

UNIVERSITY OF CALIFORNIA,
IRVINE

Robotic Cars Test Platform for Connected and Automated Vehicles

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Mechanical and Aerospace Engineering

by

Marc Julià Carrillo

Thesis Committee:
Professor Kenneth Mease, Chair
Professor Wenlong Jin
Professor Solmaz Kia

2015

TABLE OF CONTENTS

LIST OF FIGURES	IV
LIST OF TABLES.....	VI
ACKNOWLEDGEMENTS	VII
ABSTRACT OF THE THESIS	IX
1. INTRODUCTION	1
1.1. Overview on Driver-less Cars	1
1.2. Goals.....	4
1.3. Examples of autonomous cars test platforms in the world.....	5
1.3.1. M City Test facility.....	5
1.3.2. “Robotic Urban Like Environment” from Boston University	6
2. ROBOTIC CARS PLATFORM SETUP.....	8
2.1. Implementation of Automated Vehicles Features in a Robotics Platform	8
2.1.1. Arduino robots.....	8
2.1.2. Sensors and detection technologies	10
2.1.3. Wireless Communications.....	11
2.1.4. Chosen Platform.....	13
2.2. Escalation to real vehicles	15
2.2.1. Optimal Velocity Model.....	16
2.3. Assembling the robotic cars.....	20
2.4. Programming the robotic cars	30
2.4.1. XBee module.....	30
2.4.2. Pixy Camera	32
2.4.3. Arduino.....	34
2.5. Communication and control of the robotic cars.....	35
2.6. Calibrating the Robotic Cars	36
2.6.1. Distance Measurement with Pixy Cam.....	36
2.6.2. Zumo Robot Speed Control	38
2.6.2.1. Straight speed test.....	38
2.6.2.2. Line following speed test.....	40
2.7. Main functionality: Line following adapting speed to previous vehicle	42
2.7.1. Declaration of variables	43
2.7.2. Setup	45

2.7.3.	Main loop	47
2.7.4.	Auxiliary functions	51
2.8.	Robotic cars power autonomy test	56
3.	TESTING THE ROBOTIC VEHICLES	58
3.1.	Formation of a traffic jam without bottlenecks	58
3.2.	Known Issues	62
3.2.1.	Pan/tilt servo problems	62
3.2.2.	Differences in performance	62
3.2.3.	Auto resetting	63
4.	CONCLUSIONS AND FUTURE POSSIBILITIES	64
4.1.	Conclusions	64
4.2.	Future possibilities	64
5.	ANNEXES	67
5.1.	Annex 1: Codes used	67
5.1.1.	Distance Measurement with Pixy Cam	67
5.1.2.	Code to print distance measured through serial monitor	69
5.1.3.	Code for straight speed calibration test	70
5.1.4.	Code for speed calibration while following a line	71
5.2.	Annex 2: Purchase orders	73
5.2.1.	1 st order (3 robots)	73
5.2.2.	2 nd order (12 more robots)	77
5.2.3.	3 rd order	79
5.2.4.	4 th order	80
6.	REFERENCES	81

LIST OF FIGURES

Figure 1. Current technologies for Assisted Driving.	1
Figure 2. MCity Test Platform for driver-less cars	5
Figure 3. <i>RULE</i> in Boston University	7
Figure 4. Arduino Leonardo board and Zumo robot for Arduino shield.	9
Figure 5. Pixy Cam with Pan/Tilt Kit.....	11
Figure 6. XBee module with XBee shield for Arduino and USB adaptor for a computer.	12
Figure 7. Acceleration rate for $T = 0.5$	18
Figure 8. Acceleration rate for $T = 1.2$	18
Figure 9. Acceleration rate for $T = 1.5$	19
Figure 10. How to trim Pan/Tilt Kit for <i>Pixy Cam</i>	21
Figure 11. Pan/Tilt Kit ready to attach Pixy Cam.	21
Figure 12. Pixy Cam with double sided tape in the back and finally stacked to the Pan/Tilt Kit.....	22
Figure 13. How to connect cable between Pixy cam and Pan/Tilt Kit.....	23
Figure 14. How to stick Pixy Cam with Pan/Tilt Kit to top of Arduino board.....	23
Figure 15. Connecting Pixy Cam to Arduino board with grey cable.....	24
Figure 16. Soldering the headers into the <i>XBee</i> shield for <i>Arduino</i>	25
Figure 17. <i>XBee</i> module in the <i>XBee</i> shield with the headers to connect it to the <i>Arduino Leonardo</i> board.....	25
Figure 18. Blue jumper selecting "32u4" configuration in <i>Zumo</i> robot shield.....	26
Figure 19. Main parts of the robotic car assembled together, with cable ties to prevent the cables from moving.....	27
Figure 20. Robotic car with green dot label attached in the back for distance measurement with <i>Pixy Cam</i>	28
Figure 21. 14 completed robotic cars present in the platform seen from the front.....	29
Figure 22. 14 completed robotic cars present in the platform seen from the back.....	29
Figure 23. Main screen of <i>XCTU</i> software used to configure the <i>XBee</i> modules and communicate with the robotic cars.	31
Figure 24. Main screen of the <i>PixyMon</i> Software to set the color signatures of the <i>Pixy Cam</i>	33
Figure 25. Main screen of the <i>Arduino</i> software to program the codes that determinate the behavior of the robotic cars.	34

Figure 26. Graph of Distance against Object Size for distance measurement calibration.....	37
Figure 27. Graph relating real speed of the robotic car going forward with the <i>Speed</i> variable.....	38
Figure 28. Graph relating the <i>Speed</i> variable with the actual speed of the robotic car while following a line.....	40
Figure 29. Graph of Robotic Cars autonomy test.....	57
Figure 30. 11 robotic cars driving in a ring road.....	60

LIST OF TABLES

Table 1. Distance with previous vehicle for different *Tau* and *Speed* [Vehicle lengths]..17

ACKNOWLEDGEMENTS

First of all, I would like to thank the Institute of Transportation Studies, and specially Professor Wenlong Jin, for giving me the opportunity to work in such an interesting project as this one. My background is in mechanical engineering focused in automation, but I wanted to carry on research related to autonomous cars and traffic management, where I see a lot of future, so when Professor Jin explained me his idea of a small scale robotic cars test platform I knew I had found my research topic. I'm also very grateful for all his guidance along the project, and the help he gave me with things related to civil engineering that I was not so familiar with. He was also very nice and giving me freedom to work and trust to take decisions in which components to use for the platform. That made me learn a lot about how to organize my work and advance independently.

I want to give special thanks to Professor Kenneth Mease, who kindly agreed to serve as chair of this Thesis, acting as a link between the ITS and the Department of Mechanical and Aerospace Engineering, to which I belong. Professor Solmaz Kia deserves a special mention too, as she agreed to review my work and even shared her knowledge and research in robotics with me, allowing me to visit her lab and talk with her students about the things they do and possible collaborations in the future.

I also want to express my gratitude to my family, and specially to my parents, who were always supportive to my decisions, even when I applied to go live at the other side of the world after already being some years abroad. My brother has also helped me a lot in the

difficult moments and important decisions. And even more now that I got to share the last months here in UCI with him, making the time far from the family easier.

I am also very grateful to all my friends here in California and the ones who visited me from Catalonia. Specially to al the Catalan group in UCI, who were totally strangers about a year ago and now are like family to me. The time here wouldn't have been so good if it wasn't for them and all the adventures we had together exploring the country on weekends and holidays.

Last but not least, all this adventure wouldn't have been possible without the funding from the Balsells Fellowship and the Generalitat de Catalunya, so I am very thankful to them for the opportunity to study and live in the United States this last year.

ABSTRACT OF THE THESIS

Robotic Cars Test Platform for Connected and Automated Vehicles

by

Marc Julià Carrillo

Master of Science in Mechanical and Aerospace Engineering

University of California, Irvine, 2015

Professor Kenneth Mease, Chair

This thesis is about implementing a test platform to carry on research about connected and automated vehicles in a small scale. The main goal is to design, set up and try the platform, together with documenting everything so other people can use it in the future.

The technology needed for autonomous cars is already available. However, a lot of testing needs to be done before they become commercially available, in order for them to be totally safe and reliable.

The test platform to be built had to be able to reproduce the main features of autonomous vehicles (self-guidance, monitoring of the surroundings, wireless communication, etc.) but keeping it simple to manage and low in price, so many vehicles could be used at the same time. For these reasons, the final vehicles were built for less than \$300 using an *Arduino Leonardo* as the main controller, a *Zumo Robot Shield* as structure, a little Computer Vision

Camera named *Pixy Cam* to monitor the surroundings, and an *XBee* module for the wireless communication. These robotic vehicles move around circuits created with black tape on white vinyl, so to keep the platform flexible and not tied to any specific place.

Finally, with the basis of the platform set up, the creation of a traffic jam in a ring road without bottlenecks has been reproduced. These has shown the reliability of the platform and its possibilities to use it for traffic management and automated vehicles studies.

1. INTRODUCTION

1.1. Overview on Driver-less Cars

Autonomous cars, also known as driverless, self-driving or robotic cars, are vehicles capable of fulfilling the transportation capabilities of a traditional vehicle without human input.

Many of the cars on the roads nowadays have electronic systems that help the human drivers. Starting from pretty old ones like “Anti-lock Braking System” (ABS) to optimize the braking distance in case of emergency, cruise control, etc. To the more modern ones like line change detection, assisted cruise control or automatic emergency braking when an obstacle is detected. Even Tesla recently updated their “Autopilot” system to allow their cars to autonomously drive in highways, including changing lanes when necessary.

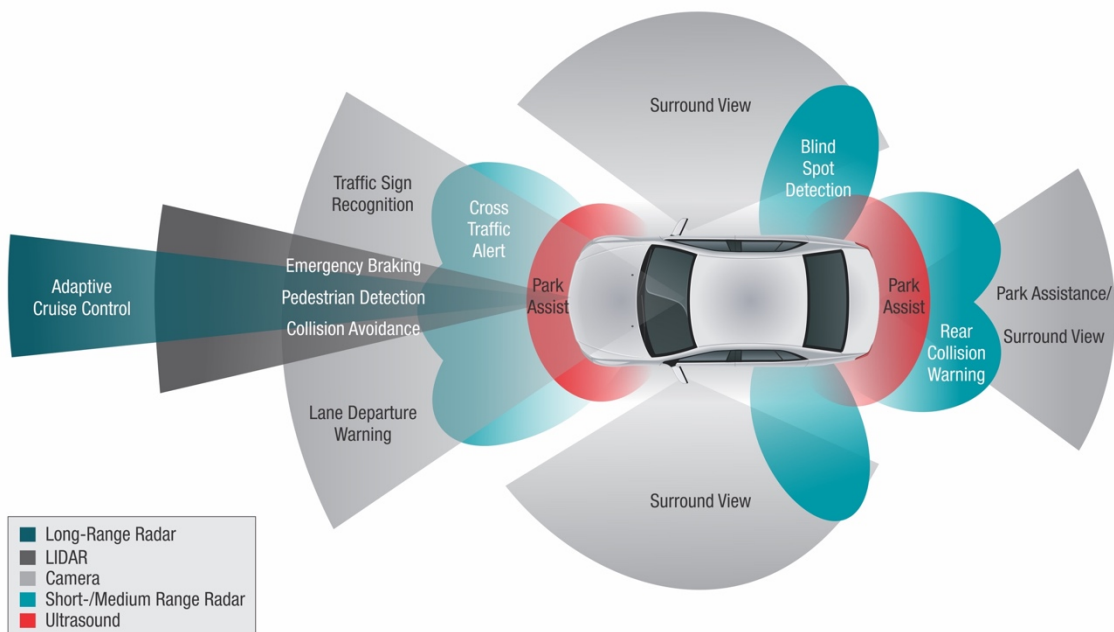


Figure 1. Current technologies for Assisted Driving.

However, all these systems still need of a human driver paying attention behind the wheels to take action when needed. There are many unexpected things that can go wrong suddenly, and even if they are very unlikely they will happen at a certain moment taking into account the amount of cars driving around. A driver-less car needs to be able to handle all these situations fast and in an optimal way.

That is the reason why driver-less cars will not come as an upgrade to the existing assisted driving systems, but as a whole new concept. Google, for example, has been working for years in autonomous cars, even driving for thousands of miles through real roads without almost any accident; and their concept of driver-less car does not even have a steering wheel, resembling more the concept of an “elevator style” vehicle that brings you from A to B than a normal car that just drives itself.

Apart from the usual sensors included already in many cars nowadays, these new concept of vehicles will need a technology capable of completely monitoring their surroundings in real time, adapting their maps and location on the go. The main technology used for that is called LIDAR, which relies on Laser light used like a radar to map the environment very precisely. With this and computer vision to recognize traffic lights and signals, a software can be programmed that drives the car more efficiently and safer than human drivers.

It is important to note that this software will not be the typical software with “if” and “while” conditions found in computers nowadays. As it needs to react to many different circumstances, some of them new to the platform, the approach used is based on Machine Learning. The software of the cars will learn by probabilistic rules while driving, for example avoiding an obstacle in the road one way or another depending if it is a ball, a dog or a person

crossing; or reducing the speed when detecting certain patterns that are usually related to children playing around. These kind of software, however, needs a lot of training and testing, in the most real conditions possible. Here is where robotic platforms like the one built in this thesis can help to get data, try features and detect possible problems.

Having a big amount of cars autonomously driven by onboard computers also opens the possibility to optimize the whole traffic flow using Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) communication. With all the cars connected in a peer to peer network, vehicles can benefit from information collected by other drivers and adapt better to sudden events like congestions, accidents, road blocks, etc. while getting coordinated to optimize the traffic flow instead of each individual vehicle acting on each own. Moreover, the big network of vehicles allows sending a message far beyond the original range of the emitter, by hopping it from one car to another.

1.2. Goals

The main goal of this Thesis is to set the basis of a test platform with small robotic cars that can imitate the main features of real autonomous cars; and test its reliability and functionality to try traffic management algorithms on it.

1.3. Examples of autonomous cars test platforms in the world

1.3.1. M City Test facility

On July 20th 2015, University of Michigan celebrated the grand opening of *M City*, a unique test facility where autonomous vehicles can be tried under many different circumstances before taking them to real roads. It consists on 32 acres of terrain where they recreated a whole urban environment with crossroads, traffic lights, signals, buildings, etc. It features all kinds of elements or obstacles that can be present in the real world, from curbs, benches or hydrants, to construction barriers, different road surfaces, variable light conditions or even robots representing people in the area.

This test facility can be very useful for the cars to learn from many different possible situation and dangers in a safe way, but is far bigger and more complex than the intended test bench for our project.



Figure 2. MCity Test Platform for driver-less cars

1.3.2. “Robotic Urban Like Environment” from Boston University

The most similar thing to the platform intended to be built was found in Boston University. It was a platform created some years ago that they called “Robotic Urban Like Environment” (RULE). It consisted of a circuit of streets, crossroads and traffic lights, with a group of robots navigating through it. They used it mainly to try algorithms to optimize parking search, with the robots going around looking for a free spot to park.

They explained that the software was written by themselves, based in Windows, without using any special environment like *ROS (Robot Operating System)*. The robots have a unique combination of three colored circles in their top, used by the main software to get the robots information (such as ID, position and direction) through 2 webcams installed in the ceiling.

The traffic lights were implemented using an infrared LED that turns on when the light is red. The robots detect this light with their infrared sensors.

Traffic lights communicate with the main PC through *XBee*, and robots through *WiFi* or Bluetooth. The different elements can communicate directly between them, but they said they usually route all the communications through the PC.

Finally, they also said the platform had been changing during the years, and they set different configurations depending on the application of interest every time. The board of the platform they are using nowadays is 11x8.5 sq. ft. (3.4x2.6 m²).

The robots used by the research group in Boston University were an already built robot called *Khepera 3*, with lots of sensors and possibilities but a bit too big and expensive for what was intended in our project.



Figure 3. *RULE* in Boston University

2. ROBOTIC CARS PLATFORM SETUP

2.1. Implementation of Automated Vehicles Features in a Robotics Platform

The first stage to build the test lab was to think about the different technologies and platforms to use in order to replicate the most interesting functionalities of real cars in a small scale workbench.

2.1.1. *Arduino robots*

Arduino was chosen as the best platform to take care of the main control of the robots. Arduino is a very popular platform, with a decent processing capacity, good price, and many upgrade possibilities due to the amount of hardware compatible with it. Different options were found:

- a) **Official *Arduino Robot***: Very basic but with a lot of space for adding sensors and options. Not very good reviews. [1]

Price: 200\$ + 60\$ sensors kit

- b) ***Parallax BoeBot Robot Shield with Arduino***: Arduino compatible version of a very popular robotic kit. Lots of accessories from the main manufacturing company. [2]

Price: 150\$ + 80\$ (sensors for distance measuring and line following) + 60\$ (Bluetooth module).

c) **Zumo robot for Arduino:** Well built robot with good motors and lots of sensors included (reflectance sensor for line sensing, 3-axis accelerometer, gyro, compass...). Possibility to add distance sensor and others hardware. [3]

Price: 100\$ + 25\$ (Arduino Leonardo) + 30\$ (Bluetooth module)

d) **Make Rovera 2WD Arduino Robotic Kit:** Arduino robot kit that come with distance sensor and reflectance sensor for line following already built in. Together with IR sensor to control it from any TV remote. [4]

Price: 170\$

e) **Sparki:** New robot with lots of sensors integrated. It features distance measurement, line following and edge detecting, light sensing, accelerometer, magnetometer, remote control and even Bluetooth module. However, it seems a bit more difficult to upgrade and the software environment is not so standard. [5]

Price: 160\$

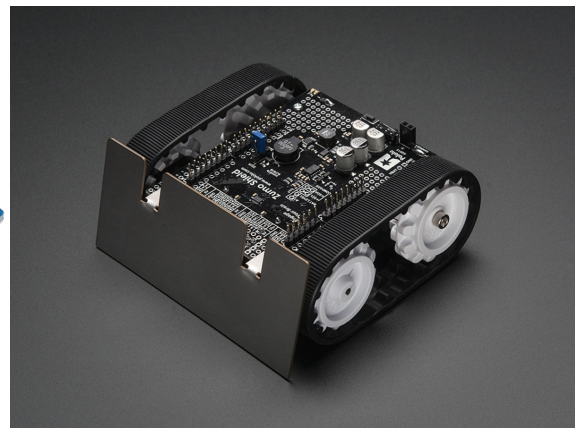
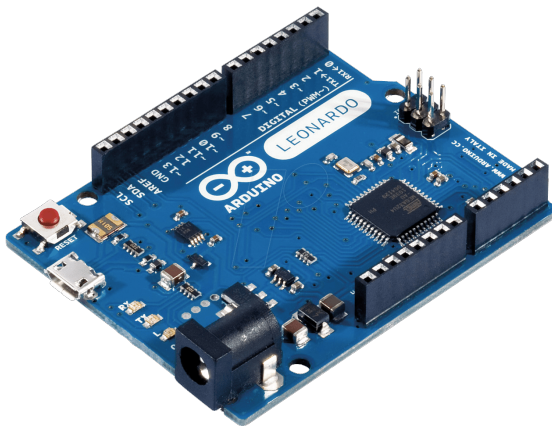


Figure 4. Arduino Leonardo board and Zumo robot for Arduino shield.

2.1.2. Sensors and detection technologies

Implementing a LIDAR system in small robots in order to monitor the surroundings would have been very expensive and complicate. Moreover, the main goal of the robotic cars test platform was to perform traffic management studies, therefore there was no need to monitor the surroundings in real time looking for pedestrians or other sudden problems. So that, it was decided to limit the amount of technology needed by reducing the platform to a “2-dimension like” environment. So the robotic cars can follow a track and interact with other robots, signals, etc. along this track.

To create the track, line following seemed the best option, as with a simple array of light sensors one can program a robot to follow a black line drawn on a white surface.

In order to implement car following feature and signal detection many options were contemplated. Ultrasound or infrared distance sensors are pretty accurate when trying to detect and follow objects, but lack the capability to differentiate the type of object seen. With LIDAR also discarded for being too expensive and complicate to implement, computer vision appeared as a balanced option. Precisely, a sensor called Pixy was found that seemed to provide all the features needed.

The *CMUCam5 (Pixy)* [6] is a camera integrated with a chip that can recognize and track different objects by their colors. It does also all the computing and just sends the useful information (number and kind of objects, location, etc.) to the Arduino board. This is done very fast (50 frames analyzed per second they say), so it can be used for following these

objects. And it can also be used to detect objects with color codes, so it can be implemented to detect traffic lights or other traffic elements in the workbench.

This same camera or a similar one could be used to implement a Computer Vision system on the ceiling of the lab, and so control, position and monitor the robotic vehicles. Like creating and indoor positioning system.

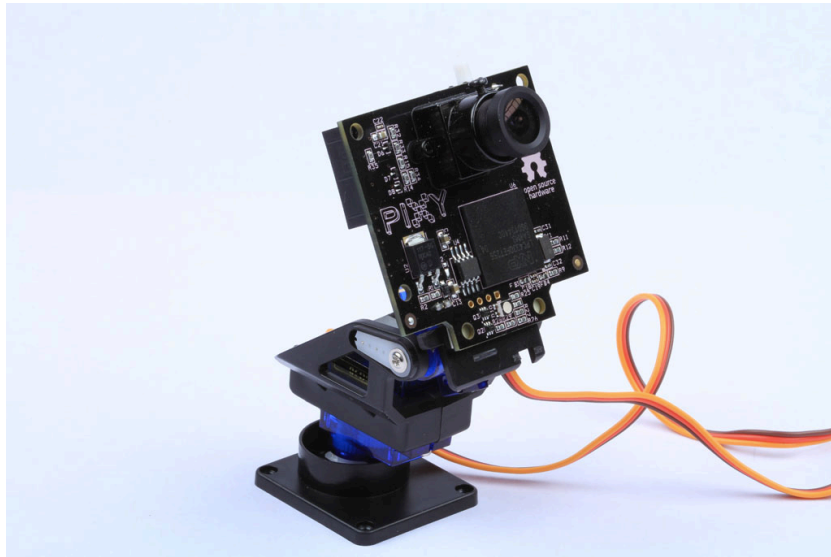


Figure 5. Pixy Cam with Pan/Tilt Kit.

2.1.3. Wireless Communications

Regarding wireless communication, the main features needed were fast speed and refresh rate and low power usage, but some meters of range were already enough. The most common wireless technology, *WiFi*, seemed too powerful for the size of the robots and intended use. *Bluetooth* seemed a better option, with reduced size and range, but still with a pretty high power usage. Finally, a technology called *XBee* was found to be very suitable for

robotic applications, as it has less power consumption than Bluetooth with a similar range and better multi-device communication features (allowing more nodes on the net).

There are two kinds of *XBee* devices, *Series 1* and *Series 2*. *Series 1* modules work out-of-the-box in a point-to-multipoint configuration; *Series 2* need more setup, but they work directly in a “mesh” configuration (allowing to reach devices out of range by jumping the messages through other nodes in the net). After some considerations, *XBee Series 1* were chosen for the robotic cars platform, as its range covers by far the whole setup and with these kind of devices the basic control and communications from the computer to all the robots can be done in a very easy way. If more complex communication is needed in the future (like talking to a specific car from the computer or talking directly from one specific vehicle to another), the firmware of the modules can be changed to a platform called *ZigBee*, which brings them capabilities similar to those of an *XBee Series 2* module.

To attach the *XBee* module to the Zumo robot, an *XBee Arduino* shield plus headers needs to be used. [7] [8]

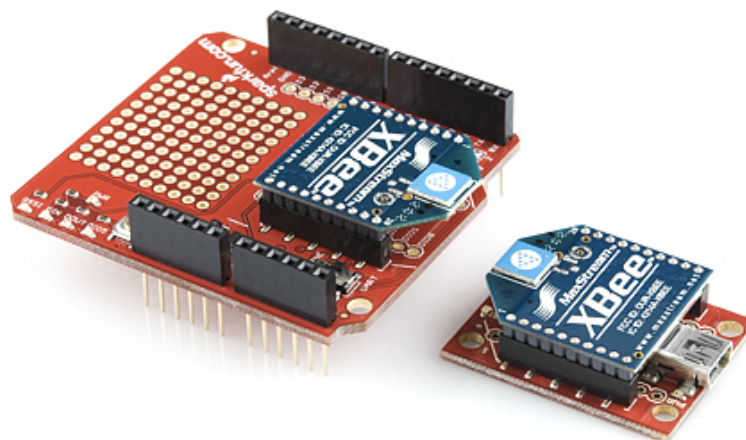


Figure 6. XBee module with XBee shield for Arduino and USB adaptor for a computer.

2.1.4. Chosen Platform

After all the previous considerations, **Zumo Robot for Arduino + Pixy CMUCam-5 + XBee Series 1** was the platform chosen. The total price of each robot unit with Pixy Cam and wireless module would be around **270\$**.

Main advantages of Zumo robot for Arduino + Pixy Cam robotic platform:

- + Can be used with rechargeable batteries, that can be recharged directly from inside the robot.
- + 3-axis accelerometer + magnetometer + gyroscope integrated.
- + Good adaptable speed.
- + Good line following capabilities.
- + Possibility to add IR or Ultrasonic sensor for obstacle detection.
- + Can track objects with Pixy Cam.
- + Can estimate distance to object with Pixy Cam (by size change).
- + Can save and load Pixy parameters (to have exactly the same in all the robots).
- + Easy to implement remote control and V2I Communication with XBee Series 1.
- + Good Price.

Main disadvantages of Zumo robot for Arduino + Pixy Cam robotic platform:

- Distance to object and distance travelled not very precisely measured.
- Lack of encoders in the wheels to get feedback of speed.
- Requires some assembly (Pixy Cam and wireless communication adaptor).
- Need change of firmware in XBee Module to implement V2V Communication.

2.2. Escalation to real vehicles

In order to compare the speed and data collected from the robotic vehicles to those of real vehicles, an escalation factor needs to be determined. This factor has been chosen to be **50**, as the *Zumo* robots measure 10 cm in length, thus being equivalent by a factor of 50 to a 5-meter-long car, a pretty average length for passenger cars nowadays.

Using this escalation factor we can convert the speeds of the robots to real car speeds. The theoretical maximum speed in the robots is around 60 cm/s, but when following a line and in our floor conditions we reached around 45 cm/s as maximum speed. Applying the 50 escalation factor this is equivalent to 81 km/h (50 mph), a decent speed for an urban environment.

This escalation factor, however, can't be directly applied to the mass of the vehicle. Our robots with the accessories and batteries weight around 400 grams, which multiplied by 50 would give a 20 kg vehicle in real size. This means we can't count on inertia of the vehicle to emulate the acceleration and braking conditions of real cars, as these little robots can stop or apply big changes in their speed in almost real time without drifting. So that, software models that imitate the conditions we want to reproduce will need to be implemented. The basic model that will be used to reproduce human driving is the so called "Optimal Velocity Model".

2.2.1. Optimal Velocity Model

In order to reproduce the behavior of real drivers, the Optimal Velocity Model is used to calculate the desired speed for the vehicle at each moment, based on the distance with the previous vehicle.

This model first calculates the speed needed in order to reach the previous vehicle in a specific time (a Tau constant, which in practice will determine the distance left with the previous vehicle). Then, it calculates the acceleration needed to reach this speed in another specified time (a constant T , found to be around 1.2 seconds in human driver behavior). Finally, it calculates the speed to apply after a specific time (the typical reaction time of people when driving), considering the current speed and the calculated acceleration.

Changing the Tau parameter one can allow bigger or smaller distances among vehicles, in order to emulate real human driving or to increase density of vehicles thanks to the fast response of electronics and robots. Table 1 shows some distances at different speed depending on the Tau used:

Table 1. Distance with previous vehicle for different Tau and $Speed$ [Vehicle lengths].

		Speed		
		25 km/h (16 mph)	50 km/h (31 mph)	80 km/h (50 mph)
Tau	0.5	0.7	1.4	2.3
	1	1.4	2.8	4.5
	1.6	2.2	4.5	7.1
	2	2.8	5.5	8.9

The T parameter can be modified too, affecting how fast the robot vehicle accelerates or brakes. So the lower the value, the faster the robot will be able to adapt to changing conditions and the more precisely will mimic the movements of the previous vehicle.

Using *Matlab*, some simulations of the behavior of the robots with different T values can be done. With the standard T of 1.2 to imitate human driver behavior, the robot vehicles take around 5 seconds to reach the maximum speed of 45 cm/s (equivalent to 80 km/h with the escalation previously defined). Changing the value of T we can get different acceleration rates, as it can be seen in the following figures.

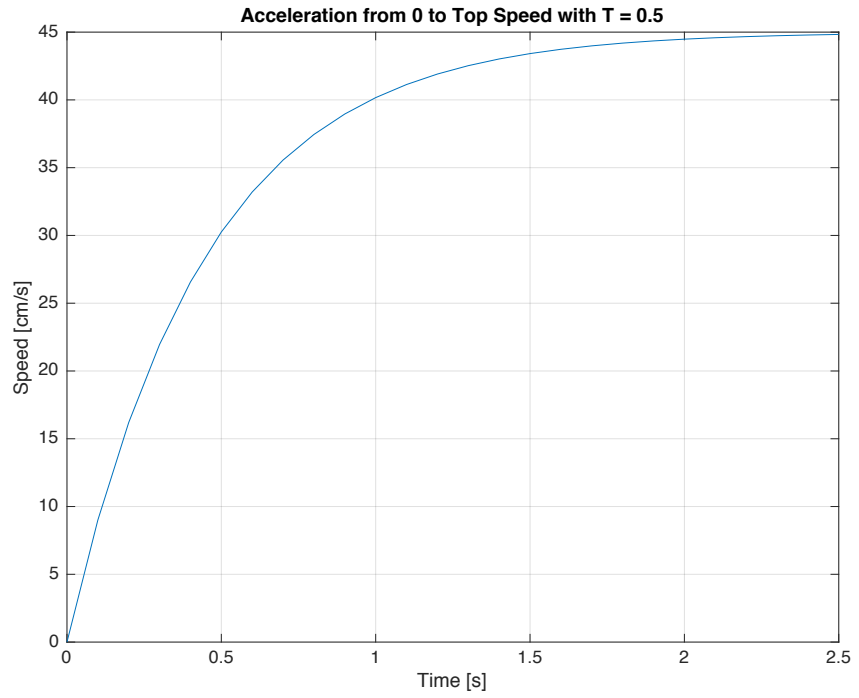


Figure 7. Acceleration rate for $T = 0.5$.

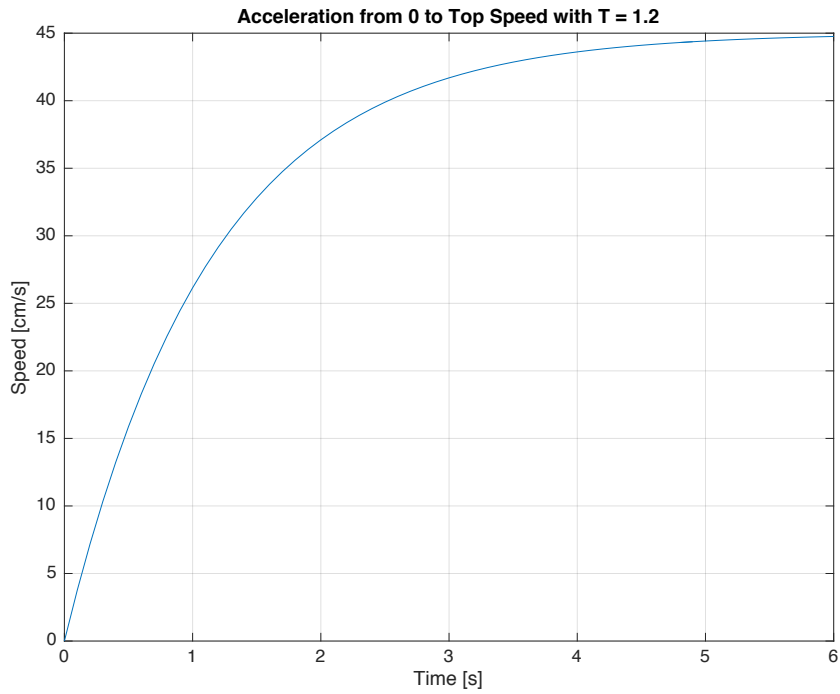


Figure 8. Acceleration rate for $T = 1.2$.

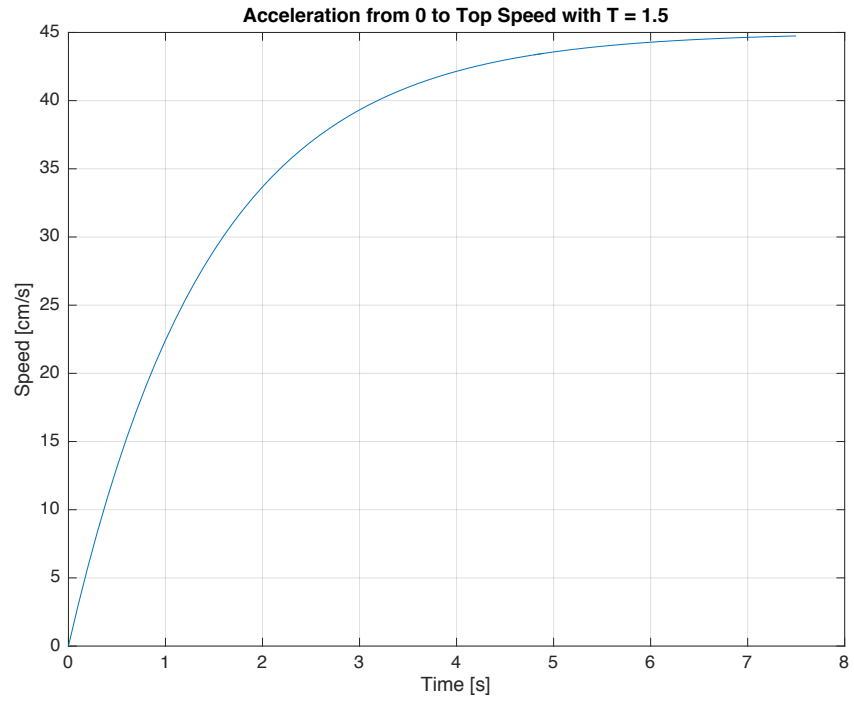


Figure 9. Acceleration rate for T = 1.5.

2.3. Assembling the robotic cars

The robot vehicles are based on the *Pixy Pet Robot* project by *Adafruit* [9] adding some wireless possibilities and adapting them to the specific needs of the platform. They are built by putting together 4 main parts:

- An ***Arduino Leonardo*** board to act as main controller of the robot.
- A ***Zumo Robot for Arduino*** shield, which gives the basic structure to the robot, including wheels, basic sensors, buttons, battery case and an electronic board to connect everything with the Arduino.
- A ***Pixy CMU5 Camera*** and a ***Mini Pan/Tilt kit Assembled with Micro Servos***, which brings computer vision to the robot by processing the images and sending the data to the Arduino.
- An ***XBee Series 1*** wireless modules with its Arduino shield, in order to provide the robotic vehicle with wireless communication capabilities.

To assemble the robotic vehicles, the following steps need to be followed:

1) Assemble the Pixy Cam with the Pan/Tilt kit

As it is very well detailed in *Adafruit's Pixy Pet Robot* project webpage [9], the first thing to do is to cut all the tabs that stand out from the pan/tilt kit mounting bracket, so to get a flat

surface where to attach the *Pixy Cam*. One of the sides needs to be cut a bit too in order to make space for the cables, as it can be seen in the following figures.

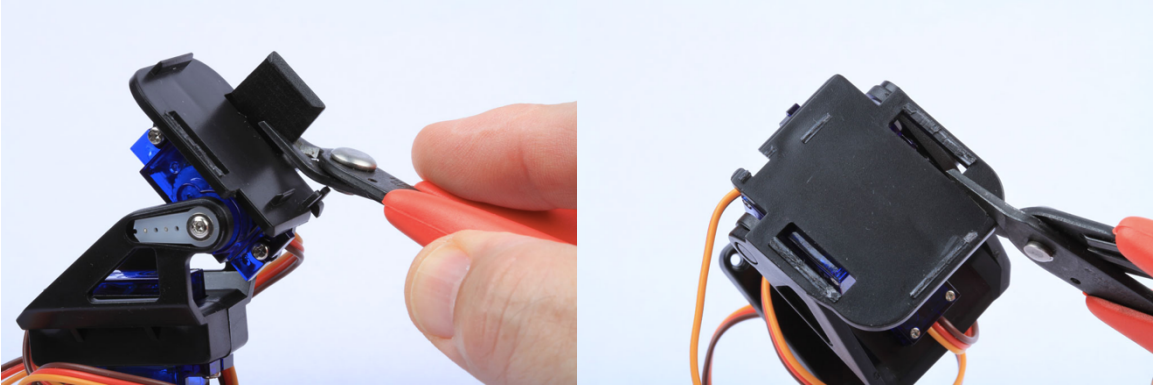


Figure 10. How to trim Pan/Tilt Kit for *Pixy Cam*.

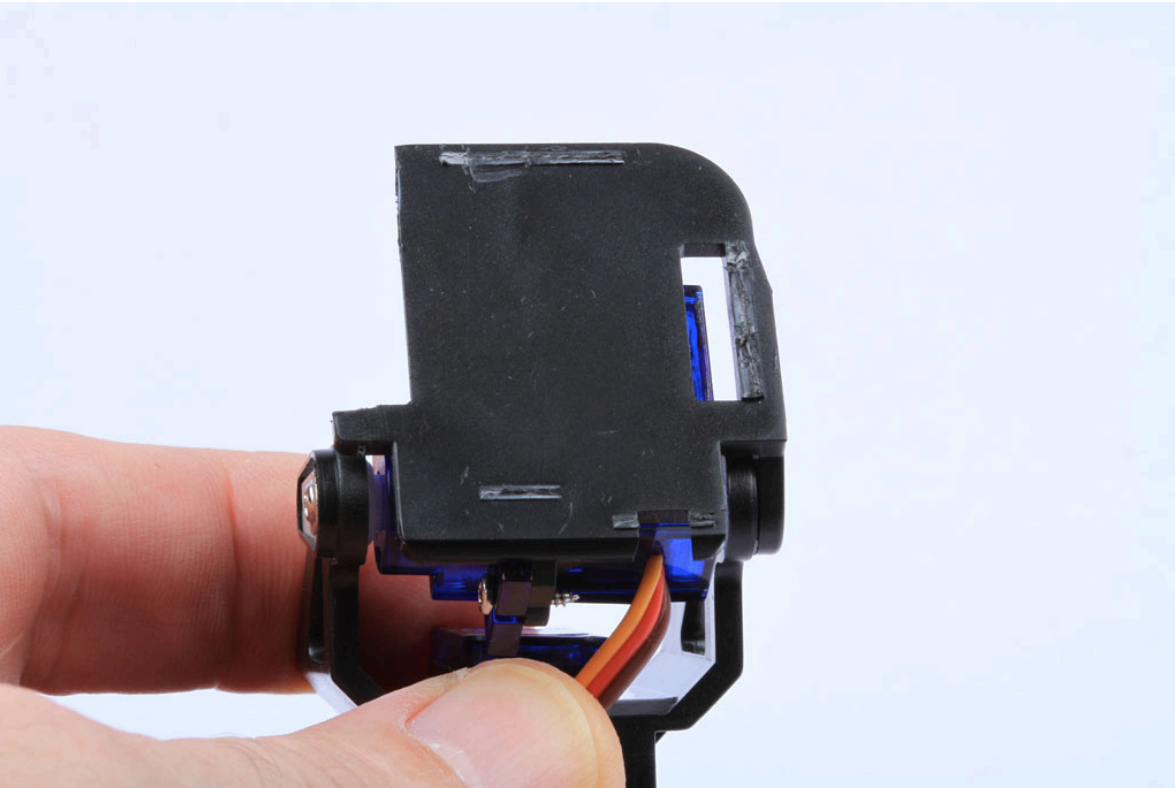


Figure 11. Pan/Tilt Kit ready to attach *Pixy Cam*.

Later, the *Pixy Cam* module needs to be attached to the flattened surface of the pan/tilt kit using double sided tape.

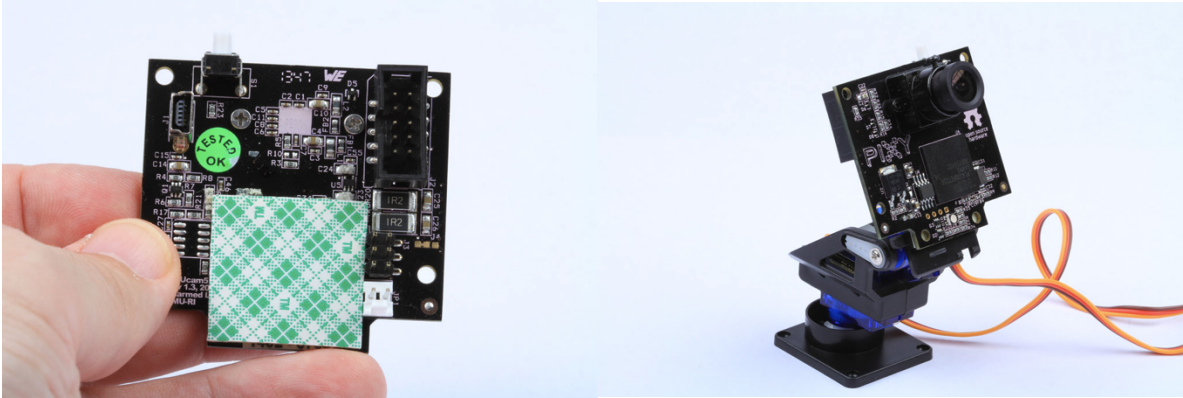


Figure 12. Pixy Cam with double sided tape in the back and finally stacked to the Pan/Tilt Kit

Once the camera is attached to the moving support, the cables need to be connected as shown in the pictures below, using these 3 guidelines:

- Brown cable always in the bottom / Yellow cable always on top.
- Pan servo cable (the bottom one that comes from the back) goes to the left.
- Tilt servo cable (the top one that comes from the front) goes to the right.

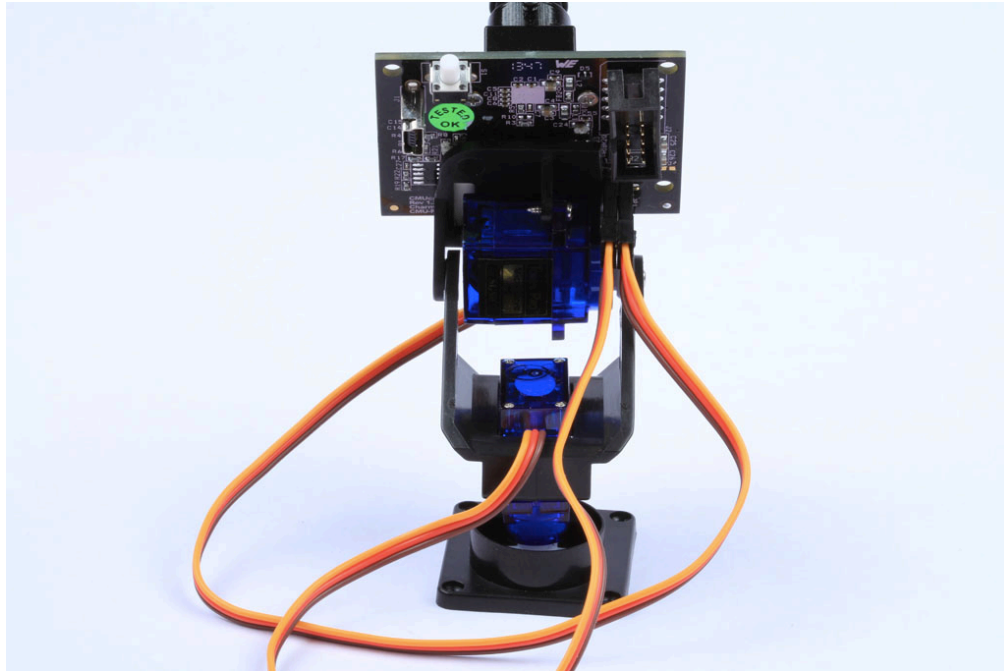


Figure 13. How to connect cable between Pixy cam and Pan/Tilt Kit.

Then, using double sided tape again, the pan/tilt support with the camera is attached at the top of the *Arduino* board, trying to keep it aligned with the dotted line above “RoHS Compliant” and with the front part starting between the “C” and the “E” in “FC CE”.

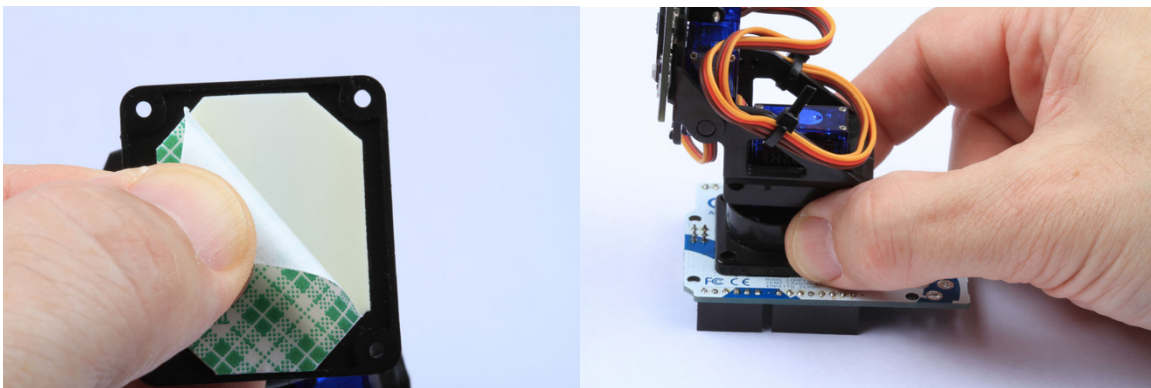


Figure 14. How to stick Pixy Cam with Pan/Tilt Kit to top of Arduino board.

Finally, the grey ribbon cable that came with the *Pixy Cam* needs to be connected from the camera to the Arduino. Before that, however, the little black plastic covering a bit of the

Arduino 6-pin port needs to be taken out in order for the base of the cable not to get out too much.

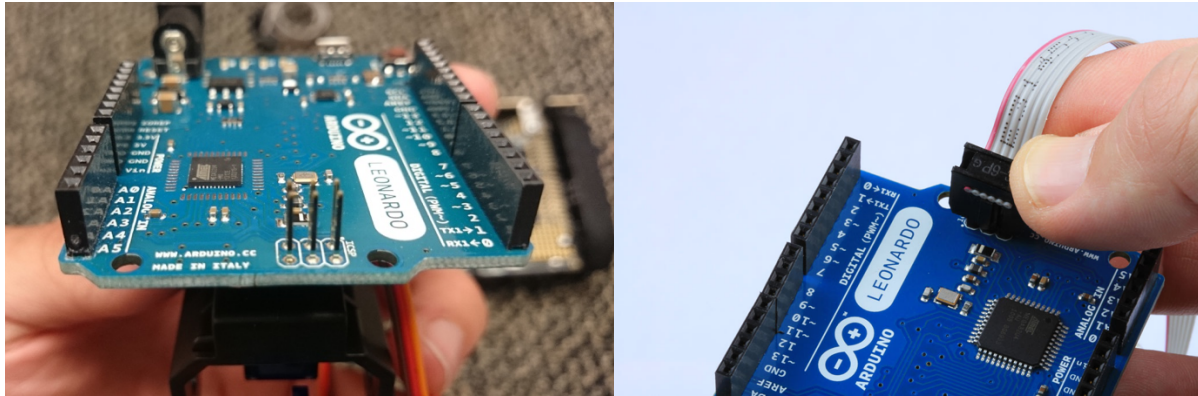


Figure 15. Connecting Pixy Cam to Arduino board with grey cable.

Now, the module with the Arduino and the *Pixy Cam* is complete, and could be attached directly to the *Zumo* shield. However, the *XBee* module needs to be attached in the middle in order for the robot to have wireless capabilities.

2) Solder the *XBee* shield

The *XBee* shield for *Arduino* comes without headers, so a set of headers needs to be soldered in order for it to be attachable to the *Zumo* shield and the Arduino. The black part of the headers needs to be on the same side as the back header where the *XBee* module will be attached, as it can be seen in the following pictures.



Figure 16. Soldering the headers into the *XBee* shield for *Arduino*.

Once the headers are soldered, the *XBee* module can be attached to the shield (following the guide lines drawn for the angled part in order to get the right orientation). Finally, it is very important to set the little switch in the shield to “UART” mode instead of “DLIN”, in order for the *XBee* to communicate through the “Serial1” port of the *Arduino Leonardo*.

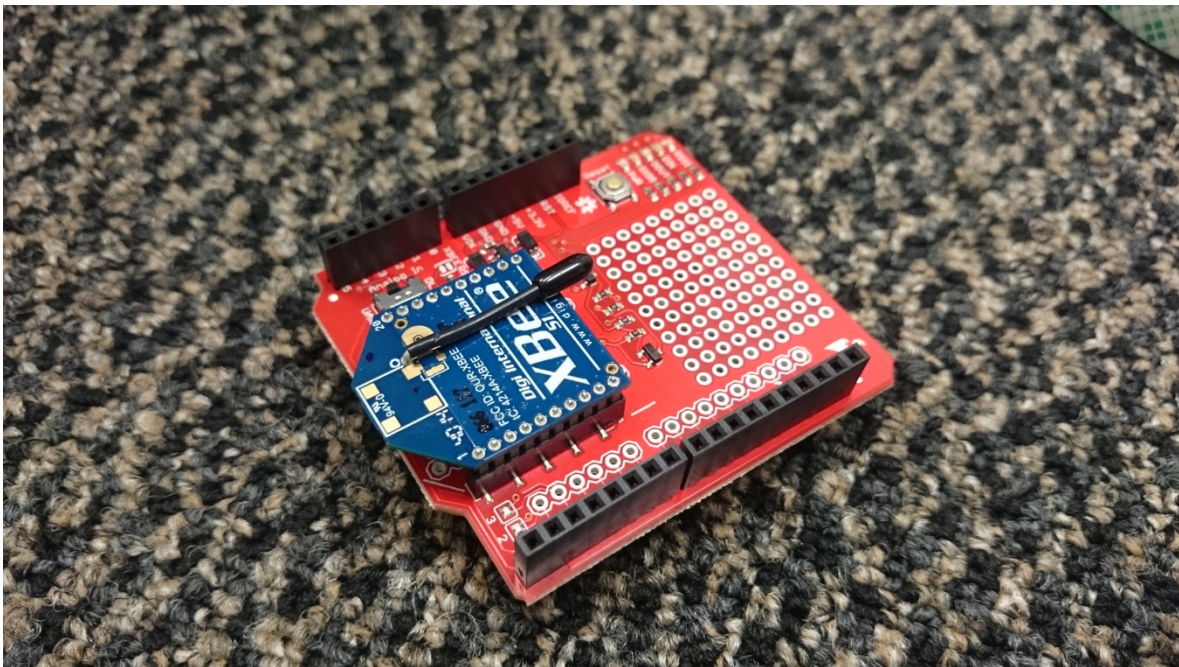


Figure 17. *XBee* module in the *XBee* shield with the headers to connect it to the *Arduino Leonardo* board.

3) Attach everything together

Before putting everything together, one last adjustment needs to be done in the *Zumo* robot shield. The blue jumper allowing two different configurations needs to be set covering the two pins closer to the front, in order to select “32u4”, the type of Microcontroller present in the *Arduino Leonardo*, as it can be seen in Figure 18.

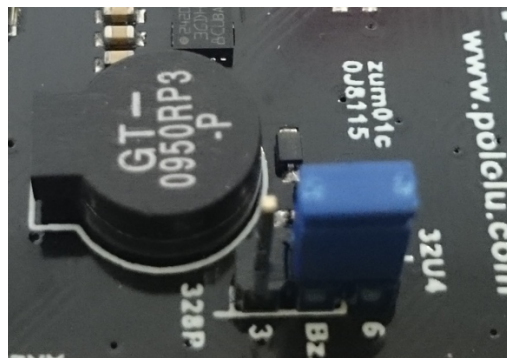


Figure 18. Blue jumper selecting "32u4" configuration in *Zumo* robot shield.

Now the robot can finally be assembled by attaching the 3 main modules obtained (the *Zumo* robot shield in the bottom, the red *XBee* shield upside down in the middle, and the *Arduino* with the *Pixy Cam* on top of everything). These three parts are now easy to mount and dismount, in case something needs to be changed or more layers need to be attached to add features.

Finally, the cables connecting the servos and the camera can be organized using cable ties, so they don't annoy or get stuck with anything when the robot is moving. When doing so, it is important to check there is enough slack for the camera module to pan and tilt in all directions.

It is good to point also that the camera comes with a black plastic cover in front of the lens, to protect it from dust and scratches. This cover can be taken out by simply pulling it (and it needs to be done before trying to use it, logically).

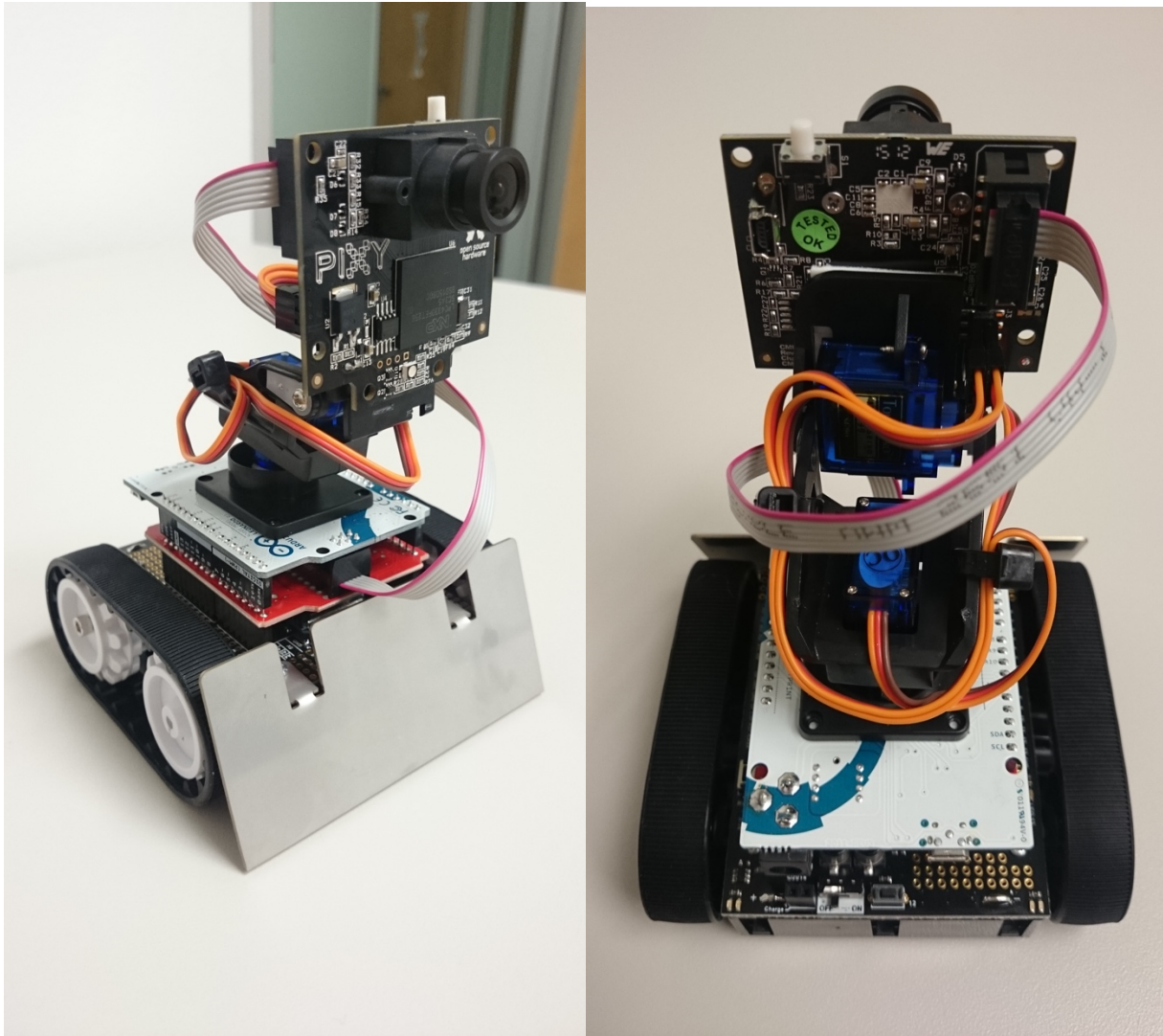


Figure 19. Main parts of the robotic car assembled together, with cable ties to prevent the cables from moving.

4) Attach label for Pixy Cam recognition

In order for the *Pixy Cam* to recognize the car in front, a piece of cardboard with a green dot has to be attached to the back of the robots. The piece of cardboard used so far was the front of the *Zumo* robot shield box, as it has an appropriate size and it is white, so one can paint easily on top of it. The green dot used was 3.5 cm in diameter so the robotic cars can guess the right distance according to the calibration set. And it has been drawn with the top edge at 0.5 cm from the top and centered (with 0.9 cm to the edges from both sides of the circle), so that it sits around the position of the *Pixy Cam*.

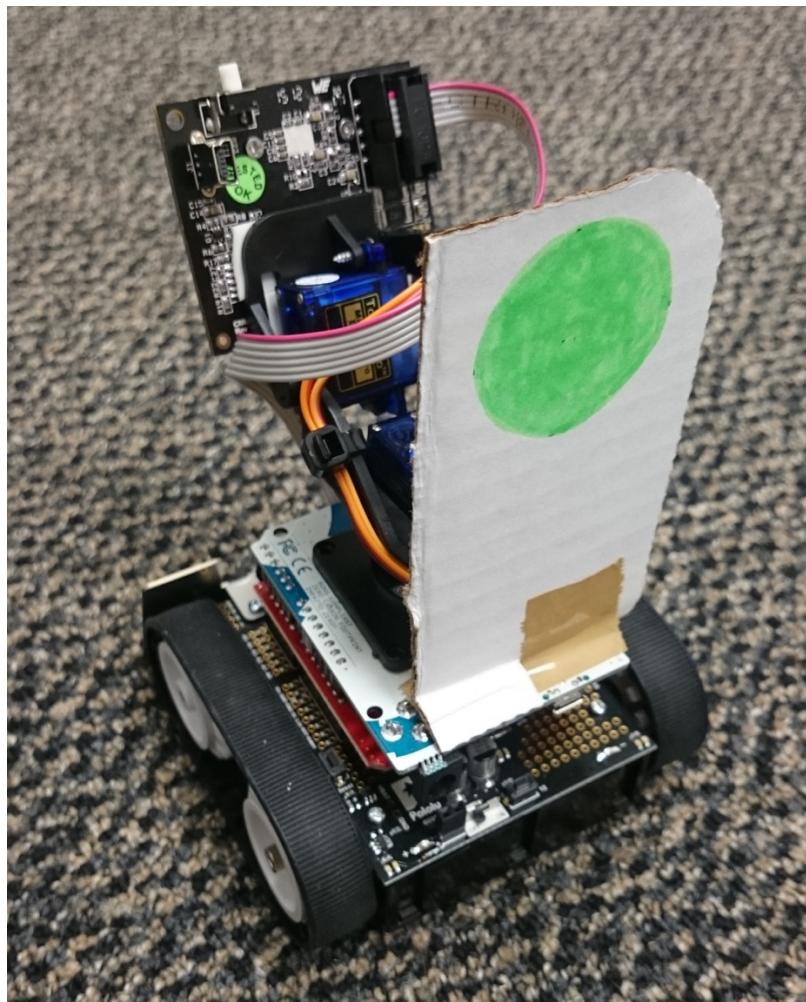


Figure 20. Robotic car with green dot label attached in the back for distance measurement with *Pixy Cam*.

To conclude this part, the 14 finished robotic vehicles currently present in the lab can be seen in the following figures.



Figure 21. 14 completed robotic cars present in the platform seen from the front.



Figure 22. 14 completed robotic cars present in the platform seen from the back.

2.4. Programming the robotic cars

There are 3 main things that need to be programmed in the robot vehicles:

- The ***XBee module***, so that it can communicate correctly with the other *XBees*.
- The ***Pixy Cam***, so that it can recognize the objects correctly
- The ***Arduino Leonardo***, in order to control what the vehicles do.

Following it is explained how to program each of these parts.

2.4.1. *XBee module*

The *Series 1 XBee* modules (the ones implemented in the robot vehicles) have to be configured with a program called “XCTU”, so they can talk to each other. *Sparkfun* has a good tutorial on how to configure and use these *XBees* with *XCTU* [10], but here the basic configuration needed for the robot vehicles will be summarized:

First of all, the *XBee* module is connected to the computer through the *XBee Explorer USB* adaptor. This can be used to change the parameters in the *XBee* module or to talk to the other *XBees* from the computer.

For the *XBees* to be able to communicate between them they need to have the same parameters, specially the “Baud Rate” (how fast they communicate), “Channel” (the frequency they use to communicate) and “PAN ID” (Personal Area Network ID, a number that identifies the network that *XBee* is talking in).

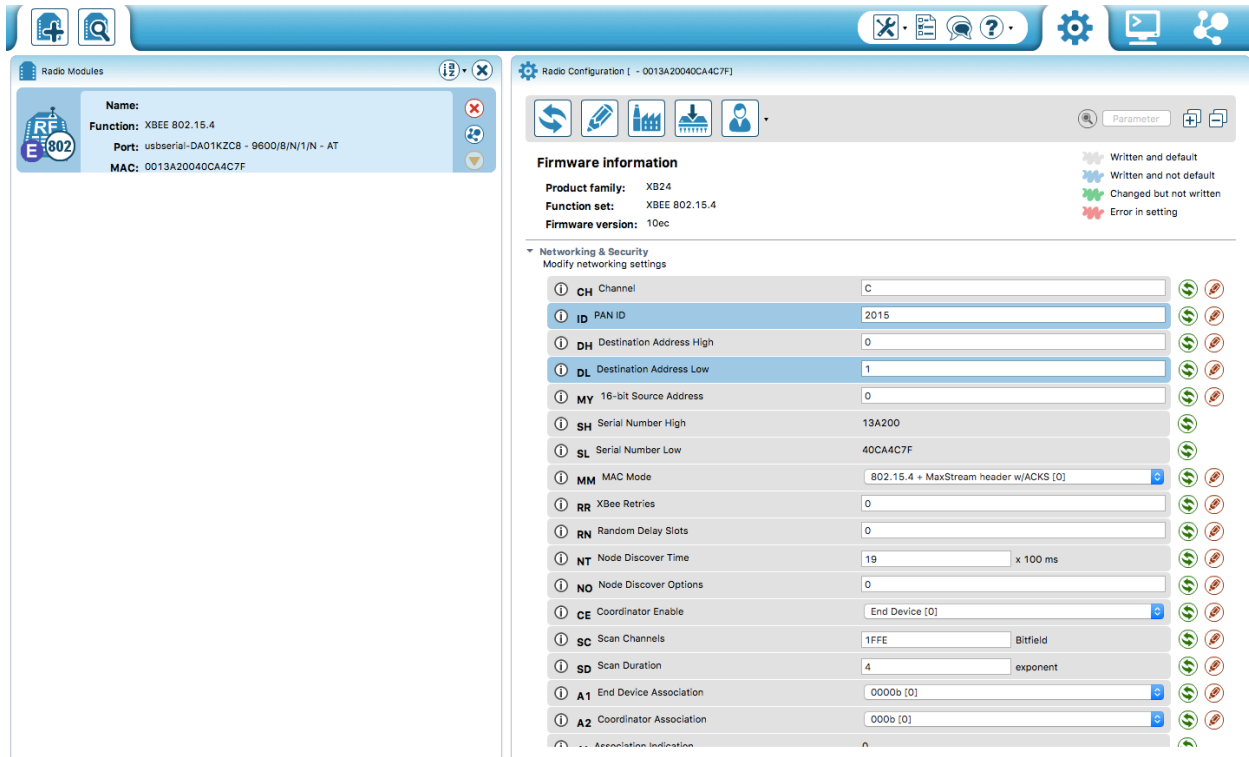


Figure 23. Main screen of *XCTU* software used to configure the *XBee* modules and communicate with the robotic cars.

The only of these parameters changed from the factory configuration is the “PAN ID”, which has been decided to be set to “**2015**”. All the *XBees* need to have the “PAN ID” set to “2015” in order to send and receive messages in the platform.

The other important parameters are the “MY” address and destination addresses (“DH” and “DL”). Each *XBee* has an address (“MY” address) and sends messages to a destination address, which will only be received by the *XBee* modules that have this destination address as “MY” address.

In order to control and send messages to all the robots at once from the computer, one of the *XBees* has been assigned a “MY” address of “0” and a destination address of “1” (“DH” set to

0 and “DL” set to 1). This module is the one with the antenna extended up and will be the one connected to the computer. All the other modules, that will be attached to the robots, have been assigned a “1” into the “MY” address field, and “0” into the “DH” and “DL” destination addresses fields. This way, all the robot vehicles get the messages from the computer, while the computer gets the messages from all the robots.

2.4.2. Pixy Camera

The *Pixy Camera* needs to be configured in order to recognize a specific color pattern as an object. This can be done directly pressing the button in the camera or by using *PixyMon* (a software from the manufacturer), as it is explained in their website [11].

For the use needed with the robotic vehicles, the most comfortable way is to connect the camera to a computer using a *Mini USB to USB* cable and program it through the *PixyMon* software, as it allows the user to see how the camera is processing the images, and to store and load the parameters to every camera without having to teach the object again. However, some adjustments may need to be done for each camera, as some differences in the way they process the images have been appreciated.

The *Pixy Cam* can store up to 7 color signatures (meaning that it can recognize up to 7 different colors), and many more objects using the color code feature (different color signatures put next to each other). In order to set these color signatures, the object to be recognized needs to be placed in front of the camera, select “Set Signature #” in the “Action”

menu, and the color pattern has to be selected with the mouse. The camera should now highlight objects as soon as it detects that color pattern.

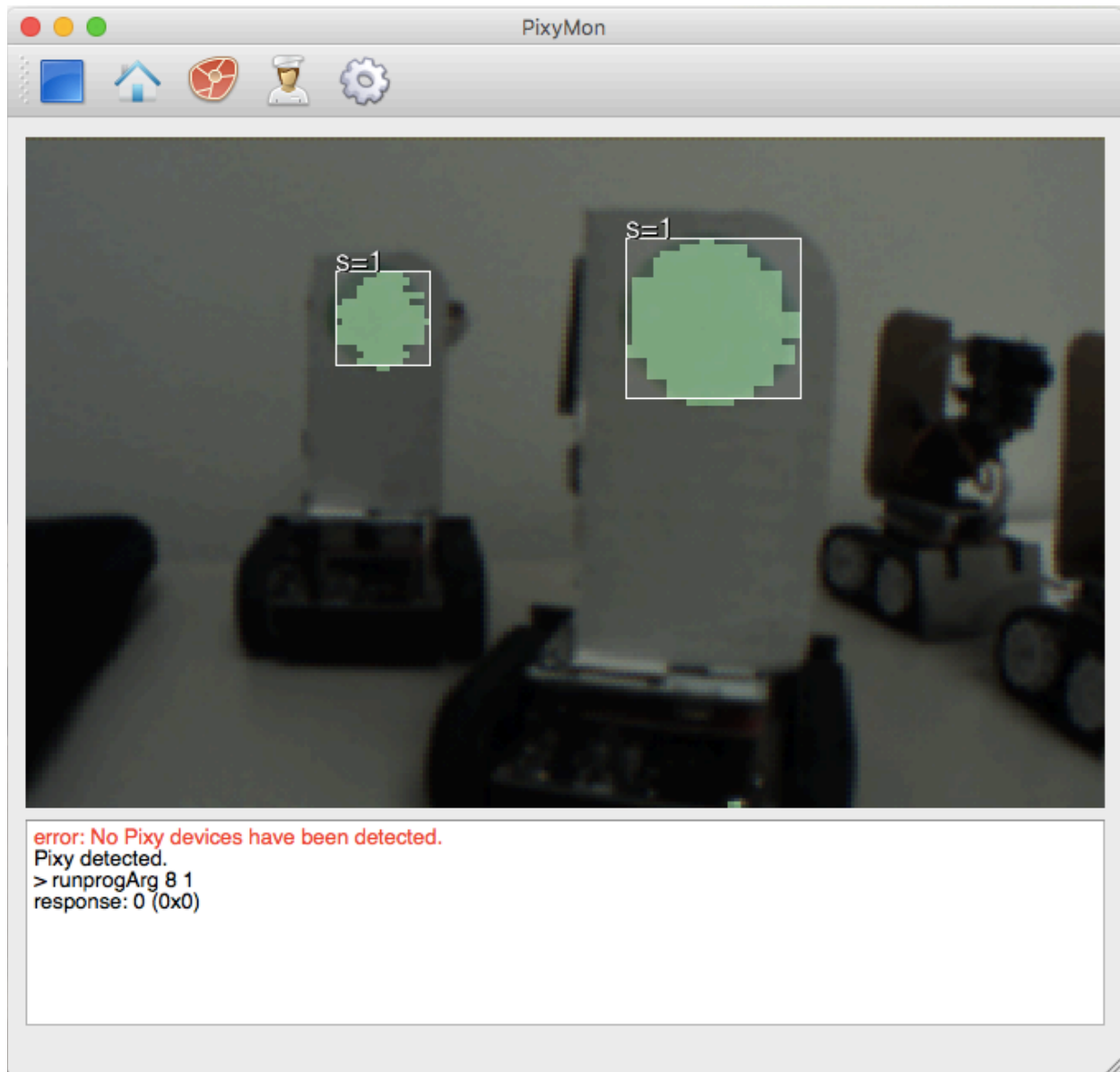


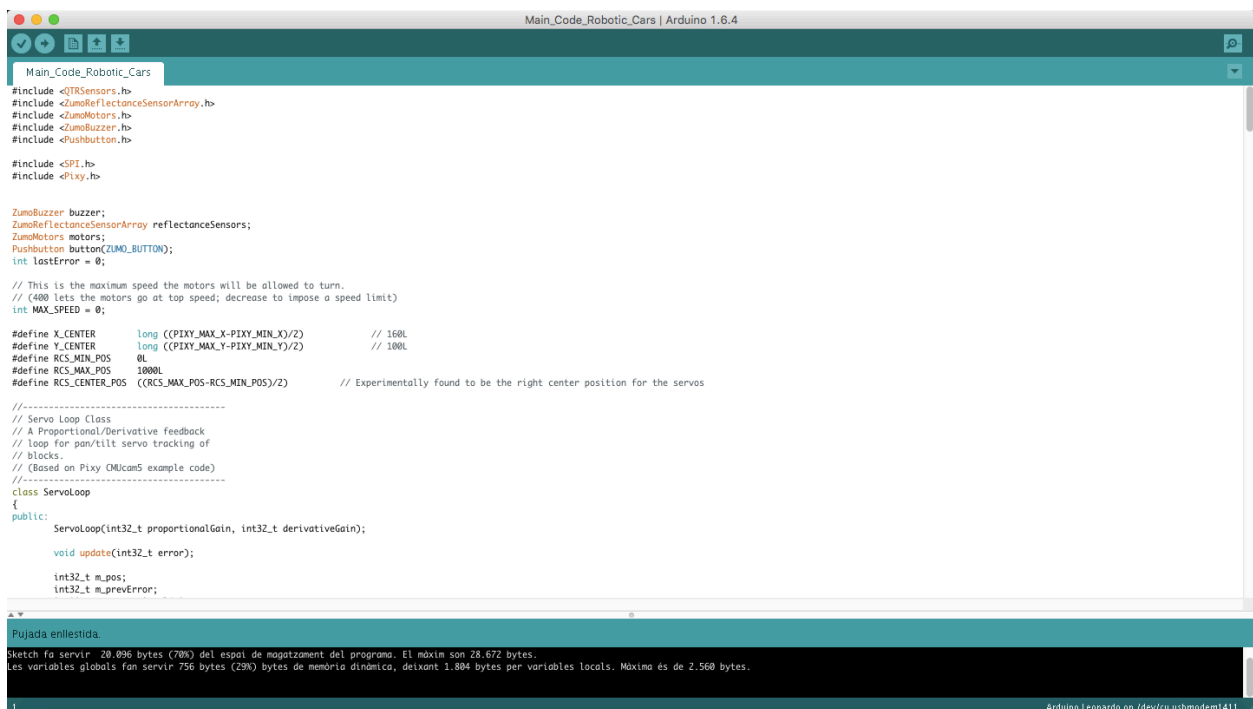
Figure 24. Main screen of the *PixyMon* Software to set the color signatures of the *Pixy Cam*.

2.4.3. Arduino

The *Arduino Leonardo* board is the brain of the robotic vehicle. It is through it that one can control all what the robot does and get data from the sensors. It can be programmed using a software called *Arduino* from the developers of the board.

The language used is based on “C”, with many libraries and resources available online to make the programming pretty simple and easy to learn. Later in this document, the different codes used in the robotic vehicles are detailed and explained.

In order to upload the code to each robot vehicle, the *Arduino* board needs to be connected to the computer using a *Micro USB* to *USB* cable.



```
Main_Code_Robotic_Cars | Arduino 1.6.4
Main_Code_Robotic_Cars
#include <I2CSensors.h>
#include <ZumoReflectanceSensorArray.h>
#include <ZumoMotors.h>
#include <ZumoBuzzer.h>
#include <Pushbutton.h>

#include <SPI.h>
#include <PLxy.h>

ZumoBuzzer buzzer;
ZumoReflectanceSensorArray reflectanceSensors;
ZumoMotors motors;
Pushbutton button(ZUMO_BUTTON);
int lastError = 0;

// This is the maximum speed the motors will be allowed to turn.
// (400 lets the motors go at top speed; decrease to impose a speed limit)
int MAX_SPEED = 0;

#define X_CENTER      long ((PIXY_MAX_X-PIXY_MIN_X)/2)      // 160L
#define Y_CENTER      long ((PIXY_MAX_Y-PIXY_MIN_Y)/2)      // 100L
#define RCS_MIN_POS   0L
#define RCS_MAX_POS   1000L
#define RCS_CENTER_POS ((RCS_MAX_POS-RCS_MIN_POS)/2)        // Experimentally found to be the right center position for the servos

//-----
// Servo Loop Class
// A Proportional/Derivative feedback
// Loop for pan/tilt servo tracking of
// blocks.
// (Based on PLxy (Mjcam5 example code)
//-----
class ServoLoop
{
public:
  ServoLoop(int32_t proportionalGain, int32_t derivativeGain);

  void update(int32_t error);

  int32_t m_pos;
  int32_t m_prevError;
};

Pujada enllestida.
Sketch fa servir 20.096 bytes (70%) del espai de magatzem del programa. El màxim són 28.672 bytes.
Les variables globals fan servir 756 bytes (29%) bytes de memòria dinàmica, deixant 1.804 bytes per variables locals. Màxima és de 2.560 bytes.
Arduino Leonardo on /dev/cu.usbmodem1411
```

Figure 25. Main screen of the *Arduino* software to program the codes that determinate the behavior of the robotic cars.

2.5. Communication and control of the robotic cars

Thanks to the *XBee* modules installed in the robots, these ones can talk with the computer and also be controlled from this last one. *XBees* use serial communication, this means that a set of characters are sent from the emitter to the receiver, and it is the receiver's job to interpret them and act accordingly. This communication system is pretty simple, but it has lots of possibilities.

Using a "Serial monitor" software in the computer (like the one integrated in *XCTU*) one can receive data from the vehicles and send other characters to them. One first use for the technology is to receive information from the vehicles. By introducing a specific line of code in the *Arduino* program, the robot can print some variables or values through the *XBee*, so they can be seen in the computer.

The other main use of the wireless communication is the remote control of the robotic cars. The way to do it is by adding a piece of code to the *Arduino* that reads the characters sent from the computer and reacts differently according to the character or string received. The exact commands to control the robotic cars are explained later when detailing the *Arduino* codes.

In the future, a program could be run in the computer that automatically interacts with the vehicles. This could be very interesting if traffic lights, signals, crossroads, etc. are added.

2.6. Calibrating the Robotic Cars

2.6.1. Distance Measurement with Pixy Cam

In order to estimate the distance to an object with the *Pixy Cam*, the technique used is based on the change in size of the tracked object. The Pixy Cam give values of height and width of the objects it tracks in pixels. Hence, after some calibration specific to its object, one can guess the distance from the value of height or width.

For testing the accuracy, a 5.5x4.5 cm object was used. The object was set at the height of the pixy cam, and a code was run in the robot in order for the camera to point at it and write the values of *height* and *width* through the serial monitor (Annex 1, page 67). The robot was moved in a straight line looking at the object in order to take measurements every 5 cm ranging from a closest distance of 5 cm to a farthest of 50 cm. To make the measurements more stable and easy to read in the serial monitor, the Arduino read the values 100 times and gave the average of them.

After collecting the data, it can be plotted in excel in order to get a tendency of distance against *height* or *width* of the specified object. This equation is implemented in the Arduino code in order for the robot to guess the distance to the object. In a 2 dimensional environment like the one the robot vehicles will be moving on, using the *height* value rather than the *width* seems a better option, as it is less sensible to changes in the angle of view. The code to print the distance measured through the serial monitor is like the one used before but with some differences in the setup and main loop (Annex 1, page 69).

This technique relies a lot on calibration with each specific object that needs to be tracked, but once calibrated has shown to give pretty good accuracy.

For the stated object used for the test, the following graph and tendency lines were obtained.

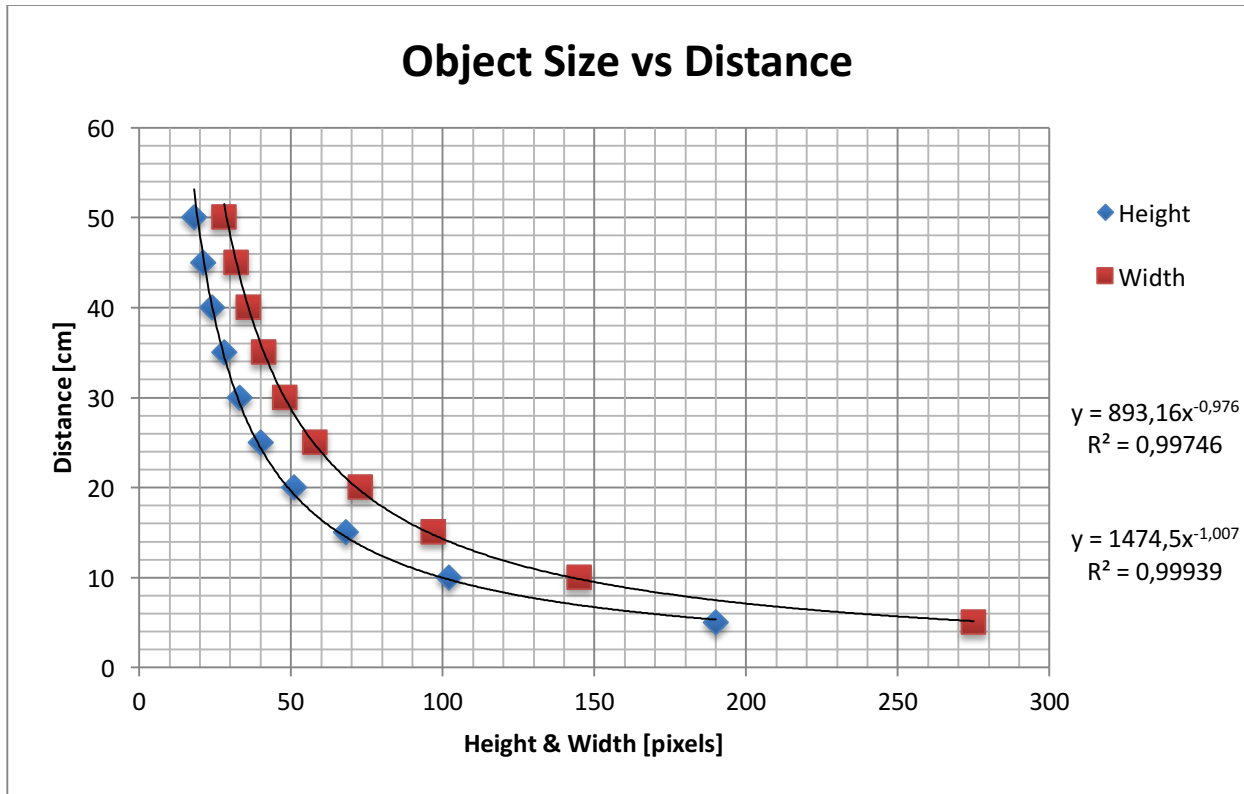


Figure 26. Graph of Distance against Object Size for distance measurement calibration.

2.6.2. Zumo Robot Speed Control

2.6.2.1. Straight speed test

In order to control the speed of the robot, the *ZumoMotors.h* library allows to set a variable between 0 and 400 proportional to the speed of the left or right motor. Hence, to find the equivalence of this variable with the real speed of the robot, a program has been written that drives the robot forward and then backwards for a certain time at a specific speed (Annex 1, page 70). Changing the value of the *Speed* variable and measuring the distances travelled by the robot each time, a graph can be obtained with the relation of *Speed* variable and real speed.

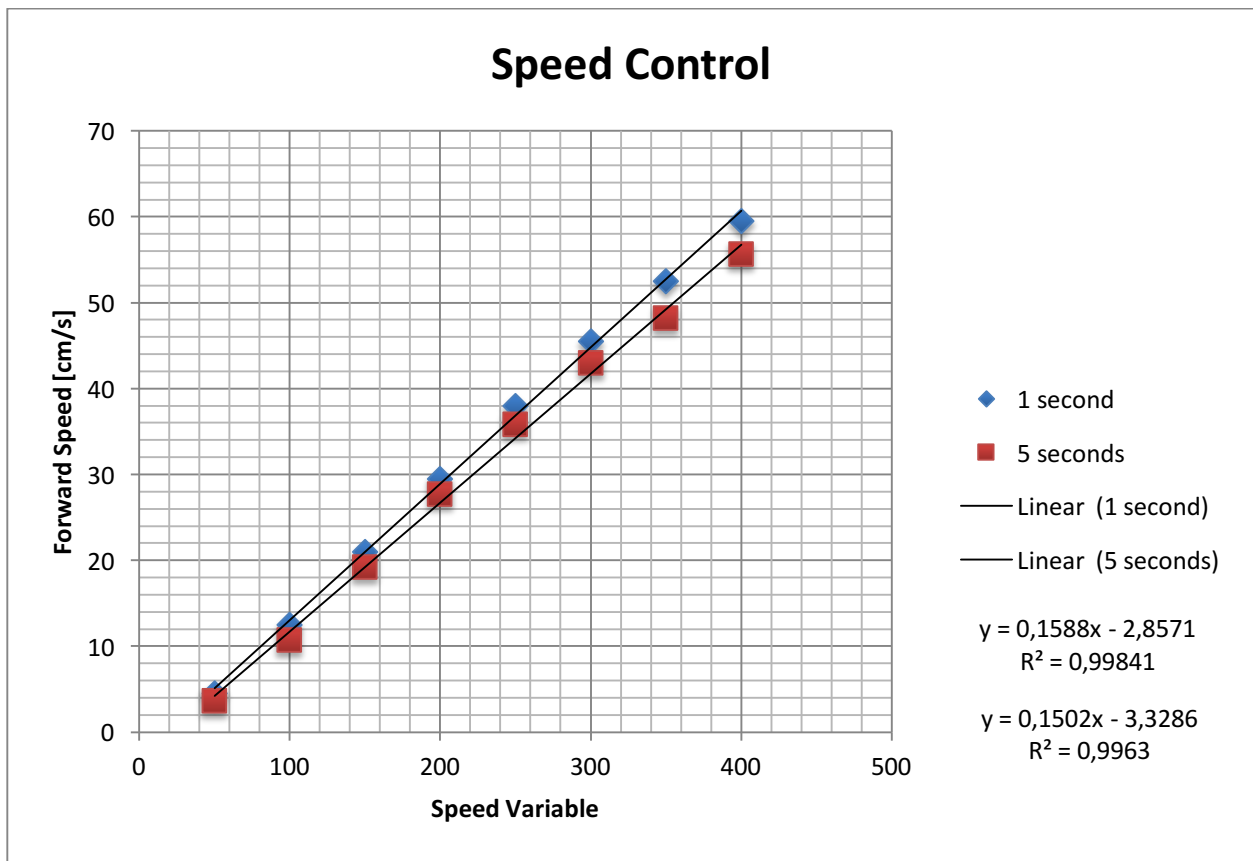


Figure 27. Graph relating real speed of the robotic car going forward with the *Speed* variable.

As it can be seen, there are some differences between travelling for 1 or 5 seconds. These might be in part due to irregularities on the carpet floor where the experiment was carried out. However, the data is pretty consistent with the specifications of the manufacturer.

Some differences could also be found when going forward or backwards, usually the last one being slower. However, there was no plan on driving the robotic cars backwards, so the focus will be on forward speed.

It was also appreciated a tendency of the robot to turn to the left, probably due to some movement of the wheels and roughness on the ground. This behavior was more important when going backwards, but, as said before, there is no interest on moving the robotic cars backwards accurately. However, it was also a bit noticeable when going forward, so in order to compensate it and make the robot go straight, the left motor speed was increased by a value between 3% and 5% respect to the right motor speed. Anyway, as in the circuit the robots will follow a line, this asymmetry will be automatically compensated by the line following algorithm.

Taking into account all the previous data, it seems the speed of the robots is pretty accurately controllable by changing the *Speed* variable in the *ZumoMotors.h* library. However, due to its big dependency on external factors (like roughness of terrain, movement of the wheels, weight of the robot, etc.) it needs specific calibration in conditions as similar as possible to the ones used in each circuit. This means to start by taking the same kind of data from the robot following a line.

2.6.2.2. Line following speed test

To check the speed of the robot while following a line, a setup similar to the previous one was used. The following line algorithm from the *Zumo* robot examples was modified to run during a specified time, stop for 5 seconds and repeat this sequence again and again (Annex 1, page 71). In order to get data for the different speeds, the *MAX_SPEED* variable was modified each time.

A straight black line was drawn and installed over the same carpet floor as the previous tests. After running the robot on it, measuring the distance travelled and analyzing it, the following graph was obtained:

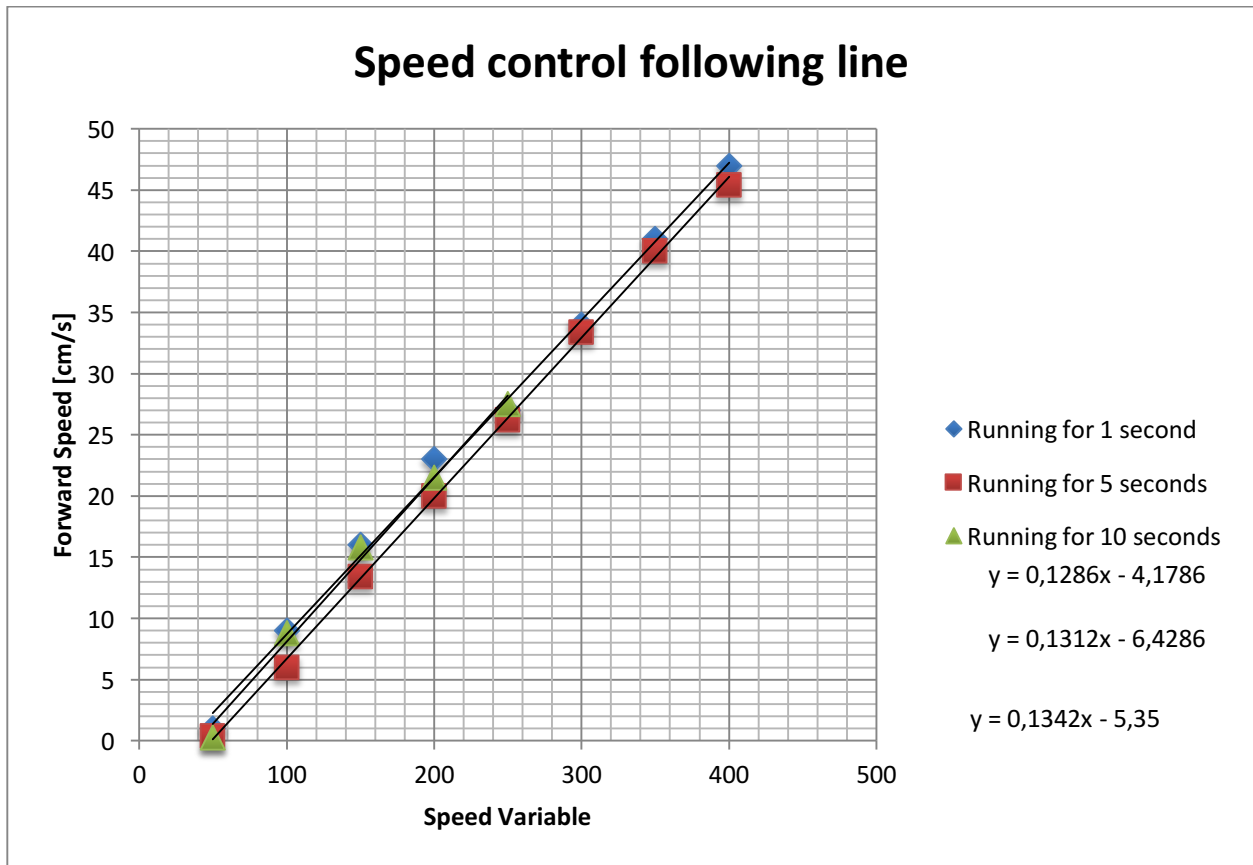


Figure 28. Graph relating the *Speed* variable with the actual speed of the robotic car while following a line.

As it can be seen, the relation between speed variable and actual speed of the robot while following a line is lineal, and seems to be pretty steady regardless the time the robot is running. It can also be seen, as expected, that the robot following a line is a bit slower than going straight, but the speed achieved is pretty good anyway.

The most important parameters affecting the speed of the vehicle seem to be the actual weight of the robot itself and the type of surface is moving on. So that, in order to get the equation to implement in the codes that allows to set a speed to the robot, a similar calibration has to be done for each change in the vehicle design or circuit surface.

2.7. Main functionality: Line following adapting speed to previous vehicle

The basic code that will run the robotic cars through the circuit is based on two main features: **line following** and **object tracking**.

In order for the robotic cars to be controlled and guided through a circuit, the line following feature will be used. This way, any kind of circuit can be drawn with a black line on white background, so the vehicles can follow it. *Pololu* provides a line following code for the *Zumo Arduino Robot* among the example codes in its library.

To adapt the speed according to the distance with the previous vehicle, the *Pixy Cam* object-tracking feature will be used, attaching a colored dot in the back of each robot, so the camera of the approaching robotic car can see it. *Adafruit* provides a code for the *Zumo Robot for Arduino* to track and follow objects using the *Pixy Cam*.

In the specific platform being built, the black lines are meant to represent roads that the robotic cars will keep following, while checking the distance with the previous vehicles using the object-tracking mechanism. To do so, parts of the two basic stated codes had to be merged together with new implementations for the whole system to work. Following, the basic structure for this code is explained:

2.7.1. Declaration of variables

The first part of the code is run once when the robot is started, and declares the libraries and variables used, together with the “Servo Loop Class”, a proportional/derivative feedback loop intended to move the pan and tilt servo of the *Pixy Cam* in order to keep the tracked object always in the center of the view. More information on the exact way this is done can be found on the website of the *Pixy Pet* project by *Adafruit* [12].

```
#include <QTRSensors.h>
#include <ZumoReflectanceSensorArray.h>
#include <ZumoMotors.h>
#include <ZumoBuzzer.h>
#include <Pushbutton.h>

#include <SPI.h>
#include <Pixy.h>

ZumoBuzzer buzzer;
ZumoReflectanceSensorArray reflectanceSensors;
ZumoMotors motors;
Pushbutton button(ZUMO_BUTTON);
int lastError = 0;

// This is the maximum speed the motors will be allowed to turn.
// (400 lets the motors go at top speed; decrease to impose a speed limit)
int MAX_SPEED = 0;

#define X_CENTER      long ((PIXY_MAX_X-PIXY_MIN_X)/2)           // 160L
#define Y_CENTER      long ((PIXY_MAX_Y-PIXY_MIN_Y)/2)           // 100L
#define RCS_MIN_POS   0L
#define RCS_MAX_POS   1000L
#define RCS_CENTER_POS ((RCS_MAX_POS-RCS_MIN_POS)/2)           // Experimentally found to be the
right center position for the servos

//-----
// Servo Loop Class
// A Proportional/Derivative feedback
// loop for pan/tilt servo tracking of
// blocks.
// (Based on Pixy CMUcam5 example code)
//-----
class ServoLoop
{
public:
    ServoLoop(int32_t proportionalGain, int32_t derivativeGain);

    void update(int32_t error);

    int32_t m_pos;
    int32_t m_prevError;
    int32_t m_proportionalGain;
    int32_t m_derivativeGain;
};

// ServoLoop Constructor
ServoLoop::ServoLoop(int32_t proportionalGain, int32_t derivativeGain)
{
    m_pos = RCS_CENTER_POS;
```

```

        m_proportionalGain = proportionalGain;
        m_derivativeGain = derivativeGain;
        m_prevError = 0x80000000L;
    }

// ServoLoop Update
// Calculates new output based on the measured
// error and the current state.
void ServoLoop::update(int32_t error)
{
    long int velocity;
    char buf[32];
    if (m_prevError!=0x80000000)
    {
        velocity = (error*m_proportionalGain + (error - m_prevError)*m_derivativeGain)>>10;

        m_pos += velocity;
        if (m_pos>RCS_MAX_POS)
        {
            m_pos = RCS_MAX_POS;
        }
        else if (m_pos<RCS_MIN_POS)
        {
            m_pos = RCS_MIN_POS;
        }
    }
    m_prevError = error;
}
// End Servo Loop Class
//-----

Pixy pixy; // Declare the camera object

ServoLoop panLoop(200, 200); // Servo loop for pan
ServoLoop tiltLoop(150, 200); // Servo loop for tilt

```

2.7.2. Setup

This is a basic structure of the *Arduino* codes, based in an action called “setup” that is run once at the beginning of the program, in order to prepare the robot for operation. In our case it is used to initialize the reflectance sensors, the serial ports and some more variables. It also runs a piece of code allowing the user to decide via remote control or via pressing the button when to start calibrating the reflectance sensors. Finally, it waits for the button in the robotic car to be pressed or any character to be sent via remote control in order to start the main program:

```
void setup()
{
    // Initialize the reflectance sensors module
    reflectanceSensors.init();

    // Set up both ports at 9600 baud. This value is most important
    // for the XBee. Make sure the baud rate matches the config
    // setting of your XBee.

    Serial1.begin(9600); //XBee/UART1/pins 0 and 1
    Serial.begin(9600); //USB

    // Wait for wireless command or for the user button to be pressed and released
    // button.waitForButton();

    Serial1.println(F("Calibrate? Press 'Y' to calibrate."));
    while (Serial1.available() < 1 && !button.getSingleDebounceRelease())
    ; // Wait for button or
answer to be pressed
    char temp = Serial1.read(); // Read the pressed
key.

    switch (temp) {
        case 'Y':
        case 'y':
            CalibrateLineSensors();
            break;
        default:
            CalibrateLineSensors();
            break;
    }
    pixy.init(); // Initialize Pixy Cam

    // Wait for wireless command or for the user button to be pressed and released
    // button.waitForButton();
    Serial1.println(F("Calibrated!"));
    Serial1.println(F("Press any key to start"));
    while (Serial1.available() < 1 && !button.getSingleDebounceRelease())
    ; // Wait for any key to
be pressed
    temp = Serial1.read(); // Read the pressed key.
}
```

```
uint16_t blocks;
uint32_t lastBlockTime = 0;
uint32_t lastSpeedUpdate = 0;
int32_t BlockHeight = 0;
float DesiredSpeed = 0;          // Starting speed of 21 cm/s (equivalent to around 200 in speed
variable, maximum speed of robot)

int SafetyDistance = 10;        // Minimum Safety Distance with the previous vehicle [cm] (1/3 -
2/3 of vehicle length)
float VarT = 0.1;               // Delay in drivers reaction [s]
float Tau = 0.3;                // Constant of time (time we want to stay to reach the previous
vehicle) [s]
float T = 0.5;                  // Time to reach the desired velocity [s]

float SpeedVariable = 0;
float Distance = 100;           // Distance to previous vehicles [cm]
```

2.7.3. Main loop

The main loop is the most important structure of the Arduino code. It is the set of instructions that will be run repeatedly forever after the setup is run once. In our case, this piece of code is the one in charge of keeping the robotic car on the track while checking the distance and adapting the speed to the previous vehicle.

To track the line, the Arduino reads the values of the reflectance sensors under the robot at each run of the loop, and using the parameters calibrated in the setup part, calculates an error value representing how far the robot is from the center of the line. This error value is then used in a PID controller in order to calculate the speed difference to be applied between right and left motor in order for the robot to go to the center of the line. The Integer term is usually not very useful in line following, so actually a PD controller is implemented. At the moment, the Proportional constant used is $\frac{1}{4}$, and the derivative constant 6, but they can be changed if needed to adapt the response of the robots to each specific track:

```
int speedDifference = error/4 + 6 * (error - lastError)
```

The other important feature implemented in the main loop is the control of the distance with the previous vehicle. This is done by modifying the “Maximum Speed” allowed to be applied at the motors by the line following set of instructions. To do so, the Arduino reads the parameters from the Pixy Cam in order to get the number of tracked objects in sight and the height of the biggest one. From this height, the distance is calculated using the equations found during the calibration of the distance detection. It also runs some auxiliary functions for the servos to point the camera to the biggest tracked object.

Then, it applies the Optimal Velocity Model to calculate the desired speed according to the distance estimated, and it sets this value to the `MAX_SPEED` parameter. This last part is done every certain time only (currently every 0.1 seconds), in order to simulate the real time of reaction of human beings. If no blocks are seen or its height is smaller than a certain threshold, the distance is set to a value big enough for the models to calculate the speed parameters as if there was no vehicle in sight.

It is important to consider the refresh times of the different devices. The *Arduino* will go through the code at the fastest rate in the system (16 MHz). The *Pixy Cam*, however, will send information at a slower rate (around every 10 cycles of the *Arduino* board). So that, it is important to update the speed at an even slower rate, as to make sure there are updated distances based on the readings of the camera. This slower rate is not a problem, as it is still faster than what one can appreciate. Moreover, it allows the line following corrections to be applied at a much higher speed than anything else, so ensuring a smooth movement of the robotic vehicle along the track.

Finally, at the beginning of the code there are also some lines in charge of the remote control of the robotic cars. At every cycle the *Arduino* checks for received characters, and if a letter 's' is typed it stops the vehicle and waits for any other character to be typed to resume the movement. If the character 'r' is pressed, the robotic cars stop and launch the *RemoteControl* action, in order to update some parameters of the vehicle without having to reprogram it.

```
void loop()
{
    // Wireless Communication features: Stop program when letter 's' or 'S' is pressed, stop and
    // enter remote control mode when 'r' or 'R', and listen for communication when getting 'c' or 'C'
    if (Serial1.available() >= 1) {
        char WirelessCommand = Serial1.read();
```

```

switch (WirelessCommand) {
  case 's':
  case 'S':
    motors.setSpeeds(0, 0);
    Serial1.println(F(" Stopped!"));
    Serial1.println(F("Press any key to start again"));
    while (Serial1.available() < 1)
      ; // Wait for a key to be pressed
    WirelessCommand = Serial1.read();
    break;

  case 'r':
  case 'R':
    motors.setSpeeds(0, 0);
    Serial1.println(F(" Stopped!"));
    Serial1.println(F("Entered remote Control mode"));
    RemoteControl();
    break;
}
}
// Follow line and every 'VarT' seconds update MAX_SPEED according to: 1) Object Tracking. 2)
Distance Detection. 3) Optimal Velocity Model

blocks = pixy.getBlocks();
float Speed = 0.13 * MAX_SPEED - 5; // Current
speed [cm/s]

// If we have blocks in sight, track and follow them
if (blocks)
{
  int trackedBlock = TrackBlock(blocks);
  BlockHeight = pixy.blocks[trackedBlock].height;
  lastBlockTime = millis();

  Serial.print("Block Height [pixels] = ");
  Serial.println(BlockHeight);
}
else if (millis() - lastBlockTime > 100) {
  panLoop.m_pos = RCS_CENTER_POS;
  tiltLoop.m_pos = RCS_CENTER_POS;
  pixy.setServos(panLoop.m_pos, tiltLoop.m_pos);
}

Serial.print("Run ");

if (millis() - lastSpeedUpdate > VarT*1000) {
  if (BlockHeight >= 5) {
    Distance = 782.66 * pow(BlockHeight,-0.981); // Distance to previous vehicle
[cm]
  }
  SpeedVariable = 7.5 * DesiredSpeed + 40;
  MAX_SPEED = (int) SpeedVariable;
  if (MAX_SPEED > 300) {
    MAX_SPEED = 300;
  }

  if (Distance < 8) {
    DesiredSpeed = 0;
  }
  else {
    DesiredSpeed = OptimalVelocityModel(Distance, Speed, VarT); // Desired speed [cm/s]
  }

  Print_Values_Serial(BlockHeight, Distance, Speed, DesiredSpeed, MAX_SPEED);
  // Print_Values_Serial1(BlockHeight, Distance, Speed, DesiredSpeed, MAX_SPEED);

  BlockHeight = 0;
  Distance = 100; // What we
consider "Infinite Distance": no vehicle on sight.
  lastSpeedUpdate = millis();
}
}

```



```

unsigned int sensors[6];

// Get the position of the line. Note that we *must* provide the "sensors"
// argument to readLine() here, even though we are not interested in the
// individual sensor readings
int position = reflectanceSensors.readLine(sensors);

// Our "error" is how far we are away from the center of the line, which
// corresponds to position 2500.
int error = position - 2500;

// Get motor speed difference using proportional and derivative PID terms
// (the integral term is generally not very useful for line following).
// Here we are using a proportional constant of 1/4 and a derivative
// constant of 6, which should work decently for many Zumo motor choices.
// You probably want to use trial and error to tune these constants for
// your particular Zumo and line course.
int speedDifference = error / 4 + 6 * (error - lastError);

lastError = error;

// Get individual motor speeds. The sign of speedDifference
// determines if the robot turns left or right.
int m1Speed = MAX_SPEED + speedDifference;
int m2Speed = MAX_SPEED - speedDifference;

// Here we constrain our motor speeds to be between 0 and MAX_SPEED.
// Generally speaking, one motor will always be turning at MAX_SPEED
// and the other will be at MAX_SPEED-|speedDifference| if that is positive,
// else it will be stationary. For some applications, you might want to
// allow the motor speed to go negative so that it can spin in reverse.
if (m1Speed < 0)
    m1Speed = 0;
if (m2Speed < 0)
    m2Speed = 0;
if (m1Speed > MAX_SPEED)
    m1Speed = MAX_SPEED;
if (m2Speed > MAX_SPEED)
    m2Speed = MAX_SPEED;

motors.setSpeeds(m1Speed, m2Speed);
}

```

2.7.4. Auxiliary functions

These are parts of code called from the *main* or *setup* loops for specific purposes. They can be functions or actions and are written at the end of the code, out of the loops. They can be called more than one time in each cycle with different parameters, thus avoiding to write the same lines of code several times in the main structure.

Using these auxiliary functions is the easiest way to implement more functions to the robots in the future, like wireless communication, traffic signs detection, or other logics of behavior.

Currently, the following auxiliary functions are present in the code:

1) *Trackblock* function

It is the piece of code in charge of looking for the biggest block the *Pixy Cam* has in sight, and move the servos for the camera to point at it:

```
int oldX, oldY, oldSignature;

//-----
// Track blocks via the Pixy pan/tilt mech
// (based in part on Pixy CMUcam5 pantilt example)
//-----
int TrackBlock(int blockCount)
{
    int trackedBlock = 0;
    long maxSize = 0;

    Serial.print("blocks =");
    Serial.println(blockCount);

    for (int i = 0; i < blockCount; i++)
    {
        if ((oldSignature == 0) || (pixy.blocks[i].signature == oldSignature))
        {
            long newSize = pixy.blocks[i].height * pixy.blocks[i].width;
            if (newSize > maxSize)
            {
                trackedBlock = i;
                maxSize = newSize;
            }
        }
    }

    int32_t panError = X_CENTER - pixy.blocks[trackedBlock].x;
    int32_t tiltError = pixy.blocks[trackedBlock].y - Y_CENTER;
```

```

    panLoop.update(panError);
    tiltLoop.update(tiltError);

    pixy.setServos(panLoop.m_pos, tiltLoop.m_pos);

    oldX = pixy.blocks[trackedBlock].x;
    oldY = pixy.blocks[trackedBlock].y;
    oldSignature = pixy.blocks[trackedBlock].signature;
    return trackedBlock;
}

```

2) *OptimalVelocityModel* function

It calculates the desired speed for the robot given the distance with the previous vehicle, the current speed, and the time set to represent reaction delay:

```

float OptimalVelocityModel(float d, float v, float VarT)
{
    float Vd = (d - SafetyDistance) / Tau;           // Desired velocity [cm/s]

    float a = (Vd - v) / T;                          // Acceleration to reach desired velocity in
    desired time [cm/s^2]

    float vf = v + a*VarT;                            // Velocity to apply after delay time [cm/s]

    return vf;
}

```

3) *Remote Control* action

Action used to modify parameters of the program without reprogramming it. At the moment it is implemented to change the *Safety Distance*, *VarT*, *Tau* and *T*.

```

void RemoteControl()           // Modify parameters remotely.
{
    while (Serial1.available() < 1)
    ;
    char WirelessCommand = Serial1.read();
    switch (WirelessCommand) {
        case 's':
            of Optimal Velocity Model. // Modify Safety Distance
            case 'S':
                Serial1.println(F("Enter new 'Safety Distance' as ## cm:"));
                while (Serial1.available() < 2)
                ; // Wait for all the expected
                values to be entered
                SafetyDistance = int(Serial1.read()) * 10; // Convert next 2 values
                SafetyDistance += int(Serial1.read()); // into a number.
                break;
    }
}

```

```

        case 'a':
Velocity Model.
        case 'A':
            Serial1.println(F("Enter new delay in drivers reaction 'VarT' as #.# seconds (type only the #
numbers, not the '.'):"));
            while (Serial1.available() < 2)
                ;
            ;
            values to be entered
            VarT = int(Serial1.read());
            VarT += int(Serial1.read()) * 0.1;
            break;
// Modify 'Tau' of Optimal
// Convert next 2 values
// into a number.

        case 'w':
Velocity Model.
        case 'W':
            Serial1.println(F("Enter new 'Tau' parameter for OVM as #.# seconds (type only the # numbers,
not the '.'):"));
            while (Serial1.available() < 2)
                ;
            ;
            values to be entered
            Tau = int(Serial1.read());
            Tau += int(Serial1.read()) * 0.1;
            break;
// Modify 'VarT' of Optimal
// Convert next 2 values
// into a number.

        case 't':
Velocity Model.
        case 'T':
            Serial1.println(F("Enter new 'T' parameter for OVM as #.# seconds (type only the # numbers,
not the '.'):"));
            while (Serial1.available() < 2)
                ;
            ;
            values to be entered
            T = int(Serial1.read());
            T += int(Serial1.read()) * 0.1;
            break;
// Modify 'Tau' of Optimal
// Convert next 2 values
// into a number.
    }
}

```

4) *Print_Values_Serial* action

It prints a series of values through the serial port (usually *USB* cable connection):

```

void Print_Values_Serial (int32_t BlockHeight, float Distance, float Speed, float DesiredSpeed,
int MAX_SPEED)
{
    Serial.print("Block Height [pixels] = ");
    Serial.println(BlockHeight);

    Serial.print("Distance [cm] = ");
    Serial.println(Distance);

    Serial.print("Current Speed [cm/s] = ");
    Serial.println(Speed);

    Serial.print("Desired Speed [cm/s] = ");
    Serial.println(DesiredSpeed);

    Serial.print("Applied Max. Speed variable = ");
    Serial.println(MAX_SPEED);
}

```

5) *Print_Values_Serial1* action

As the previous function, it also prints some values, but this time through the “Serial1” port (usually the *XBee* wireless module):

```
void Print_Values_Serial1 (int32_t BlockHeight, float Distance, float Speed, float DesiredSpeed,
int MAX_SPEED)
{
  Serial1.print("Block Height [pixels] = ");
  Serial1.println(BlockHeight);

  Serial1.print("Distance [cm] = ");
  Serial1.println(Distance);

  Serial1.print("Current Speed [cm/s] = ");
  Serial1.println(Speed);

  Serial1.print("Desired Speed [cm/s] = ");
  Serial1.println(DesiredSpeed);

  Serial1.print("Applied Max. Speed variable = ");
  Serial1.println(MAX_SPEED);
}
```

6) *CalibrateLineSensors* action

Action called from the *setup* loop to calibrate the reflectance sensors by moving the robot from side to side on the black line. This way, the robot can detect the exact values of luminosity of the line and background, and create a good scale to differentiate them:

```
void CalibrateLineSensors()
{
  // Turn on LED and send string to indicate we are in calibration mode
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);
  Serial1.println(F("Calibrating..."));

  // Wait 1 second and then begin automatic sensor calibration
  // by rotating in place to sweep the sensors over the line
  delay(1000);
  int i;
  for(i = 0; i < 80; i++)
  {
    if ((i > 10 && i <= 30) || (i > 50 && i <= 70))
      motors.setSpeeds(-200, 200);
    else
      motors.setSpeeds(200, -200);
    reflectanceSensors.calibrate();

    // Since our counter runs to 80, the total delay will be
    // 80*20 = 1600 ms.
  }
}
```

```
    delay(20);  
  }  
  motors.setSpeeds(0,0);  
  
  // Turn off LED to indicate we are through with calibration  
  digitalWrite(13, LOW);  
  buzzer.play(">g32>>c32");  
}
```

2.8. Robotic cars power autonomy test

A very important thing for the platform is the autonomy and performance of the robotic cars, which run with 4 AA batteries. For such a platform the best option were rechargeable batteries, so there is no need to buy new ones all the time. Two different kinds of batteries were first bought and tried:

- **Amazon Basics rechargeable NiMH AA Batteries (2000 mAh) [13]**
- **Amazon Basics High-Capacity rechargeable NiMH AA Batteries (2400 mAh) [14]**

To test them, two robots with fully charged batteries were put to run at $\frac{3}{4}$ of their top speed (the configuration at which are usually used) while following a line in a ring circuit a bit more than 4 meters long. Every 15 minutes the time they spend to complete a lap was measured and noted, until batteries are no longer able to move them anymore.

It was also interesting to see the effect of the *Pixy Cam* image processing and movement in the battery drain. To get that, the same experiment was repeated but with the robotic cars also measuring distance to the other vehicle in the ring and adapting the speed to it.

After averaging the data obtained from different runs, the graph in Figure 29 could be plotted.

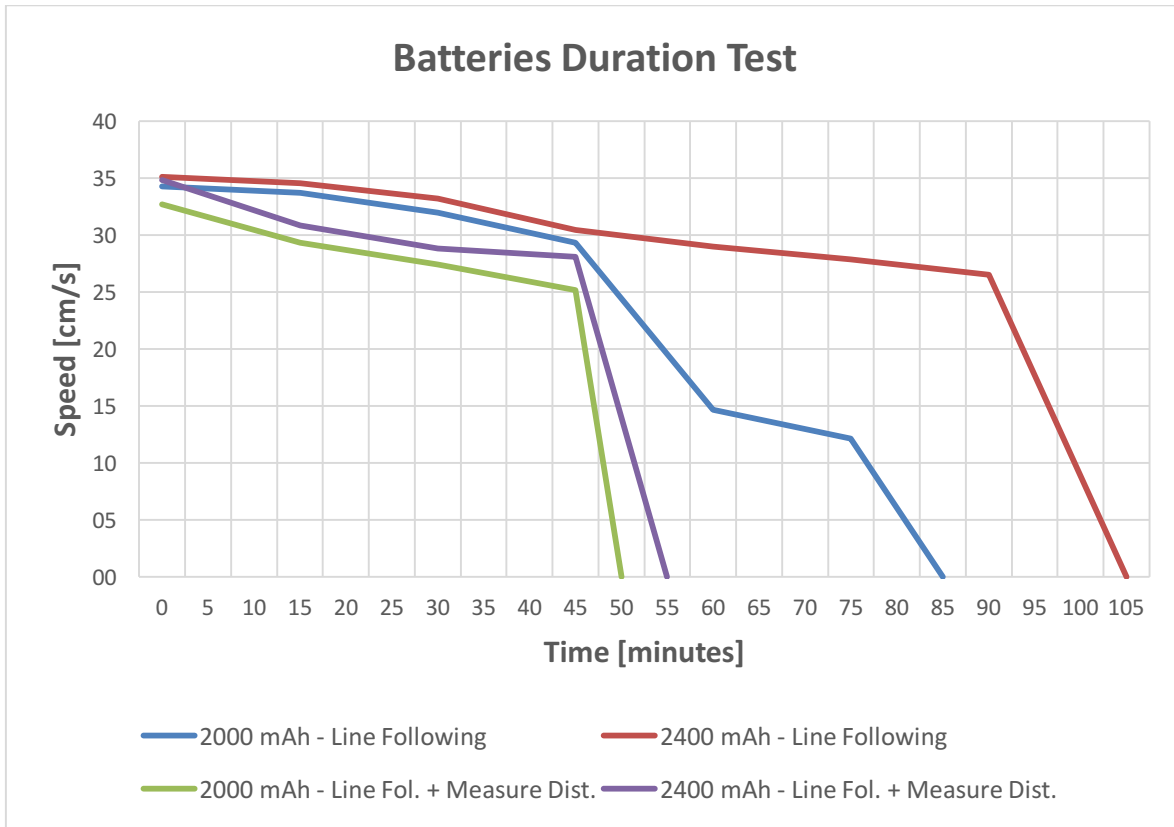


Figure 29. Graph of Robotic Cars autonomy test.

As expected, the 2000 mAh batteries ran out earlier and gave a bit less performance in general than the 2400 mAh ones. It's also very clear from the graph that turning on the *Pixy Cam* reduces significantly the battery life, at about 50% to 60%.

All in all, and taking into account that in real conditions the robotic cars wouldn't be driving always at constant high speed, it can be expected to get an average autonomy of 45' to 1 hour with good performance. It would be also advisable to mount the same kind of batteries fully charged for all the robotic cars used at the same time (and if possible the highest capacity ones), so to reduce the differences in performance between the vehicles to the minimum.

3. TESTING THE ROBOTIC VEHICLES

3.1. Formation of a traffic jam without bottlenecks

A flow of cars in a road can be seen as a non-equilibrium system of interacting particles (the different vehicles). It is commonly thought that traffic jams occur due to a bottleneck somewhere, that forces the vehicle flow to be reduced and so the cars to slow down or stop. However, it has been demonstrated that even without bottlenecks traffic jams can appear when the density of vehicles is bigger than a certain threshold. This happens because even the littlest fluctuation gets amplified, with every car reducing the speed more than the one before, until they start to stop, generating a wave that travels backwards against the flow of traffic.

This behavior of vehicles as multi-particle systems had been mathematically demonstrated and simulated, but a Japanese study managed to recreate it with real cars in 2008. To do it, they used a ring road with 22 vehicles on it, and asked the drivers to try to keep a certain speed while staying on the road and not crashing with the car in front. After a certain time, the traffic jam appeared, as it is explained in the paper they published [15].

The aim of this test is to recreate the Japanese experiment with the robotic cars. To do so, the main code of following a line while adapting the speed to the vehicle they have in front was programmed. In order to calculate the speed, the Optimal Velocity Model was used (as explained before), with different parameters that can be tweaked to adapt the way the robotic cars interact and therefore make the shockwave appear with more or less vehicles.

Results

As explained before, the formation of the shockwave happens after a critical flow is reached, so in the robotic cars setup it was possible to reproduce it under many different circumstances by changing the size of the ring road used, the number of robots, the speed at which they were moving or the parameters of the Optimal Velocity Model.

In the first studies the robots were set with the following parameters:

- *Safety Distance*: 10 cm.
- *VarT*: 0.1 s.
- *Tau*: 0.3 s.
- *T*: 0.5 s.
- *Maximum Speed*: $\frac{3}{4}$ of their top speed.

Under these circumstances, the traffic jam appeared after some laps when there were 5 or more robots in a 44x54 cm oval track (as it can be seen in the video recorded [16]); and 7 or more robots in a 57x77 cm oval track. The shockwave was moving backwards at a speed around 11 and 11.5 cm/s relatively; while the robotic cars were completing each lap at an average speed of 14.5 and 15.5 cm/s. If one robot was taken out, no shockwave was appreciated and the cars completed each lap at average speeds a bit faster than 20 cm/s.

An interesting phenomenon was also observed while carrying out these experiments. When using the critical number of robotic cars in a track, if one of them was a bit slower in average

(either because of batteries running low or by purposely setting it to run at a speed a bit slower than that of the other vehicles), no shockwave was formed (or just a short and unstable one every now and then) and the average speed of all the cars to complete the lap was faster than before [17] [18] [19]. Specifically, in both tracks they reached an average speed of almost 18 cm/s, significantly faster than the one obtained when the traffic jams appeared. This is consistent with the theoretic results, which show that the maximum flow rate is reached just before getting to the critical density of vehicles, with the flow rate and average speed of individual cars reducing drastically after this point.

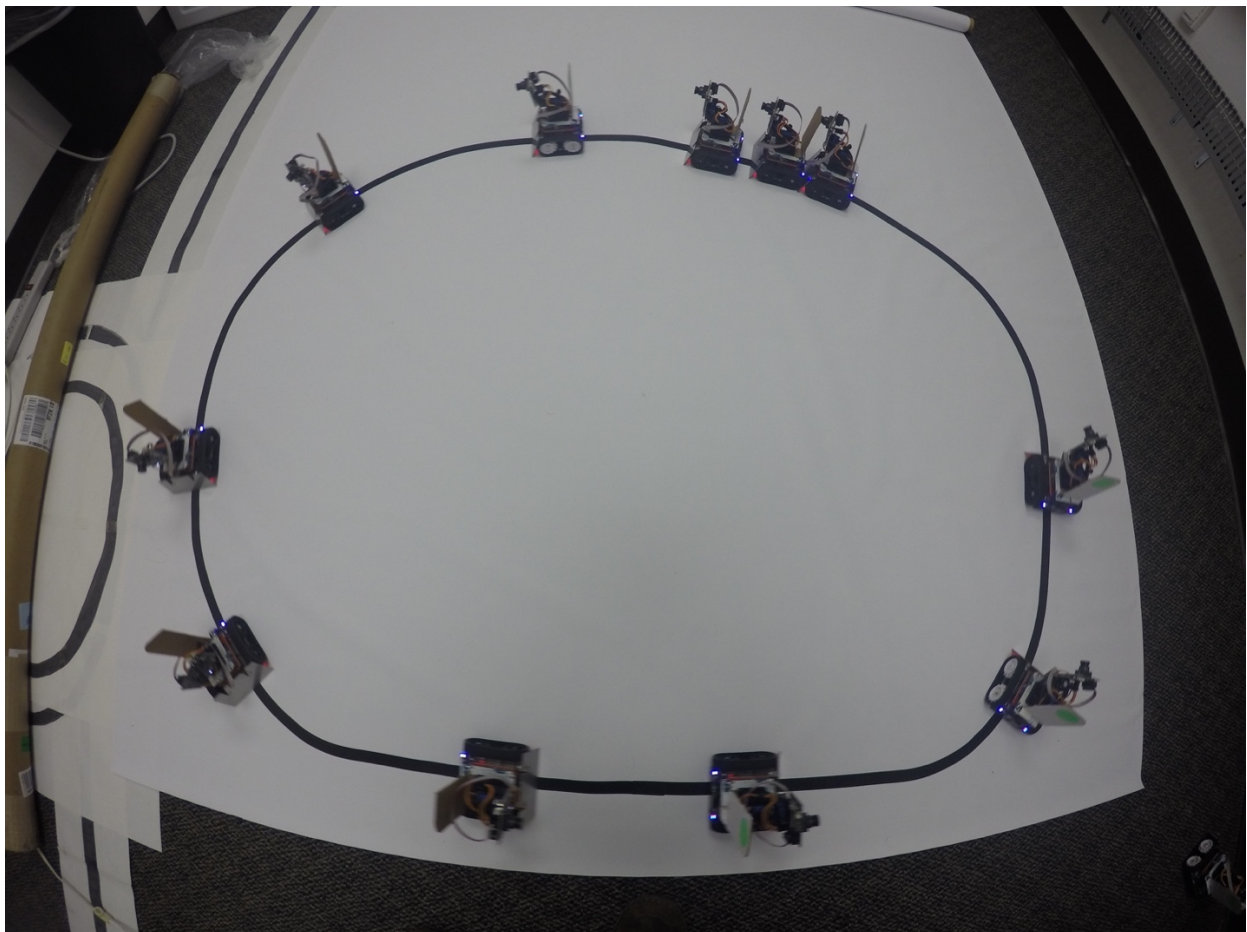


Figure 30. 11 robotic cars driving in a ring road.

Finally, the same effect wanted to be reproduced with the maximum number of robotic cars. 12 vehicles were available at that moment, so by escalating from the previous circuits it was calculated that a 435 cm long circuit was needed. As expected the shockwave appeared with the 12 robotic cars [20], travelling backwards at around 12 cm/s. The vehicles reduced their average speed to complete a lap from 28 cm/s when 11 robotic cars were in the circuit without shockwave [21] to 21 cm/s with 12 vehicles and traffic jam [20].

These results show the possibility to use autonomous vehicles to reduce traffic jams and optimize traffic flow in highways. This could be done by setting leading vehicles at the optimal speed for the level of traffic at each moment.

3.2. Known Issues

3.2.1. Pan/tilt servo problems

The pan/tilt servo where the Pixy Camera is mounted seems to be a bit low in quality. Some of them come not very well centered, so the bottom platform needs to be unscrewed, centered and screwed again in order for the camera to actually point at the center when supposed to do so. It will eventually decenter sometimes itself, too. Moreover, two of them have already broken or got stuck turned to one extreme for example.

Adafruit says it is a cheap and delicate mechanical device so they don't provide warranty for this part if it is more than a month old. So it would be advisable to order 2 or 3 spare ones before a heavy use of the platform is expected in order to replace them fast and easily if needed.

3.2.2. Differences in performance

Sometimes differences in speed and therefore performance among the different robotic cars while running the same code are appreciated. Most of the times these are due to poor battery levels, and can be easily solved by changing these batteries. However, in some specific cases they can obey to factory differences in the motors or other pieces of the shield (like a front blade dragging more friction with the ground). These cases need to be analyzed and solved case by case when they occur.

3.2.3. Auto resetting

It has been experienced that sometimes the software running the *Pixy Cam* or the *Arduino* board can reset or erase by itself. It tends to happen when the batteries are really low, but it has also been seen when the robotics cars are driving normal.

If the problem is in the *Arduino* board, the vehicle will usually stop. If it is in the *Pixy Cam*, it can be detected by the camera doing strange movements going to the extreme positions of the pan/tilt kit and back; or simply by it not detecting any object. In both cases, switching off and on the robotic car might solve the issue; otherwise, the *Arduino* code will need to be uploaded again or the *Pixy Cam* parameters reinstalled through *PixyMon*.

4. CONCLUSIONS AND FUTURE POSSIBILITIES

4.1. Conclusions

To conclude, the basis of a robotic platform for connected and automated vehicles have been set. The platform currently has 14 functional robotic cars, capable of navigating through circuits by following a black line in a white ground, measure the distance to the vehicle in front of them, and control their speed up to 50 cm/s. They can also be remotely controlled from a computer, so implementing V2I communication.

To create the different circuits, the platform features a big white foldable vinyl and black vinyl tape, so it can be rolled and deployed in different places. Finally, there are also 8 packs of 4 2800 mAh rechargeable batteries [22], 6 packs of 4 Amazon 2400 mAh rechargeable batteries [14], and 8 packs of 4 Amazon 2000 mAh rechargeable batteries [13].

4.2. Future possibilities

Due to the good capabilities of the components chosen for the platform, many things can be implemented in the future.

In a first step, crossroads and traffic lights could be set up, detected by the *Pixy Cam* through color codes, and getting information from them wirelessly via *XBee*. Line changing could be also studied, with detection of the road splitting with the reflectance sensors, for example.

Another idea of a future capability is setting up some overhead cameras to get indoor positioning of the robotic vehicles. However, these requires of a more fix setup, as the cameras need to be calibrated for the exact position of the circuit.

Finally, V2V communication could also be implemented if the firmware of the *XBees* is changed to *ZigBee*, allowing to configure the different modules to talk between them in a network mesh configuration.

5. ANNEXES

5.1. Annex 1: Codes used

5.1.1. Distance Measurement with Pixy Cam

```
#include <SPI.h>
#include <Pixy.h>

#include <ZumoMotors.h>

#define X_CENTER    160L
#define Y_CENTER    100L
#define RCS_MIN_POS    0L
#define RCS_MAX_POS    1000L
#define RCS_CENTER_POS ((RCS_MAX_POS-RCS_MIN_POS)/2)

//-----
// Servo Loop Class
// A Proportional/Derivative feedback
// loop for pan/tilt servo tracking of
// blocks.
// (Based on Pixy CMUcam5 example code)
//-----
class ServoLoop
{
public:
    ServoLoop(int32_t proportionalGain, int32_t derivativeGain);

    void update(int32_t error);

    int32_t m_pos;
    int32_t m_prevError;
    int32_t m_proportionalGain;
    int32_t m_derivativeGain;
};

// ServoLoop Constructor
ServoLoop::ServoLoop(int32_t proportionalGain, int32_t derivativeGain)
{
    m_pos = RCS_CENTER_POS;
    m_proportionalGain = proportionalGain;
    m_derivativeGain = derivativeGain;
    m_prevError = 0x80000000L;
}

// ServoLoop Update
// Calculates new output based on the measured
// error and the current state.
void ServoLoop::update(int32_t error)
{
    long int velocity;
    char buf[32];
    if (m_prevError!=0x80000000)
    {
        velocity = (error*m_proportionalGain + (error - m_prevError)*m_derivativeGain)>>10;

        m_pos += velocity;
        if (m_pos>RCS_MAX_POS)
        {
            m_pos = RCS_MAX_POS;
        }
        else if (m_pos<RCS_MIN_POS)
        {
            m_pos = RCS_MIN_POS;
        }
    }
}
```

```

    }
    m_prevError = error;
}
// End Servo Loop Class
//-----

Pixy pixy; // Declare the camera object

ServoLoop panLoop(200, 200); // Servo loop for pan
ServoLoop tiltLoop(150, 200); // Servo loop for tilt

ZumoMotors motors; // declare the motors on the zumo

//-----
// Setup - runs once at startup
//-----
void setup()
{
    Serial.begin(9600);
    Serial.print("Starting...\n");

    pixy.init();
}

uint32_t lastBlockTime = 0;
long i = 1;
long BlockHeight = 0;
long BlockWidth = 0;
long BLockSize = 0;

//-----
// Main loop - runs continuously after setup
//-----

void loop()
{
    uint16_t blocks;
    blocks = pixy.getBlocks();

    // If we have blocks in sight, track and print height and width
    if (blocks)
    {
        int trackedBlock = TrackBlock(blocks);
        BlockHeight += pixy.blocks[trackedBlock].height;
        BlockWidth += pixy.blocks[trackedBlock].width;
        BLockSize += BlockHeight * BlockWidth;
        i++;

        // Print average of last i measures of Height, width and size of object
        if (i == 100)
        {
            Serial.print("Height = ");
            Serial.println(BlockHeight/i);

            Serial.print("Width = ");
            Serial.println(BlockWidth/i);

            BlockHeight = 0;
            BlockWidth = 0;
            BLockSize = 0;
            i = 0;
        }
    }
}

int oldX, oldY, oldSignature;

//-----
// Track blocks via the Pixy pan/tilt mech
// (based in part on Pixy CMUcam5 pantilt example)
//-----
int TrackBlock(int blockCount)
{

```

```

int trackedBlock = 0;
long maxSize = 0;

for (int i = 0; i < blockCount; i++)
{
    if ((oldSignature == 0) || (pixy.blocks[i].signature == oldSignature))
    {
        long newSize = pixy.blocks[i].height * pixy.blocks[i].width;
        if (newSize > maxSize)
        {
            trackedBlock = i;
            maxSize = newSize;
        }
    }
}

int32_t panError = X_CENTER - pixy.blocks[trackedBlock].x;
int32_t tiltError = pixy.blocks[trackedBlock].y - Y_CENTER;

panLoop.update(panError);
tiltLoop.update(tiltError);

pixy.setServos(panLoop.m_pos, tiltLoop.m_pos);

oldX = pixy.blocks[trackedBlock].x;
oldY = pixy.blocks[trackedBlock].y;
oldSignature = pixy.blocks[trackedBlock].signature;
return trackedBlock;

```

5.1.2. Code to print distance measured through serial monitor

```

//-----
// Setup - runs once at startup
//-----
void setup()
{
    Serial.begin(9600);
    Serial.print("Starting...\n");

    pixy.init();
}

uint32_t lastBlockTime = 0;
long i = 1;
long BlockHeight = 0;
long BlockWidth = 0;
long DistanceToBlock = 0;

//-----
// Main loop - runs continuously after setup
//-----

void loop()
{
    uint16_t blocks;
    blocks = pixy.getBlocks();

    // If we have blocks in sight, track and print height and width
    if (blocks)
    {
        int trackedBlock = TrackBlock(blocks);
        BlockHeight += pixy.blocks[trackedBlock].height;
        BlockWidth += pixy.blocks[trackedBlock].width;
        i++;

        // Print average of last i measures of Height, width and size of object

```

```

        if (i == 100)
        {
            DistanceToBlock = 893.16 * pow(BlockHeight/i,-0.976);

            Serial.print("Height = ");
            Serial.println(BlockHeight/i);

            Serial.print("Width = ");
            Serial.println(BlockWidth/i);

            Serial.print("Distance = ");
            Serial.println(DistanceToBlock);

            BlockHeight = 0;
            BlockWidth = 0;
            i = 0;
        }
    }
}

```

5.1.3. Code for straight speed calibration test

```

#include <ZumoMotors.h>

ZumoMotors motors;

void setup() {
    // Stop 1 second

    motors.setLeftSpeed(0);
    motors.setRightSpeed(0);

    delay(1000);
}

int speed = 200;
float straight_correction = 1.05;
int running_time = 1000;

void loop() {

    // Move
    motors.setLeftSpeed(straight_correction * speed);
    motors.setRightSpeed(speed);

    delay(running_time);

    // Stop

    motors.setLeftSpeed(0);
    motors.setRightSpeed(0);

    delay(5000);

    speed = -speed;
}

```

5.1.4. Code for speed calibration while following a line

```
#include <QTRSensors.h>
#include <ZumoReflectanceSensorArray.h>
#include <ZumoMotors.h>
#include <ZumoBuzzer.h>
#include <Pushbutton.h>

ZumoBuzzer buzzer;
ZumoReflectanceSensorArray reflectanceSensors;
ZumoMotors motors;
Pushbutton button(ZUMO_BUTTON);
int lastError = 0;

// This is the maximum speed the motors will be allowed to turn.
// (400 lets the motors go at top speed; decrease to impose a speed limit)
const int MAX_SPEED = 50;

unsigned long run_time = 1000;
unsigned long init_time = millis();

void setup()
{
  // Play a little welcome song
  buzzer.play(">g32>>c32");

  // Initialize the reflectance sensors module
  reflectanceSensors.init();

  // Wait for the user button to be pressed and released
  button.waitForButton();

  // Turn on LED to indicate we are in calibration mode
  pinMode(13, OUTPUT);
  digitalWrite(13, HIGH);

  // Wait 1 second and then begin automatic sensor calibration
  // by rotating in place to sweep the sensors over the line
  delay(1000);
  int i;
  for(i = 0; i < 80; i++)
  {
    if ((i > 10 && i <= 30) || (i > 50 && i <= 70))
      motors.setSpeeds(-200, 200);
    else
      motors.setSpeeds(200, -200);
    reflectanceSensors.calibrate();

    // Since our counter runs to 80, the total delay will be
    // 80*20 = 1600 ms.
    delay(20);
  }
  motors.setSpeeds(0,0);

  // Turn off LED to indicate we are through with calibration
  digitalWrite(13, LOW);
  buzzer.play(">g32>>c32");

  // Wait for the user button to be pressed and released
  button.waitForButton();

  // Play music and wait for it to finish before we start driving.
  buzzer.play("L16 cdeg4");
  while(buzzer.isPlaying());

  unsigned long init_time = millis();
}
```

```

void loop()
{
  unsigned int sensors[6];

  // Get the position of the line. Note that we *must* provide the "sensors"
  // argument to readLine() here, even though we are not interested in the
  // individual sensor readings
  int position = reflectanceSensors.readLine(sensors);

  // Our "error" is how far we are away from the center of the line, which
  // corresponds to position 2500.
  int error = position - 2500;

  // Get motor speed difference using proportional and derivative PID terms
  // (the integral term is generally not very useful for line following).
  // Here we are using a proportional constant of 1/4 and a derivative
  // constant of 6, which should work decently for many Zumo motor choices.
  // You probably want to use trial and error to tune these constants for
  // your particular Zumo and line course.
  int speedDifference = error / 4 + 6 * (error - lastError);

  lastError = error;

  // Get individual motor speeds. The sign of speedDifference
  // determines if the robot turns left or right.
  int m1Speed = MAX_SPEED + speedDifference;
  int m2Speed = MAX_SPEED - speedDifference;

  // Here we constrain our motor speeds to be between 0 and MAX_SPEED.
  // Generally speaking, one motor will always be turning at MAX_SPEED
  // and the other will be at MAX_SPEED-|speedDifference| if that is positive,
  // else it will be stationary. For some applications, you might want to
  // allow the motor speed to go negative so that it can spin in reverse.
  if (m1Speed < 0)
    m1Speed = 0;
  if (m2Speed < 0)
    m2Speed = 0;
  if (m1Speed > MAX_SPEED)
    m1Speed = MAX_SPEED;
  if (m2Speed > MAX_SPEED)
    m2Speed = MAX_SPEED;

  motors.setSpeeds(m1Speed, m2Speed);

  unsigned long current_time = millis();
  unsigned long elapsed_time = current_time - init_time;

  Serial.println(elapsed_time);

  if (elapsed_time > run_time) {
    motors.setSpeeds(0, 0);
    delay(5000);
    init_time = millis();
  }
}

```

5.2. Annex 2: Purchase orders

5.2.1. 1st order (3 robots)

Robot and wireless modules:

- 2x **Zumo robot for Arduino + Arduino Leonardo + Pixy CMUcam5 sensor + Mini**

Pan-tilt kit – assembled with microsensors:

<https://www.adafruit.com/product/1639>

<https://www.adafruit.com/products/849>

<https://www.adafruit.com/product/1906>

<https://www.adafruit.com/products/1967>

- 3x **XBee Series 1 module:**

<http://www.amazon.com/XBee-1mW-Wire-Antenna-802-15->

[4/dp/B004G4ZHK4/ref=sr_1_5?ie=UTF8&qid=1432955640&sr=8-5&keywords=xbee](http://www.amazon.com/XBee-1mW-Wire-Antenna-802-15-4/dp/B004G4ZHK4/ref=sr_1_5?ie=UTF8&qid=1432955640&sr=8-5&keywords=xbee)

- 2x **Sparkfun XBee Shield + Arduino Stackable Header Kit – R3:**

<http://www.amazon.com/SparkFun-XBee->

[Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-](http://www.amazon.com/SparkFun-XBee-Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-)

[3&keywords=xbee+shield](http://www.amazon.com/SparkFun-XBee-Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-3&keywords=xbee+shield)

http://www.amazon.com/Arduino-Stackable-Header-Kit-R3/dp/B00PCCWEJG/ref=sr_1_1?ie=UTF8&qid=1432955480&sr=8-1&keywords=arduino+stackable+header+kit

- **1x Sparkfun XBee Explorer USB:**

http://www.amazon.com/SparkFun-5030-XBEE-Explorer-USB/dp/B008092TZS/ref=sr_1_1?ie=UTF8&qid=1432955493&sr=8-1&keywords=xbee+explorer

Accessories for the robots:

- **Rechargeable NiMH AA Batteries (1.2V, 2200 mAh, 1 cell):**

http://www.amazon.com/Exell-Size-Rechargeable-Battery-2200mAh/dp/B00YHVAAQC/ref=sr_1_5?ie=UTF8&qid=1432955238&sr=8-5&keywords=rechargeable+aa+batteries+2200+1.2

- **1x IMAX B6AC V2 Balance Charger and Discharger:**

http://www.amazon.com/Genuine-SKYRC-Power-6Amps-50Watts/dp/B00ND7J38C/ref=sr_1_1?ie=UTF8&qid=1432954962&sr=8-1&keywords=IMAX+B6AC

- **1x Cable ties:**

http://www.amazon.com/Joy-Fish-Heavy-Nylon-Cable/dp/B005OK7P3G/ref=sr_1_2?ie=UTF8&qid=1432954212&sr=8-2&keywords=cable+ties

- 1x **Double sided foam tape:**

http://www.amazon.com/3M-Scotch-Mounting-5-Inch-75-Inch/dp/B00004Z498/ref=sr_1_2?ie=UTF8&qid=1432954329&sr=8-2&keywords=double+sided+foam+tape

Tools:

- 1x **Soldering Kit + Soldering wire:**

http://www.amazon.com/Weller-WLC100-40-Watt-Soldering-Station/dp/B000AS28UC/ref=sr_1_3?ie=UTF8&qid=1432954587&sr=8-3&keywords=soldering

http://www.amazon.com/Kester-Pocket-Pack-Solder-0-031/dp/B00068IJNQ/ref=pd_bxgy_469_text_z

- 1x **Wire cutters:**

http://www.amazon.com/Hakko-CHP-170-Stand-off-Construction-21-Degree/dp/B00FZPDG1K/ref=sr_1_1?ie=UTF8&qid=1432954819&sr=8-1&keywords=wire+cutter

- **1x USB A to Micro-B cable:**

http://www.amazon.com/Mediabridge-USB-2-0-Micro-USB-High-Speed/dp/B004GF8TIK/ref=sr_1_3?ie=UTF8&qid=1432954884&sr=8-3&keywords=micro+usb

5.2.2. 2nd order (12 more robots)

- 12x (**Zumo robot for Arduino + Arduino Leonardo + Pixy CMUcam5 sensor + Mini Pan-tilt kit – assembled with microsensors**):

<https://www.adafruit.com/product/1639>

<https://www.sparkfun.com/products/11286>

<https://www.adafruit.com/product/1906>

<https://www.adafruit.com/products/1967>

- 13x (**XBee Series 1 module**):

[http://www.amazon.com/XBee-1mW-Wire-Antenna-802-15-](http://www.amazon.com/XBee-1mW-Wire-Antenna-802-15-4/dp/B004G4ZHK4/ref=sr_1_5?ie=UTF8&qid=1432955640&sr=8-5&keywords=xbee)

[4/dp/B004G4ZHK4/ref=sr_1_5?ie=UTF8&qid=1432955640&sr=8-5&keywords=xbee](http://www.amazon.com/XBee-1mW-Wire-Antenna-802-15-4/dp/B004G4ZHK4/ref=sr_1_5?ie=UTF8&qid=1432955640&sr=8-5&keywords=xbee)

- 13x (**Sparkfun XBee Shield + Arduino Stackable Header Kit – R3**):

[http://www.amazon.com/SparkFun-XBee-](http://www.amazon.com/SparkFun-XBee-Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-3&keywords=xbee+shield)

[Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-](http://www.amazon.com/SparkFun-XBee-Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-3&keywords=xbee+shield)

[3&keywords=xbee+shield](http://www.amazon.com/SparkFun-XBee-Shield/dp/B004GTQBA0/ref=sr_1_3?ie=UTF8&qid=1432955436&sr=8-3&keywords=xbee+shield)

[http://www.amazon.com/Arduino-Stackable-Header-Kit-](http://www.amazon.com/Arduino-Stackable-Header-Kit-R3/dp/B00PCCWEJG/ref=sr_1_1?ie=UTF8&qid=1432955480&sr=8-1&keywords=arduino+stackable+header+kit)

[R3/dp/B00PCCWEJG/ref=sr_1_1?ie=UTF8&qid=1432955480&sr=8-](http://www.amazon.com/Arduino-Stackable-Header-Kit-R3/dp/B00PCCWEJG/ref=sr_1_1?ie=UTF8&qid=1432955480&sr=8-1&keywords=arduino+stackable+header+kit)

[1&keywords=arduino+stackable+header+kit](http://www.amazon.com/Arduino-Stackable-Header-Kit-R3/dp/B00PCCWEJG/ref=sr_1_1?ie=UTF8&qid=1432955480&sr=8-1&keywords=arduino+stackable+header+kit)

- 2x (**AmazonBasics rechargeable NiMH AA Batteries (16 pack)**):

http://www.amazon.com/AmazonBasics-AA-Rechargeable-Batteries-16-Pack/dp/B007B9NV8Q/ref=sr_1_1?s=hpc&ie=UTF8&qid=1439411027&sr=1-1&keywords=rechargeable+batteries

- 2x (**AmazonBasics High-Capacity rechargeable NiMH AA Batteries (8 pack)**):

http://www.amazon.com/AmazonBasics-High-Capacity-Rechargeable-Batteries-Pre-charged/dp/B00HZV9WTM/ref=sr_1_3?s=hpc&ie=UTF8&qid=1439411027&sr=1-3&keywords=rechargeable+batteries

- 1x (**Battery charger**):

http://www.amazon.com/EBL%C2%AE-Version-Charger-Rechargeable-Batteries/dp/B00EB7812C/ref=sr_1_2?s=hpc&ie=UTF8&qid=1439411027&sr=1-2&keywords=rechargeable+batteries

- 1x (**AC cable**):

http://www.amazon.com/Cables-Unlimted-6-feet-Mickey-Mouse/dp/B000234TYI/ref=sr_1_4?ie=UTF8&qid=1439412633&sr=8-4&keywords=ac+cable

5.2.3. 3rd order

- 1x **CowboyStudio 6 x 9 Feet Seamless White Vinyl Background (VL-W9):**

http://www.amazon.com/CowboyStudio-Seamless-White-Background-VL-W9/dp/B005SSMSMM/ref=sr_1_12?ie=UTF8&qid=1445625762&sr=8-12&keywords=white+vinyl+roll

- 1x **3M 88 Electrical Tape, .75-Inch by 66-Foot by .0085-Inch:**

http://www.amazon.com/3M-Electrical-75-Inch-66-Foot-0085-Inch/dp/B00004WCCP/ref=sr_1_3?ie=UTF8&qid=1445626148&sr=8-3&keywords=black+tape

- 2x **3M Heavy Duty Mounting Tape, 1-Inch by 50-Inch:**

http://www.amazon.com/3M-Heavy-Mounting-1-Inch-50-Inch/dp/B00004Z4A8/ref=sr_1_2?ie=UTF8&qid=1445626133&sr=8-2&keywords=double+sided+tape

- 1x **AmazonBasics AA High-Capacity Rechargeable Batteries (8-Pack) Pre-charged:**

http://www.amazon.com/AmazonBasics-High-Capacity-Rechargeable-Batteries-Pre-charged/dp/B00HZV9WTM/ref=sr_1_2?ie=UTF8&qid=1445639652&sr=8-2&keywords=rechargeable+batteries

5.2.4. 4th order

- **2x EBL 8 Bay AA, AAA, Ni-MH, Ni-Cd Rechargeable Battery Charger:**

http://www.amazon.com/EBL%C2%AE-Version-Charger-Rechargeable-Batteries/dp/B00EB7812C/ref=sr_1_2?s=hpc&ie=UTF8&qid=1439411027&sr=1-2&keywords=rechargeable+batteries

- **2x EBL 16 Pack 2800mAh High Capacity Rechargeable AA Batteries with Charger:**

http://www.amazon.com/EBL%C2%AE-Version-Charger-Rechargeable-Batteries/dp/B00EB7812C/ref=sr_1_2?s=hpc&ie=UTF8&qid=1439411027&sr=1-2&keywords=rechargeable+batteries

6. REFERENCES

- [1] [Online]. Available: <http://www.arduino.cc/en/Main/Robot> .
- [2] [Online]. Available: <https://www.parallax.com/product/32335> .
- [3] [Online]. Available: <https://www.pololu.com/product/2510> .
- [4] [Online]. Available: <http://www.makershed.com/products/make-rovera-2wd-arduino-robot-kit> .
- [5] [Online]. Available:
http://www.dfrobot.com/index.php?route=product/product&path=37&product_id=1092 .
- [6] [Online]. Available: <http://charmedlabs.com/default/pixy-cmucam5/>.
- [7] [Online]. Available: <https://www.sparkfun.com/products/12847> .
- [8] [Online]. Available: <https://www.sparkfun.com/products/11417>.
- [9] [Online]. Available: <https://learn.adafruit.com/pixy-pet-robot-color-vision-follower-using-pixycam/overview>.
- [10] [Online]. Available: <https://learn.sparkfun.com/tutorials/exploring-xbees-and-xctu>.
- [11] [Online]. Available: http://cmucam.org/projects/cmucam5/wiki/Teach_Pixy_an_object.
- [12] [Online]. Available: <https://learn.adafruit.com/pixy-pet-robot-color-vision-follower-using-pixycam/pixy-pet-code>.
- [13] [Online]. Available: http://www.amazon.com/AmazonBasics-AA-Rechargeable-Batteries-16-Pack/dp/B007B9NV8Q/ref=sr_1_1?s=hpc&ie=UTF8&qid=1439411027&sr=1-1&keywords=rechargeable+batteries.
- [14] [Online]. Available: http://www.amazon.com/AmazonBasics-High-Capacity-Rechargeable-Batteries-Pre-charged/dp/B00HZV9WTM/ref=sr_1_3?s=hpc&ie=UTF8&qid=1439411027&sr=1-3&keywords=rechargeable+batteries.
- [15] Y. Sugiyama, M. Fukui, M. Kikuchi, K. Hasebe, A. Nakayama, K. Nishinari, S.-i. Tadaki and S. Yukawa, "Traffic jams without bottlenecks - experimental evidence for the physical mechanism of the formation of a jam," *New Journal of Physics*, vol. 10, 2008.
- [16] [Online]. Available: <https://www.youtube.com/watch?v=U0RXiLUeGgM>.
- [17] [Online]. Available: <https://www.youtube.com/watch?v=CwFeIE5FmQ0>.
- [18] [Online]. Available: <https://www.youtube.com/watch?v=SKdfm1TJKBk>.

[19] [Online]. Available: <https://www.youtube.com/watch?v=QTcw5Lj5DF4>.

[20] [Online]. Available: https://www.youtube.com/watch?v=SfsOIB_ShfY&feature=youtu.be.

[21] [Online]. Available: <https://www.youtube.com/watch?v=GK3bpEDKoZs>.

[22] [Online]. Available: http://www.amazon.com/EBL%C2%AE-Version-Charger-Rechargeable-Batteries/dp/B00EB7812C/ref=sr_1_2?s=hpc&ie=UTF8&qid=1439411027&sr=1-2&keywords=rechargeable+batteries.