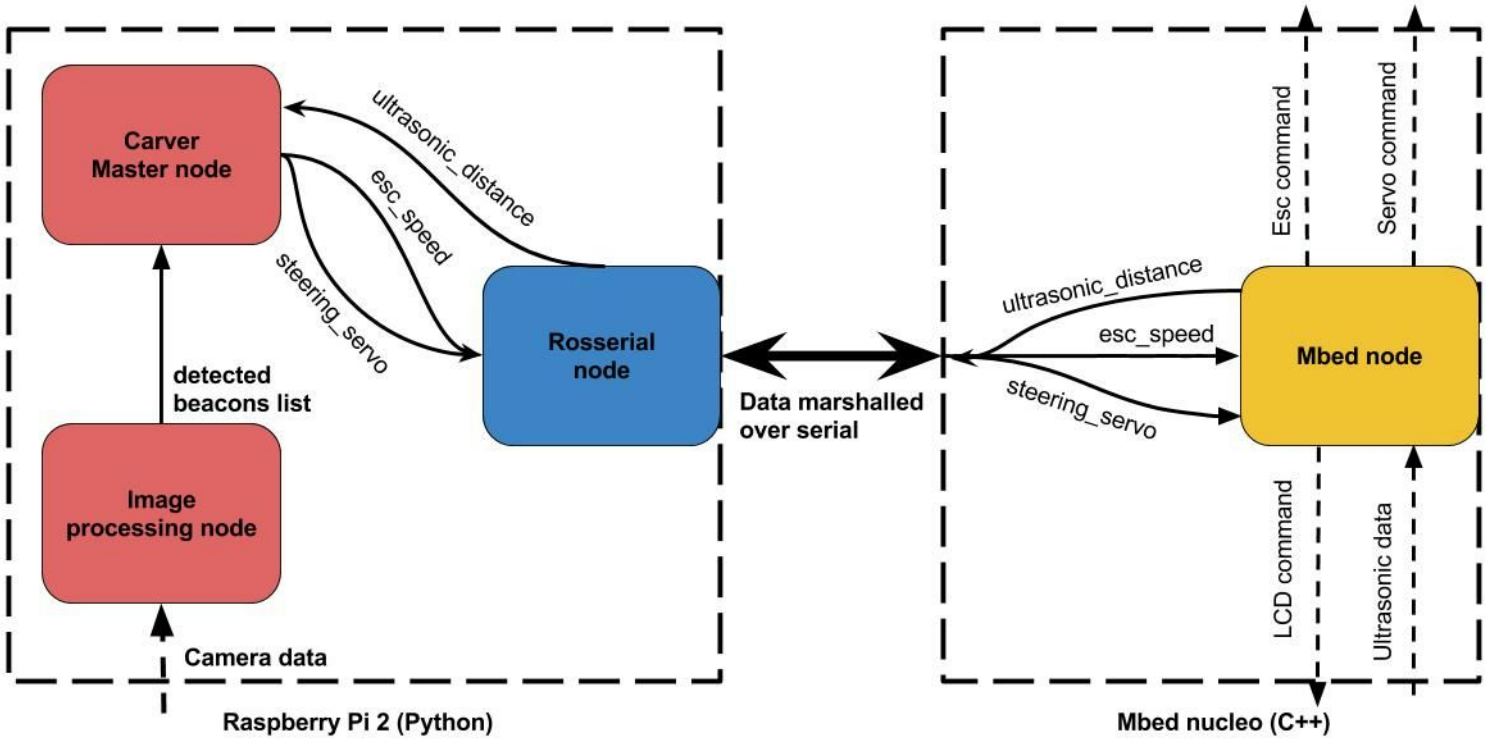
Software architecture :

Carver's software follows a **ROS Indigo** (robotic operating system) distributed architecture. It is composed of a set of nodes that run across the the two computation boards and are interconnected together using **ROS topics.**

Each node has the ability to publish / subscribe to a topic. This is allowed via the **rospy** library on Python (for the code running on the Raspberry Pi 2) , and via the **Rosserial** library on the C++ code running on the Mbed microcontroller

Here is a list of all the topics used on Carver :

- esc_speed
- steering_servo
- ultrasonic_dist
- beacons_list



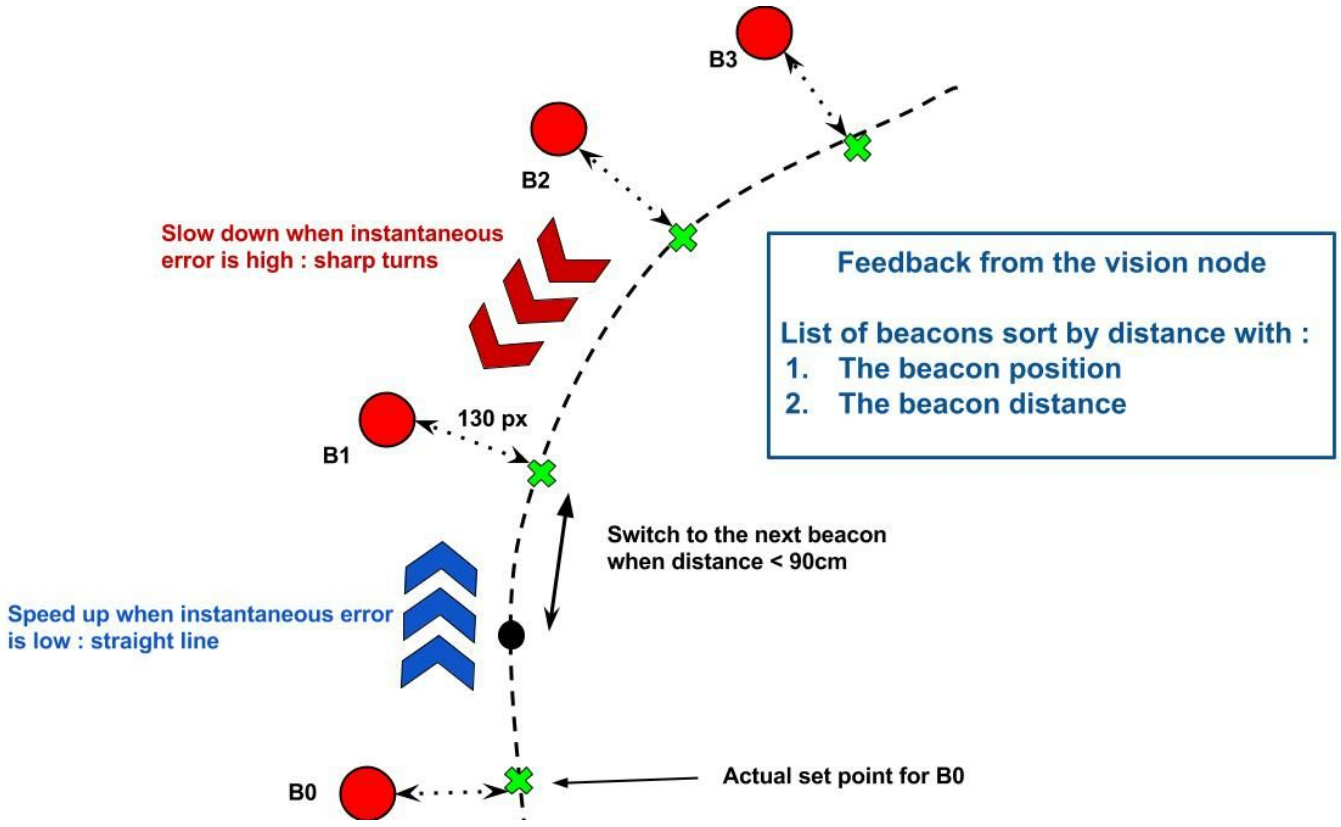
The raspberry Pi hosts the three high level processing nodes :

- **The master node** that regroups the controlling and path-finding computations (the next page will give a more precise description of the behaviour of this node)

- **The Image Processing nodes** that performs the vision processing and returns a list of the detected beacons (it will be detailed in the special sensor section)

- The **Rosserial node** that builds a bridge between the Raspberry Pi and the Mbed through serial. The Rosserial library running on the Mbed that I specially adapted for the ROS Indigo version allows this device to natively subscribe and publish to ROS topics shared on the entire system. This is a huge **improvement** compared to the homemade sender / parser design, as Rosserial is communication fault tolerant and has been deeply debugged and tested to respond to industrial standards.

Here is a brief schematic of the behaviours exposed by the master node :



Using the detected beacons list it gets from the image processing node, the first role of the master node is to define a setpoint and to make sure that Carver will reach it.

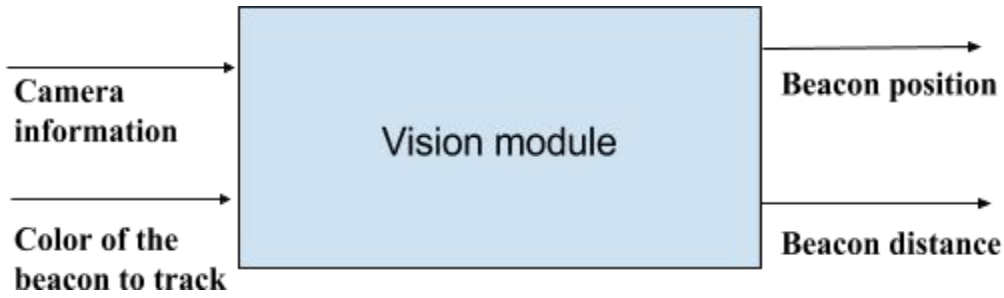
When Carver is very far from the beacon, the master nodes aims to a **120 horizontal pixels difference** from the beacon's position. Determined experimentally, this constant allows Carver **not the hit** the beacon as it gets closer from it.

When the distance to the beacon becomes inferior to 90cm, if the next beacon is detected, the master node **switches the setpoint to the next beacon**.
This “advance on the track” is required in order not to be surprised by very sharp turns. In this case, and without this feature, Carver would react too late to a far horizontally-located beacon.

An additional feature is to **speed up** when the error between the horizontal position of the beacon and Carver’s position is low, while **slowing down** when it is high. This behaviour was implemented in order to respond to the speed specification.

III Special sensor :

To begin with, let us define the interfaces of the vision module :



Initial attempt

My initial plan was to use simultaneously two instances of this module, one on the left camera and one the right camera.

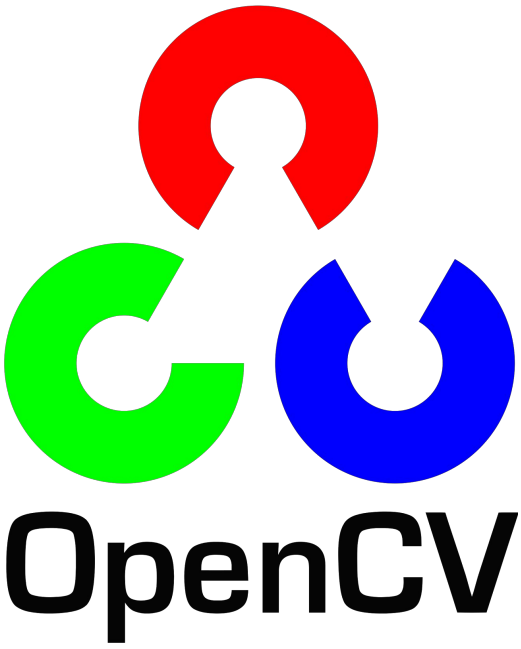
Each vision module used both one PS3 eye camera and the four cores of the Raspberry Pi 2.

The Raspberry Pi 2 is equipped with a quad core 700Mhz ARM central processing unit. To cope with the matter of speed detailed in the previous part, the CPU of the Raspberry Pi 2 has been overclocked to 1100Mhz per core. Heatsinks were added to the chip to dissipate the additional heat.

The proper stability tests were run in order not to introduce errors in further work.

The two PS3 eyes support a resolution of 640x480 at 60 frames per second, which is more than enough for our application.

The approach used was to first design the object tracking algorithm itself. To simplify the problem, speed and quality matters were not really taken into account as a first step.



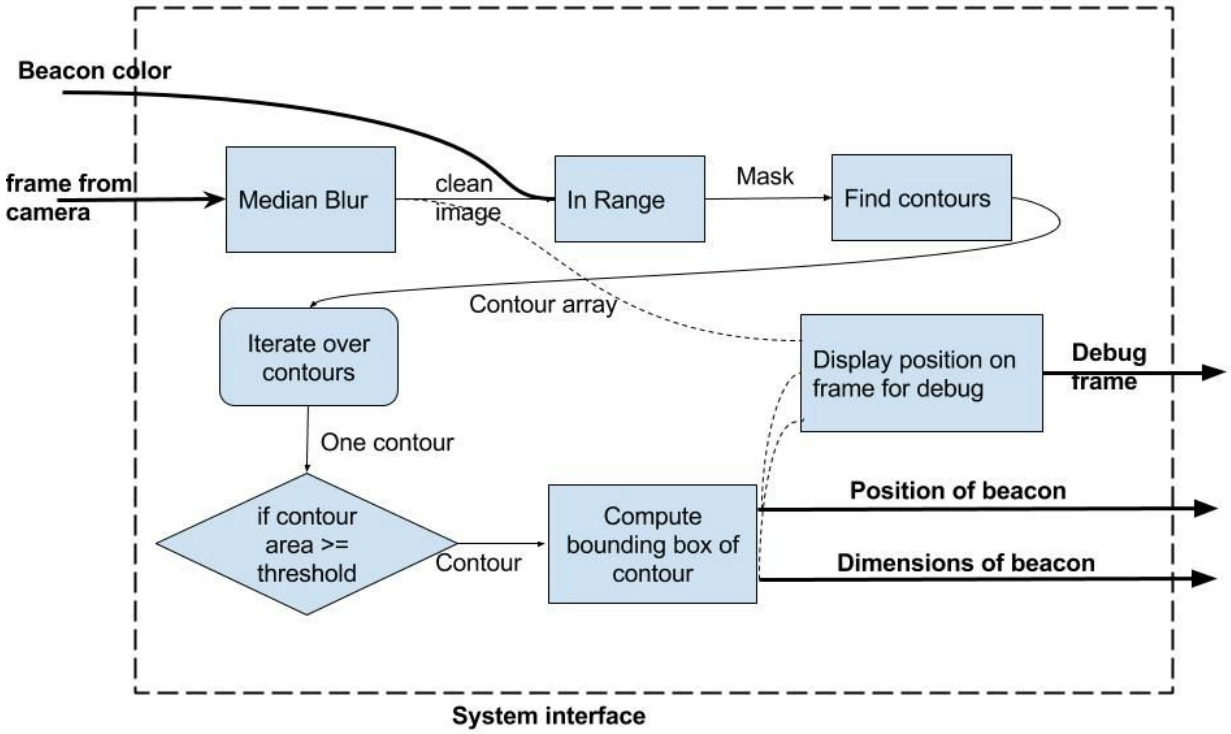
In fact, the speed requirement was relaxed by designing and testing the code on a intel i7 computer. This platform is significantly faster than the target device.

The need of precision was first ignored by tolerating a small amount of false positives.

Even if because of these decisions, the following results are not directly applicable to **Carver²**. the next part will present several optimisations that makes the work undertaken here relevant.

The vision library chosen is OpenCV. In accordance with the project's timeline, the Python language was chosen to shorten the time spent on programming. There is a lot of support for the Python bindings of OpenCV. Several parts of this work are adapted from examples from the OpenCV Python documentation

Here is a diagram explaining the inner workings of the vision algorithm :



The Median filtering removes parasites from the incoming image.

Then, the In-range function removes everything in the frame that is not the same color as the beacon.
Next, we find all contours and iterates among them.

As soon as a contour has a size that is superior to a certain threshold, we consider that this contour is the one of the beacon and we return its size and position.
An optional debugging output returns the frame with the beacon position.

The camera is then centered on the closest beacon using PID controls on the x and y axes.
The coefficients of the PID controls were adjusted experimentally.

Even after having spent a substantial amount of time on the gimbals control part, the tracking speed was still not high enough to follow a beacon at the car's moving speed.

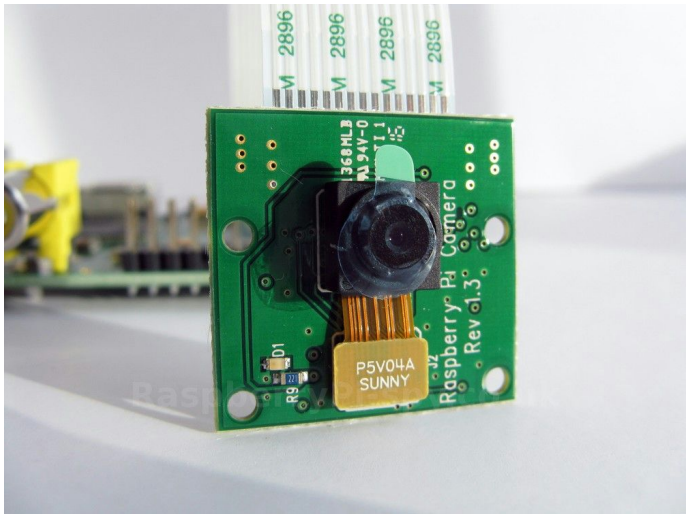
The high delay (about 400ms) in grabbing the frame from the PS3 eye was diagnosed to be the reason of this failure.

Final solution

To cope with that delay issue, I decided to replace the two PS3 eye by a single Raspberry Pi camera.

This camera is directly interfaced with the ARM central processing unit of the Raspberry Pi using a CSI device that is supposed to give the very best performance in terms of delay and throughput.

The previous software was reused with a single variant : instead of tracking a single beacon, it returned a list of the detected beacons, sorted by distance.



My first attempt was to try to grab the frames from the Raspberry Pi using the popular **Raspicam** python library. Despite my initial enthusiasm, this library only gave me very poor results, with only 17 frames per seconds and consuming 30% of the Cpu just for grabbing the frames.

Using a simple modprobe command[1], I have found a way to force the Raspberry camera to use the standard **V4L2 driver**. This setting gave the optimum results for my robot : a steady 30 frames per second with almost no delay using only 10% of the Cpu for grabbing the frames.

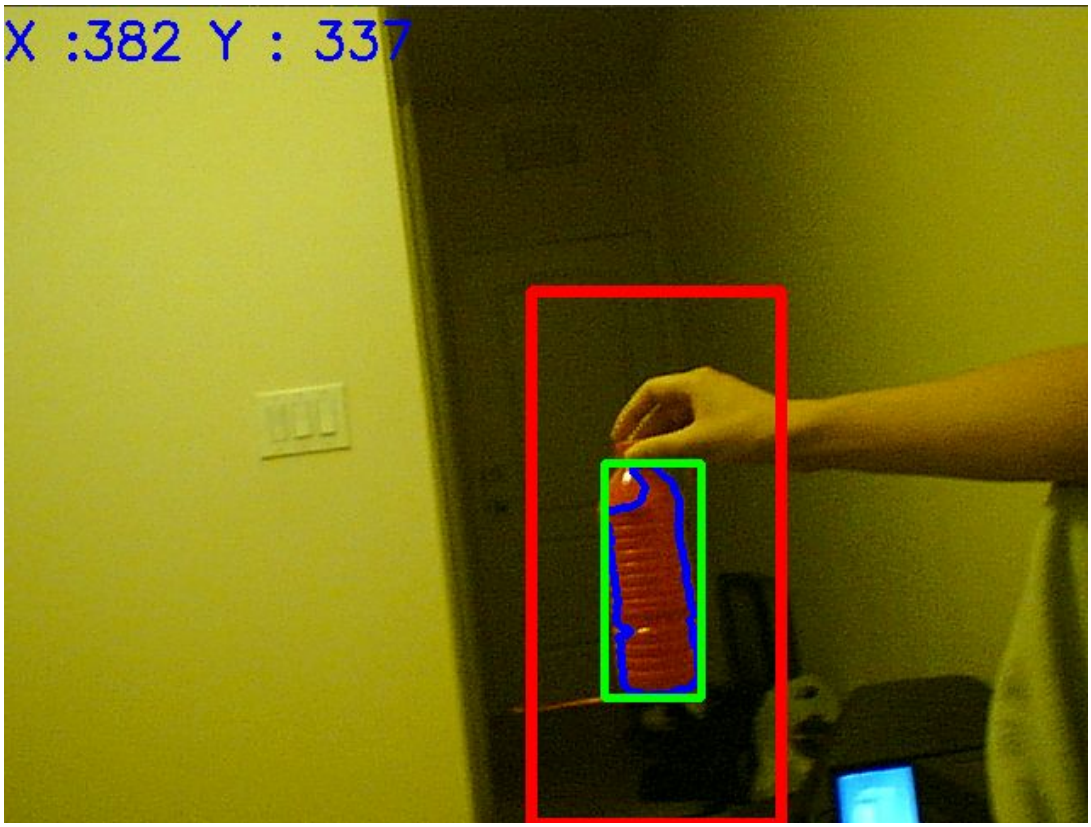
This set up also worked perfectly with my previous program because the Raspberry camera is then considered as a **standard linux webcam**.

[1] sudo modprobe bcm2835-v4l2

Software details

To respond to the matter of speed, an interesting optimization has been designed. When the module is first launched, we search the beacon in the whole frame. If the beacon is found, we save the beacon position and size into memory.

At the next iteration, we look if a previous object has been found. If yes, we retrieve the necessary data. We now define a window where the object should be, given its previous position and size. We run this this time the vision algorithm on the reduced window, which is usually three to twenty times smaller than the full frame. If the object has not been found, we consider we have missed it and rerun the algorithm on the whole frame.

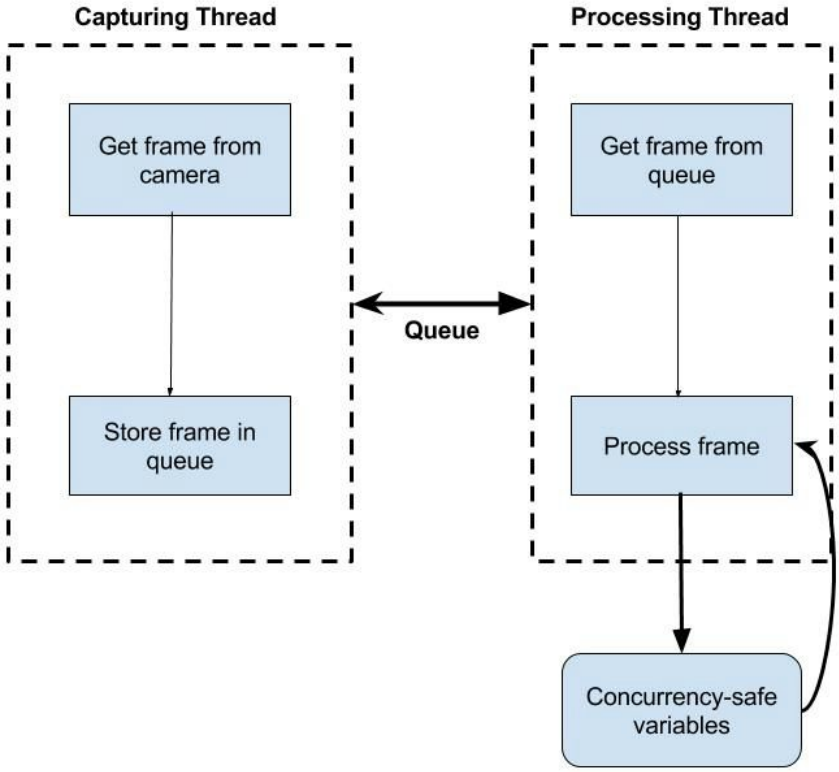


On this picture, the red rectangle displays the region of interest where the object is really searched in. The green rectangle displays the location of the object.

Interestingly, there is a tradeoff between the searching window size and the framerate in terms of CPU time : The faster the framerate, the smaller the window can be. The smaller the frame rate, the bigger the window has to be in order not to miss the object.

The experiments has shown that a window size of 3.5 times the size of the object and a frame rate of fifteen frame per second is enough. This design saves on average 50% off the CPU time.

Unfortunately, despite my best efforts, OpenCV does not use natively the four cores of the Raspberry Pi. The previous script keeps using 100% of one core, which does not satisfy our need of a fast frame rate. To cope with this matter, the code has been parallelized the following way, using the Python Multiprocessing library :



In this design, a capturing thread is sampling the frames from the camera. The frames are inserted in a FIFO Queue that is natively present in multiprocessing.

A pool of threads is processing the frames they get from the Queue. The resulting position and size data of the beacons are stored in concurrency safe variables. A pool of four threads is enough to use the four cores simultaneously.

With this design, each camera module uses 40% of the CPU at fifteen frames per second. It is thus possible to use the two PS3 eyes at the same time and still have so CPU time left for the upcoming path planning algorithm.

In order to improve precision, what I have first done is to use some information that characterize the beacons in order to remove false positives.
In fact, given the fact that the beacons are plastic bottles of a fixed width and height, it is possible to compute the ratio of height/width in pixels. Such a ratio does not change whatever the distance of the beacon from the camera.

Using this information, it is possible to remove all the false positives that were still present at previous steps.

Another improvement, is to use some sort of calibration to determine the colors thresholds that have to be used.

Here is an attempt for an automatic calibration :

The beacons are placed next to a white surface that acts as a background (ex : a wall). An algorithm detects the beacon and computes the average color within its contour during a certain amount of time.

By doing this, we should get a more precise value of the colour to pass to the vision algorithm. This process is only a matter of seconds, so it could be done before each session to adapt to the scene's ambient light.

Surprisingly, this process gave me very bad results. In fact, a manual calibration was proved to be far more effective during experimentations. This fact can be explained using the following reasons :

- Beacons colors changes slightly according to their position in the scene due to non constant indoor illumination
- HSV color thresholding requires to adjust 6 parameters at the same time. Their optimum distance from the average object color is far from being equal. A smarter algorithm (like gradient descent) to minimize both the false positives and the undetected parts of the beacon is probably a more suited solution

Considering this, and to speed up the calibration process, I have designed a simple GUI to manually set the color thresholding parameters before each run

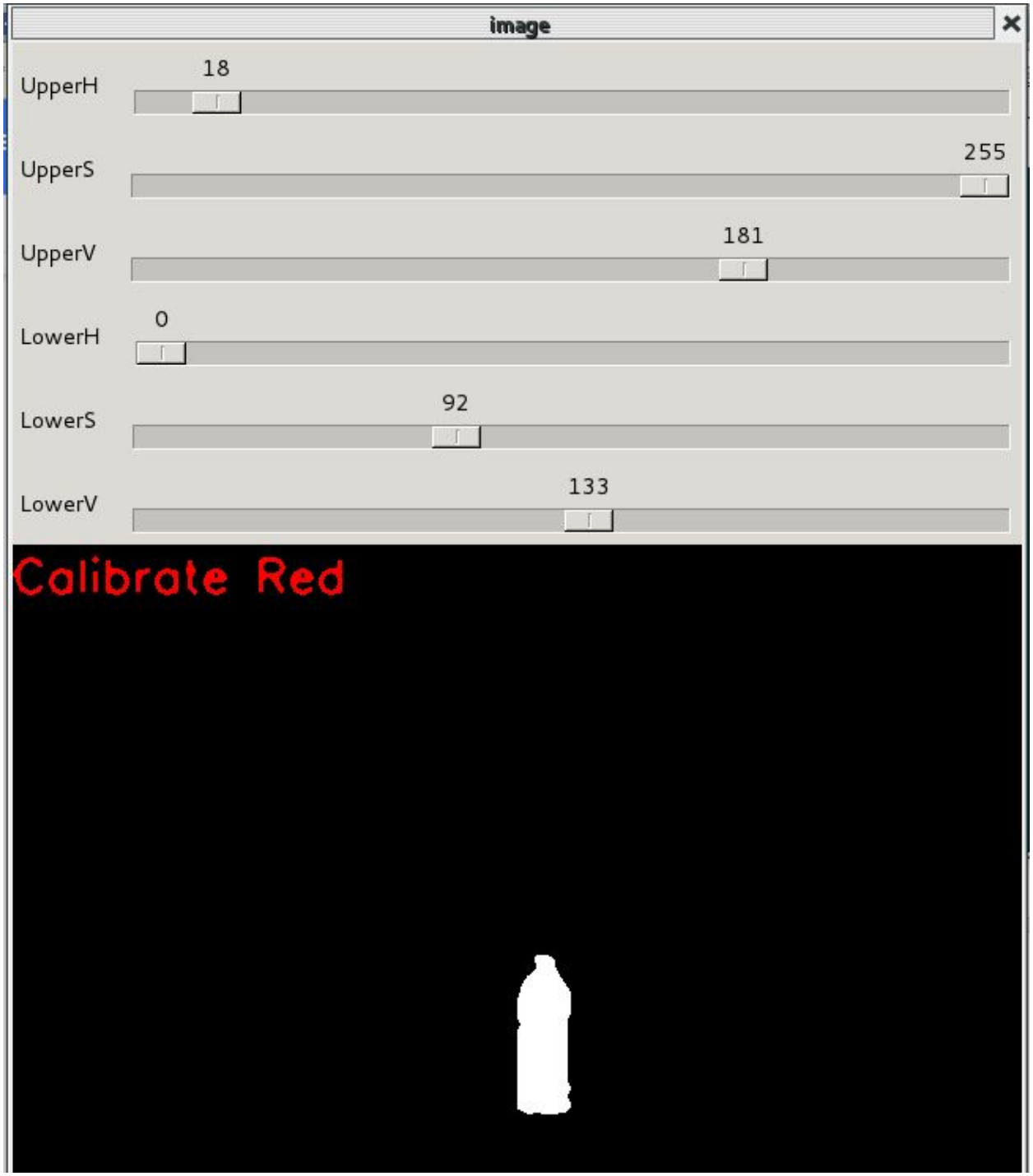


image
UpperH
18
UpperS
255
UpperV
181
LowerH
0
LowerS
92
LowerV
133
Calibrate Red

IV Mission results

Performance assessment

Here is a quick descriptive of Carver capabilities relatively to the specifications at the end of the project.

Track following :
Carver is able to follow a 30-beacon track without hitting any beacon, 90% of the time. It successfully slows down for sharp turns and speeds up in straight lines.

Dependance to calibration :
Robustness is still deeply correlated to the precision of the calibration and the settings. Color thresholds, as well as exposure needs to be carefully adjusted before each run.

Additionally, it is actually impossible to define a universal set of control settings that will be suitable for each track. In fact, the PID constants and the speed coefficients has to be adjusted relatively to the difficulty of the track.

Obstacle avoidance :
Direct obstacle avoidance using the ultrasonic sensor while following the track is not possible to achieve because it is triggered by the beacons.
However, to demonstrate the obstacle avoidance capabilities of the robot, a high speed random wandering mode with obstacle avoidance has been implemented. Back on the track, we can consider that a very big obstacle would probably prevent the robot from seeing the beacon and thus stop it.

The left-turn problem :
It is really easy for Carver to make a right turn because it sees all the upcoming beacons. However, since Carver passes to the right of the beacons, left turns are far more difficult because upcoming beacons are further on the left or hidden by previous beacons. Consequently, the admissible left turn angle is way more superior than the one of a right turn.

Possible improvements

Use a wide-angle camera : A camera equipped with a fisheye lens may considerably reduce the importance of the “left turn problem” as well as increasing the max right turn angle. I did not implement this solution since I was unsure of the time that I would have to spend on distortion removal.

Use a more powerful onboard computer : An Odroid-like device may unlock new possibilities for the vision software, like using a feature recognition algorithm to compute the position of the beacon instead of a very simple color thresholding.

Add a pan system for the camera : This could allow the software to orient the camera in the direction of the turns and therefore completely unlock field of view restrictions, as well as increasing robustness

Conclusion

This final report, with the upcoming demo day closes my work on Carver within the **I**ntelligent **D**esign **M**achine **L**aboratory.

Vision programming using OpenCV, design of a complete architecture with the Robotic operating system, controlling and path planning, test designing, it would be too long to list here all the technical skills learnt while following this class.

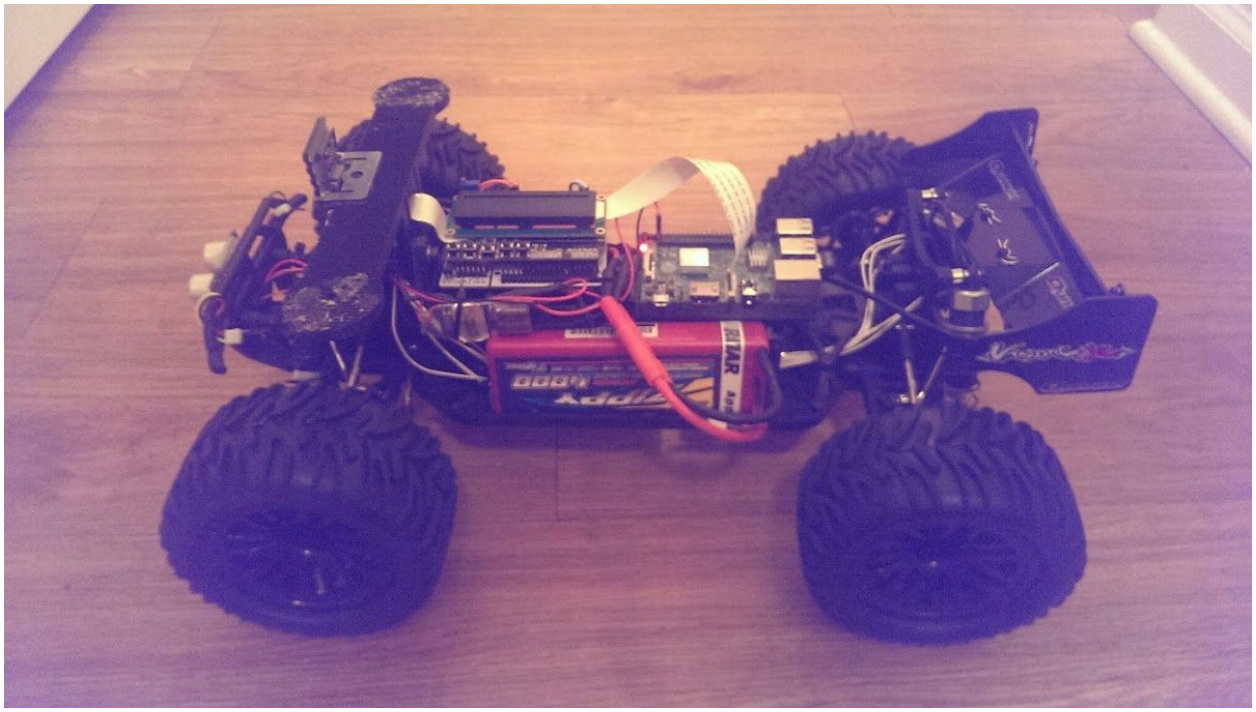
But more than technical skills, this class also gave me a strong management lesson, showing me that it is really difficult to deliver a working robotic project in a limited amount of time.

Moreover, I will probably remember for the rest of my career that as an engineer, my primary goal has to be to "**focus on the mission**" and that complexity is usually the enemy of success in meeting the specifications.

I sincerely thank Dr. Arroyo, Andrew Gray and Jake Easterling for that.

Appended

Picture gallery




QUANUM