End of Degree Project

Bachelor's degree in Industrial Technology Engineering

Control simulation of a line tracker vehicle using Gazebo

Author: Joel Lloses Miró
Director: Arnau Dòria Cerezo
Jan Rosell Gratacós

Call: June 2017

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Summary

The process of designing the control strategy for mobile robots may turn into a difficult task as can appear difficult dynamisms on a mechanical an electronic level that are difficult to model and, besides, its testing can mean a lot of time and high costs. In this project, it is introduced a simulator environment based on Gazebo to fulfil the need of implementing robot behaviours and control algorithms in a more efficient and productive way.

Gazebo is one amongst several software platforms for simulation and robot control. It has an open source program distributed by Apache 2.0 and is being used widely by the robotics community. It has an easy-going interface, a learn-by-doing program and a growing community.

Its easy programming interface has been very useful for most of the project development, which has been focused on the implementation of a line follower robot developed by other ETSEIB student within their Final Grade Project. The challenge has been to reproduce in a virtual level the vehicle navigation and performance and the environment conditions. For this reason, has been required a global vision of the robot's functionality.

This project is organized with a first part of Gazebo software description, followed by the robot implementation to the simulator, the control design and architecture, and a final part with the experimental results of the simulation.

Index

1. Preface

This project is born with the will to incorporate a new research branch to the developing of the communicate vehicle control algorithms carried out in the Institute of Industrial and Control Engineering in ETSEIB, and it aims to be the first step to an accurate simulation environment to be used as a tool for control strategies development in realistic scenarios.

1.1. Project origin

The origin of the project comes from the will to study and apply the control techniques to different scenarios of communicated vehicles such as: platooning, autonomous vehicle, overtaking manoeuvres, vehicle lane changes or shockwave traffic jam. It is a field of research to create networks in which vehicles and roadside units act as communicating nodes, providing each other with information.

1.2. Previous requirements

In this project, it is recommended to get to know the basic of C++ language, and to work with an object-oriented programming interface which is used to get access to the simulator's code.

2. Introduction

2.1. Objectives

The purpose of the Project is to create a realistic simulation environment to develop and test control algorithms for a path follower robot. The work will be focused on reproducing the vehicles path tracking performance, therefore the elements that have influence on the robot's dynamic behaviour, such as sensor, actuators, amongst others, will be implemented. In the end of the project development, the simulation and experimental results should agree very well to prove the usability of the developed environment.

3. Gazebo Simulation Environment

3.1. What is Gazebo?

Gazebo started in 2002 as an initiative of a professor, Andrew Howard, and his student, Nate Koenig, within the developing of his PhD. The goal was to fulfil the need of simulating robots in outdoor environments and under various conditions with a high-fidelity simulator. The project was required as a complementary simulator to the 2D Stage indoor simulator. The basic difference between an indoor and outdoor environment is their space boundaries, while the first one is a closest system, the second is not. Gazebo took part in Player project, as well as Stage, from 2004 through 2011. In 2009, ROS (Robotics Operating System), which is the one of the most relevant platforms for robotics software development, was integrated into Gazebo and became from since one of the primary tool used in the ROS community. In 2011, OSRF (Open Source Robotics Foundation) provided financial support and Gazebo became and independent project.

Fig 3.1.1. Gazebo logotype

From its very beginning to nowadays, it has evolved greatly and one of the reasons is its open-source basis, which means that its source code is available and can be developed in a collaborative public manner, which is more efficient and economic. Nowadays, Gazebo can be described as a 3D simulator which is able to rapidly and accurately test algorithms and design robots using realistic environment (indoor and outdoor).

3.2. Robotics simulation

Gazebo is far from being the only choice for 3D dynamics simulator. Nowadays, there are a wide range of simulators for basically two purposes: robotics research or industrial simulation. Most of them are developed by robotics companies or institutes and offered as a commercial product, in some cases case they may have a free educational license. Visual Components, V-REP, RobotStudio, Workspace, Webots are some examples. The use of simulators can significantly impact on the efficiency of a project, reducing costs involved in robot production, chance to testing before implementing, demonstrating if a system is viable or not, simulating various circumstances without involving physical costs. However, all them encounter with its own limits, in which many scenarios in the real world cannot be simulated.

Fig. 3.2.1. Webots, V-REP and Visual components graphical interface, respectively

3.3. Why Gazebo?

Gazebo it may not have so many features as a commercial simulator or a so user-friendly interface, but it fits with some other requirements that make it a leading robotic simulator. Gazebo is completely open-source, which means a complete control over the simulator, and freely available (a major advantage over the other available software). Moreover, it is the default simulator used in ROS framework, although they are separate objects, there is a package (*gazebo-ros*) for integration of both. This allows Gazebo to run large and more complex systems. ROS communication will be briefly explained in a coming chapter. To conclude, it is effective and offers rich environment tool and is the widely used on the research field.

The basic working of the simulator is as follows: to reproduce the dynamics of the robot

themselves in a simulation, they are modelled as structures of rigid bodies connected via joints (articulations). To enable locomotion and interaction with an environment, forces, both angular and linear are applied to surfaces and joints. The world (the environment) is described by landscapes, extruded buildings, and other user created objects. Almost every aspect of the simulation is controllable, from lightning conditions to friction coefficients.

3.4. Get started

Gazebo can be free installed from its website (http://gazebosim.org). To continue the code development made in this project would be recommended to clone the repository: https://joelllm@bitbucket.org/joelllm/line-tracker-vehicle.git

Before starting codify, it is important to get to know the simulator and how is structured, which is explained in the next section.

3.5. Gazebo Architecture

It consists of two executables: gzserver, which is the core of Gazebo and can run independently from the second one, gzclient, which is the graphical user interface (GUI) where the simulation is visualized and some controls are provided to actuate over the simulation properties, and is not independent. Gazebo can be executed with or without graphical interface (headless) to perform the simulations: Along the project, the headless execution has not been used. Both executables are connected with inter-process communication.

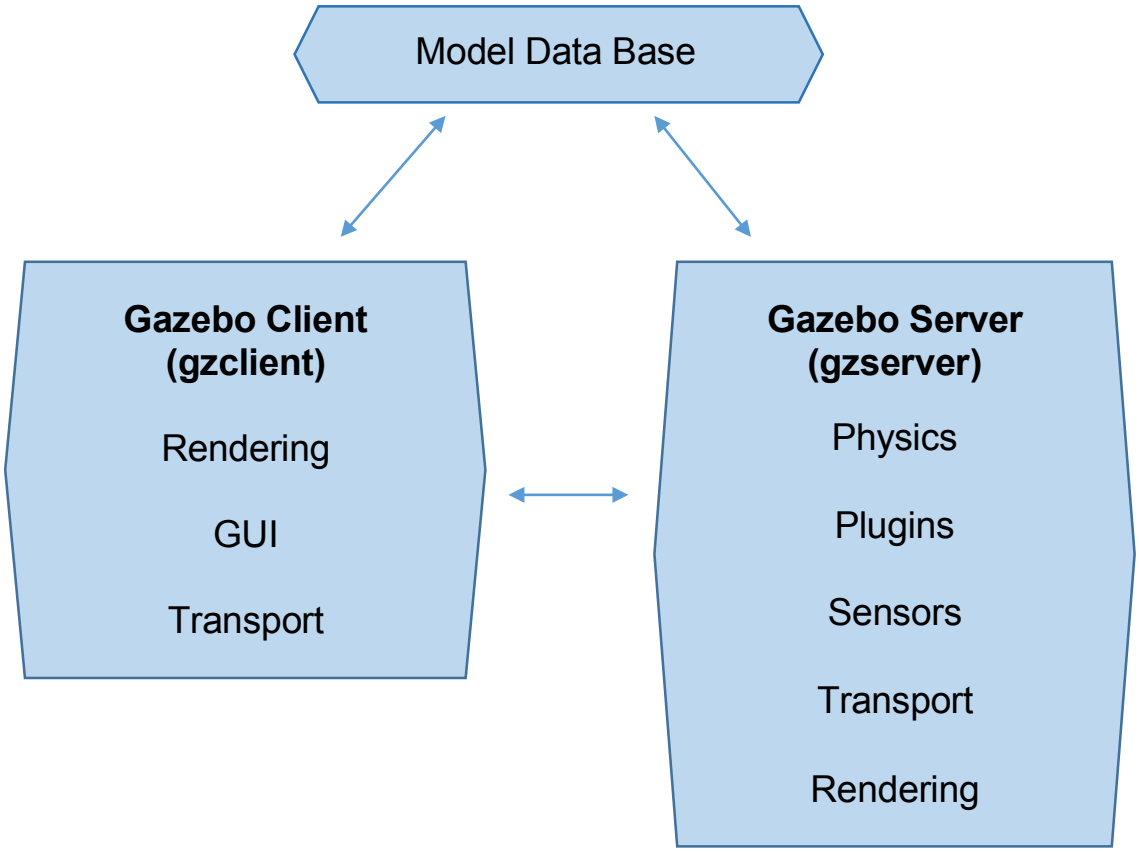


Figure 3.5.1. Gazebo architecture diagram

Gazebo uses a distributed architecture with separate libraries for physics simulation, rendering, user interface, communication, and sensor generation.

3.5.1. Physics

The physics library is responsible for making a reliable simulation, the objects of the scene that we are simulating have to interact coherently with each other following the physics laws. Gazebo supports multiple physics engines: Open Dynamics Engine (ODE), Bullet, Simbody and Dynamic Animation and Robotics Toolkit (DART). ODE is the default engine in Gazebo and the one used in the project. It is based on rigid body dynamics and collision detection; the bodies can be articulated between them using joints. It uses an absolute coordinates system, where each body has six degrees of freedom and each joint introduces a dynamic constraint.

Moreover, it offers many further features, some of which have been useful for modelling the vehicle: Friction and damping coefficients, velocity limit in the joints, rigid body inertias, amongst others.

3.5.2. Plugins

Gazebo plugins provide users direct access to the internal capabilities of Gazebo. This enables users to write custom code for controlling the behaviour of various components in the simulation, the computer language used is the C++. Plugins can operate in different levels: world, system, model, sensor, GUI and visual. For example, a world plugin can spawn objects on a scene and model plugin can used to the steer a vehicle. In our case, a model plugin has been used to control the velocity of the the wheels and two sensors plugins to get access to the two camera sensors. The model plugin retrieves information from the other two thanks to the transport library explained next.

3.5.3. Transport

The transport system uses a topic-based publisher-subscriber model. For example, in this project, the left camera sensor publishes the average colour retrieved from its visual field, it publishes the colour under the topic *ColorLeft,* then the vehicle plugin subscribe to this topic and gets the average colour. This can be understood as a net of nodes, which can act either as publisher or a subscriber, and the connections between them are the topics.

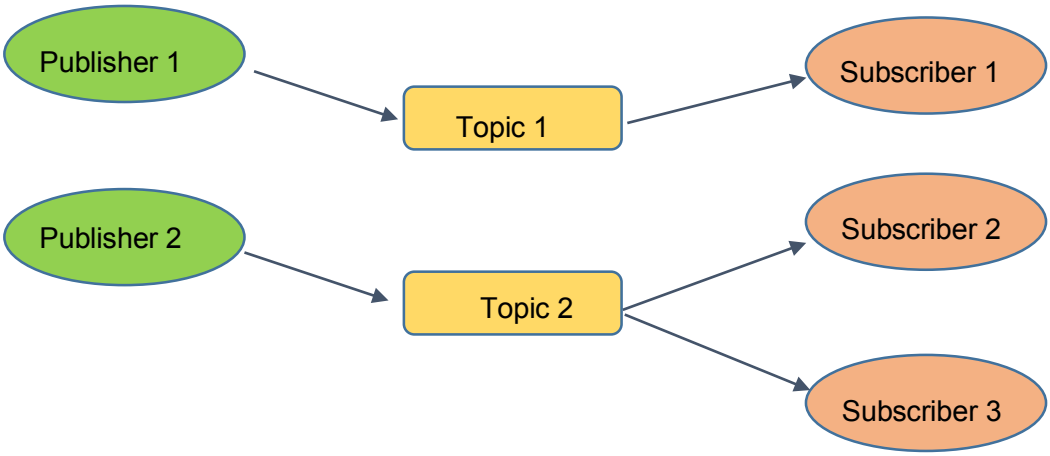


Fig. 3.5.3.1. Gazebo transport network

There is a wide range of sensors available in Gazebo, which can be divided in two groups depending on the method of generating data: image-based and physics-based. The image-based method relies on the Graphic Process Unit (GPU) which they use it to produce image data of the environment as seen from the sensor's perspective. For example, the camera used in the vehicle retrieve an image frame depending on its position and orientation in the scene, which in our case is in the vehicle and oriented downwards. The image frame gets update

each time the GPU reprocess the scene (quantified in frames per second). In some cases, the GPU can act as bottleneck, when the desired update rate of the sensor is higher than the frame per second rate of the GPU.

The physics-based method makes use of physics data such as forces, torques, velocities and contacts associated with the entities in the simulation. For example, in this project it is used an ultrasound sensor which gets the position of the nearest object in his field of effect, the sensor publishes if the position of some object in the scene is inside his field of effect and how far from the sensor is.

3.5.5. Rendering

The rendering library provides a 3D scene to both the Gazebo client and the image-based sensors. The models of the scene are rendered using position and geometry data to build an object made of vertexes and then a texture is applied on. It uses OGRE, an open-source graphics engine.

3.5.6. GUI

The GUI library uses Qt to create graphical widgets for users to interact with the simulation. The user may control the flow of time by pausing or changing time step size via GUI widgets. Additionally, there are some tools for visualizing and logging simulated sensor data.

3.6. Robot and environment modelling

The simulation environment to be generated needs a world description, which will be populated with models that can be either stationary or dynamic, from a simple sphere to a complex humanoid. To represent both, models and world, Simulation Description Format (SDF) files are used.

SDF is an XML format that describes objects and environments for robot simulators, visualization and control. Originally developed as a part of Gazebo, over the years it has become stable, robust and capable of describing all aspects of robots, static and dynamic objects, lightning, terrain and even physics. Some of this features will be explained briefly.

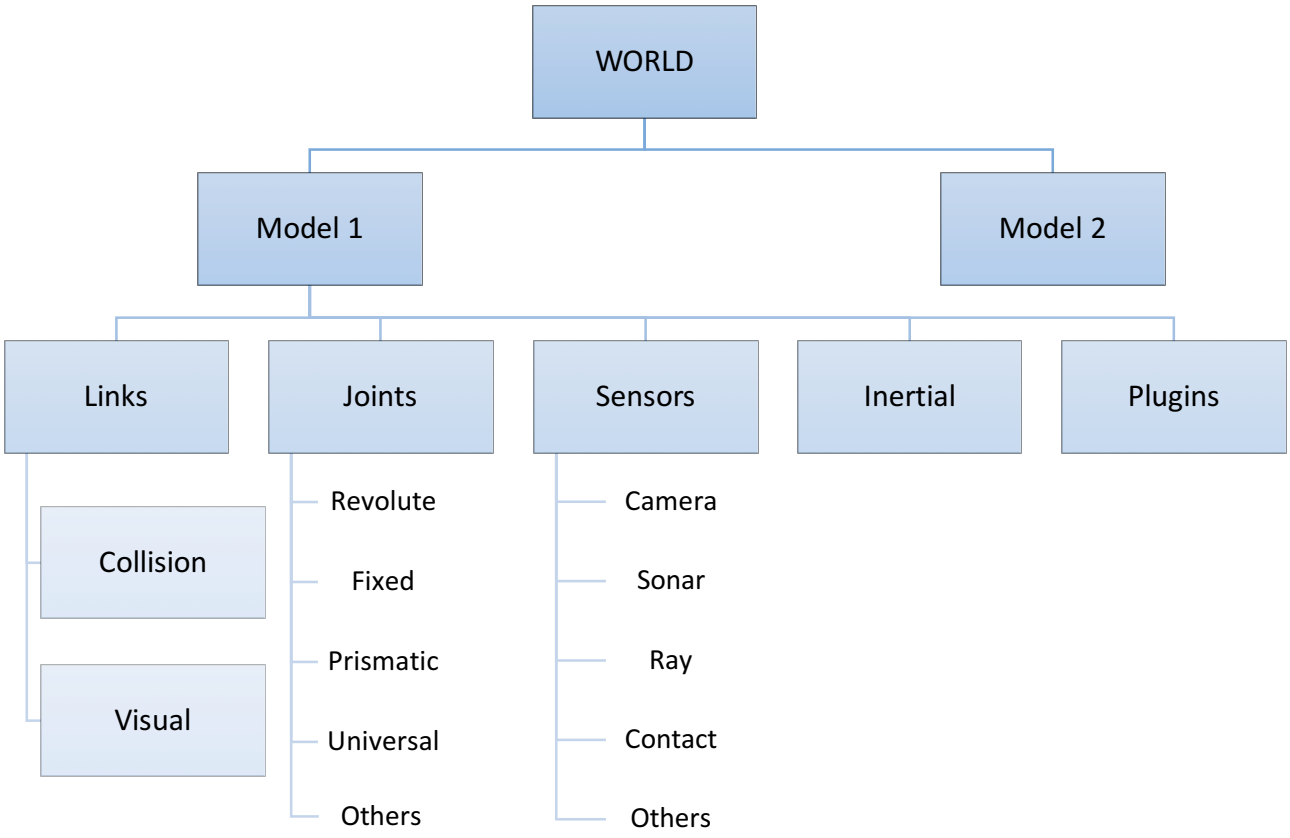


Fig. 3.6.1. Environment hierarchical structure

As the graphics gives off, SDF uses hierarchical structure which makes very intuitive the code construction.

3.6.1. World

Gazebo needs a world file to load the simulation. It describes the scene characteristics such as the wind, the light, the gravity and with which models is populated. In the project, the features of interest have been the light and the models used, which have been the vehicle and the ground.

3.6.2. Light

Gazebo supports three types of light: spot, directional and point. The spotlight is a light source that has a cone of effect, in the directional light or infinite light all the rays are parallel to one direction and the pointlight is a source that from a single point emanates in all directions.

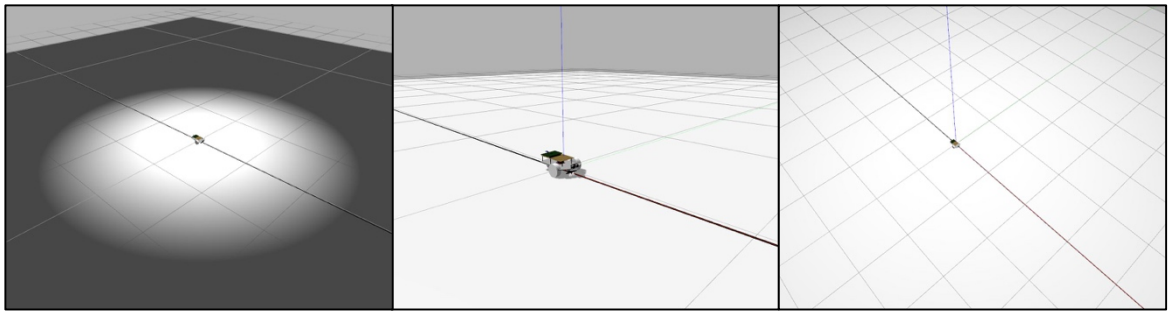


Fig. 3.6.2. Spotlight, directional light and pointlight scenes

3.6.3. Models

A model is any object that maintains a physical representation, whose design depends on the complexity of the desired model. The components that enable the design of the model are explained next.

3.6.4. Links

One model may have several links. A link contains the physical properties of the part of models body which is representing. This can be either a wheel of a vehicle or the head of a humanoid. Each link has assigned inertial properties, friction and it include many collision and visual elements. It is highly recommended to reduce the number of links to just the ones which are necessary in order to improve performance and stability.

3.6.5. Collision

A collision element encapsulates a geometry that is used to collide with the other collisions (Gazebo checks the surfaces of both collisions shapes if they are intersecting). This can be a simple shape, which is preferred, or a triangle mesh, which consumes greater resources.

3.6.6. Visual

A visual element is used to visualize parts of a link, and is useful to give realism to the simulation. More complex shapes can be described on this section. A link may contain zero visual elements. It contains the rendering properties such as colour, texture, transparency.

3.6.7. Inertial

The inertial element describes the dynamic properties of the link, such as mass and rotational inertia matrix.

3.6.8. Sensor

A robot can't perform useful tasks without sensors. It is a devise that lacks of physical representation, and only retrieve data from the simulation. It gains physical expression when is added to a model.

3.6.9. Joints

A joint connect two links. A parent and child relationship is established along with other parameters such as axis of rotation, joint limits and friction. The joints between parent and child allowed are the revolution on one axis, revolution on two axes, gearbox, screw, ball, universal and fixed. The child movement is relative to the parents system of coordinates.

3.6.10. Plugins

A plugin is a shared library, which has access to the model properties and allows to control it. It has similar behaviour to the code compiled to a microcontroller in a robot. Based to the data that it acquires produces a response signal to command some of the elements of the model.

3.7. ROS Integration

ROS is a collection of libraries, drivers, and tools for effective development and building of robot systems. In the project, the communicate with ROS was made through the package gazebo-ros which allows to use some of the ROS functionalities. It contains plugins that can be attached to the objects in the simulator scene and provide easy communication methods, such as topic published and subscribed by Gazebo. The initial purpose of the project is to be developed in stand-alone Gazebo, and in almost along all the work it has been. However, for some requirements it has been necessary to use some of ROS tools:

- **rqt_plot package:** a plotting package for visualizing in real time simulation data.
- **command function *rostopic hz*:** So as to get to know the update frequencies of the sensors (cameras and sonar) and the model plugin that performs as the controller of the vehicle, thus checking steps time of the control system elements.

4. The line tracker vehicle

The vehicle line-tracker simulated in this project has been developed by undergraduate students completing their Final Grade Project and is, at the present, being further developed. To fulfil the purpose of path tracking, the vehicle counts with some necessary elements:

- **Line sensor LRE-F22:** Based two photodetectors that detect the brightness received from the floor surface in order to detect the black line that will perform as path to follow.
- **Ultrasonic sensor HC-SR04:** A range sensor to sense an object on the vehicle path.
- **Vehicle's traction:** It is based on two DC Motors that give torque to the two wheels, a third passive caster, without traction, is implemented to the give the third support to the chassis. The electric behaviour of the motors provides the wheels with a saturation angular velocity. Moreover, an encoder is added to sense the the angular speed of each wheel so as to give feedback to the motor control.

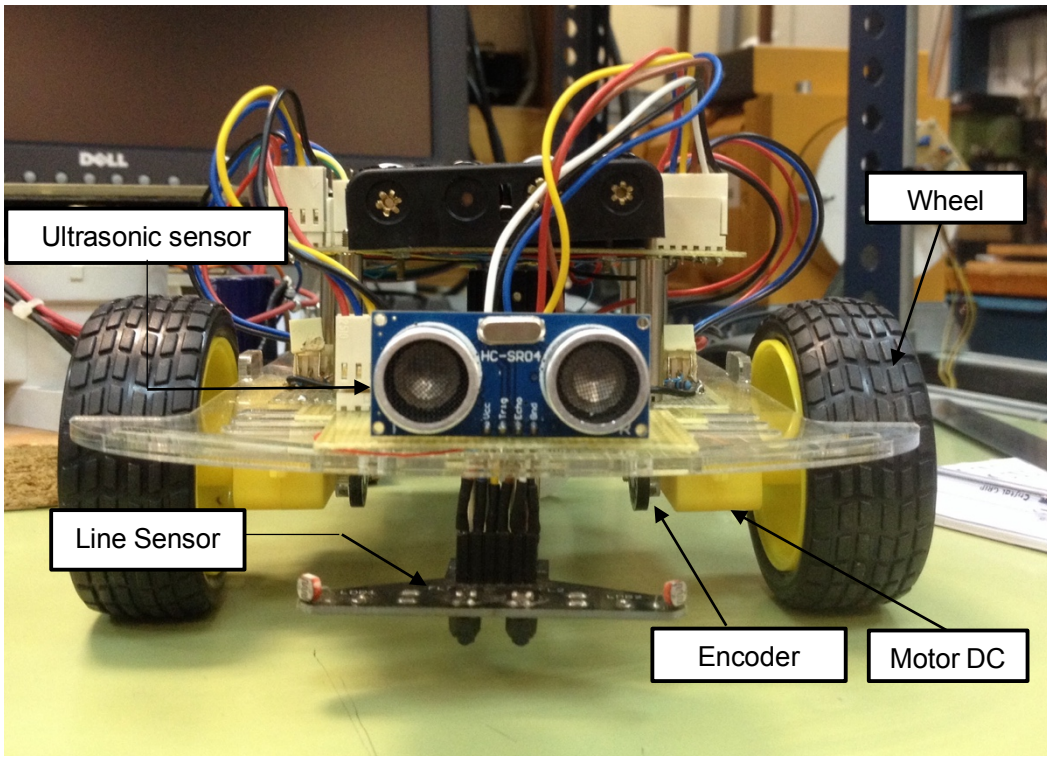


Fig. 4.1. Line follower vehicle developed in the laboratory

These three elements will be modelled in the simulation, whereas the other electronic hardware, as drivers or alimentation circuits, are not directly considered in the simulation. Finally, to successfully drive the vehicle is necessary:

- **Control architecture:** The control of the vehicle is commanded and executed through the microcontroller and implemented with electronic hardware.

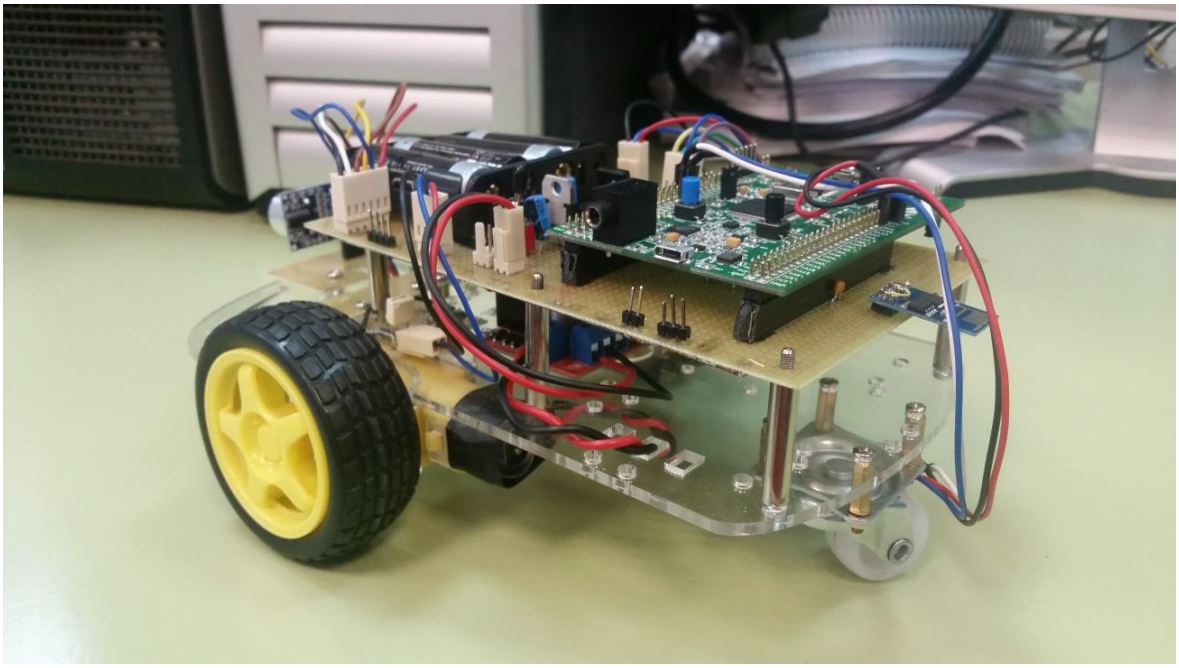


Fig. 4.2. Line follower vehicel developed in the laboratory

5. Building the Gazebo vehicle model

5.1. Assembly of the vehicle

The assembly of the vehicle has to be consistent with the SDF model structure, which has been explained in the 4.6 section.

The vehicle model is formed by one parent link and two child links, which are the chassis and the two wheels respectively. The wheels are connected with the chassis with two revolute joints of one degree of freedom.

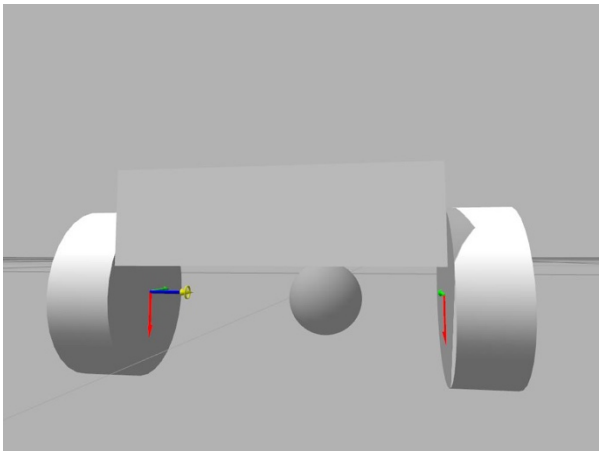


Fig. 5.1.1 Vehicles collisions connected through the joints indicated with vectors

The design of the collision must be basic and simple in order to have an efficient and rapid computation behaviour, which will be reflected in a higher simulation time. In the other hand, the design the mesh for the visual part has been can be more accurate and descriptive. Both parts have been designed following the real sizes of the robot, measured properly in the laboratory.

5.1.1. Collisions

The chassis has been created in the collision part as a box with two more necessary shapes attach to it. The first is a platform located under the box and is used to support the line sensor. The second is a ball, which represent the free central wheel. As its function is to give to vehicle the third support to the floor, so it does not fall, and does not add a kinematics constraint to the

vehicle (passive caster wheel), its mechanical behaviour has not been implemented as it didn't affect the goal of the project to study the control of the vehicle. The wheels have been built in the collision part as a cylinder.

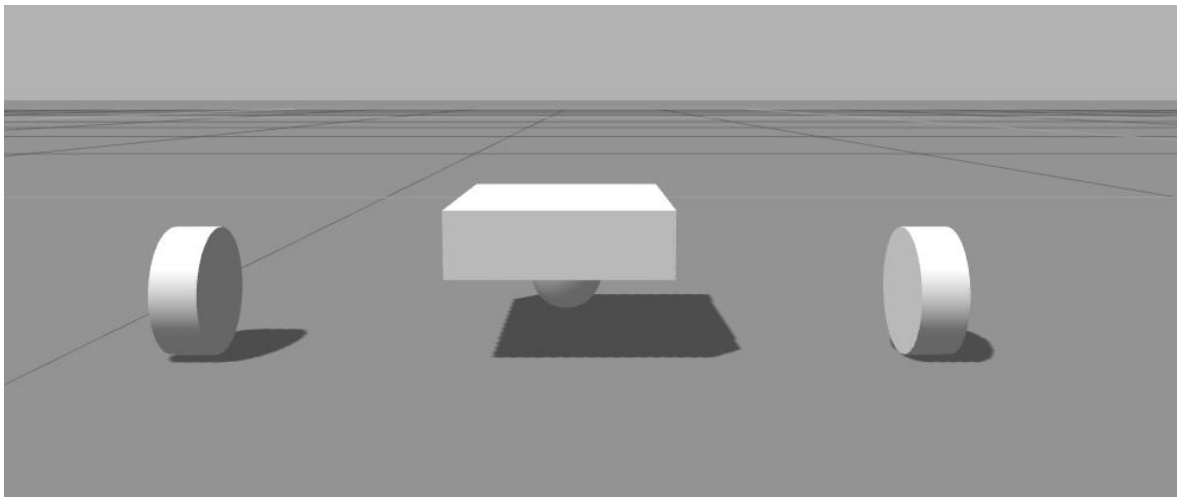


Fig. 5.1.1.1.1 The three collisions separated

5.1.2. Visual

The design of the mesh has been made with *Sketchup* 3D editor and its *3D warehouse* open platform to implement some of the vehicle elements, such as the driver, the motor, the wheels and the Wifi module.

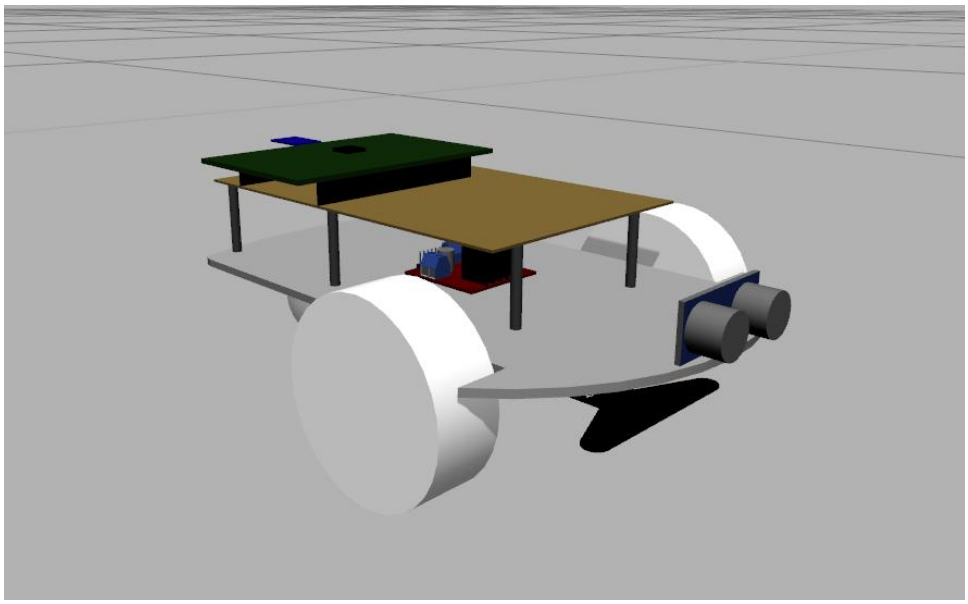


Fig. 5.1.2.1 The final design of the vehicle in the simulation scene

5.2. Line Sensor

The line sensor in the vehicle is implemented with the LRE-F22 component. It has two infrared emitters and two infrared receivers (FL1 and FL2 in the image above) that detect how much of the infrared emission comes back to the sensor. There are then two output signals from this sensor: V_FL1 and V_FL2, that go from 0[V] to “VCC” [V] depending on how black (or white) is the surface below the two infrared receivers. Thus, if the sensor FL1 or FL2 receives all the infrared light back (we are in a reflecting surface, outside of the line) then its pin will be at VCC, and viceversa.

The infrared sensor performance is physically complex as it depends of the behaviour of the electromagnetic waves and its interaction with surfaces. This feature is not directly provided by the physics engine in *Gazebo*, which its fundamental core is the dynamical interaction between rigid bodies.

As it is desired to simulate just the functionality of the line sensor and not its physics behaviour, it has been thought more adequate to simulate the line sensor with two cameras, which are sensors provided by *Gazebo* that retrieve data from the rendered scene. The next step would be carrying out an image processing of both cameras and give and output value which would represent the outputs V_FL1 and V_FL2 in the real vehicle.

Fig. 5.2.1. The image perceived by the two cameras in the line sensor

An iteration for all the pixels of the camera image is made. Each pixel has an RGBA format, the R, G and B stand for the primary colours, red, green and blue, which have a value between 0 and 1. The A stand for alpha and it represents the opacity of the pixel, 0 gives a fully transparent output and 1 fully opaque.

As the image observed is the ground, which is set by in the building of the world scene, a

hypothesis of full opaqueness is made and the alpha value is designed always 1. The iteration subtracts the average R, G and B values of the whole image. Then, the luminosity can be calculated from a linear equation:

$$Y = 0.2126R + 0.7156G + 0.0722B \quad \text{(Eqn. 5.1)}$$

The formula reflects the luminosity function: green light contributes the most to the intensity perceived, and blue light the least.

5.2.1. Ground texture

The image that the camera will retrieve comes from the ground colour characteristics that we set. This concerns the visual part of the ground model. As the goal project is to simulate line-following, the texture of the ground will be quite simple. A texture is an image that is applied to an object. To create this textures has been used the software *Inkscape* (5). When applying a texture some parameters have to be set in order to describe its interaction with the light:

- **Ambient colour:** Revealed when the object is in shadow. This colour is what the object reflects when illuminated by ambient light rather than direct light.
- **Diffuse colour:** Most instinctive meaning of the colour of an object. It is that essential colour that the object reveals under pure white light. It is perceived as the colour of the object itself rather than a reflection of the light.
- **Emissive colour:** This is the self-illumination colour of an object.
- **Specular colour:** Is the colour of the light of a specular reflection (specular reflection is the type of reflection that is characteristic of light reflected from a shiny surface).

Our parameters of interest are the ambient and diffuse colour, the other two are defined with with 0 emission and the specular colour has been set at its default value. The floor texture has a white (1 1 1 1 RGBA) diffuse colour and grey (0.8 0.8 0.8 1RGBA) ambient colour. The line texture has the same RGB values for both colours (0 0 0 1 RGBA) as the shading on the black region has no effect, as it can not turn any darker. The alpha value is always 1, as the floor is considered completely opaque.

5.2.2. Cameras characterisation

In the laboratory, a characterisation of the line sensor through the line has been made. The vehicle, which is oriented parallel to the straight black line, is moved perpendicular across the line above a white surface, beginning on the line's right side.

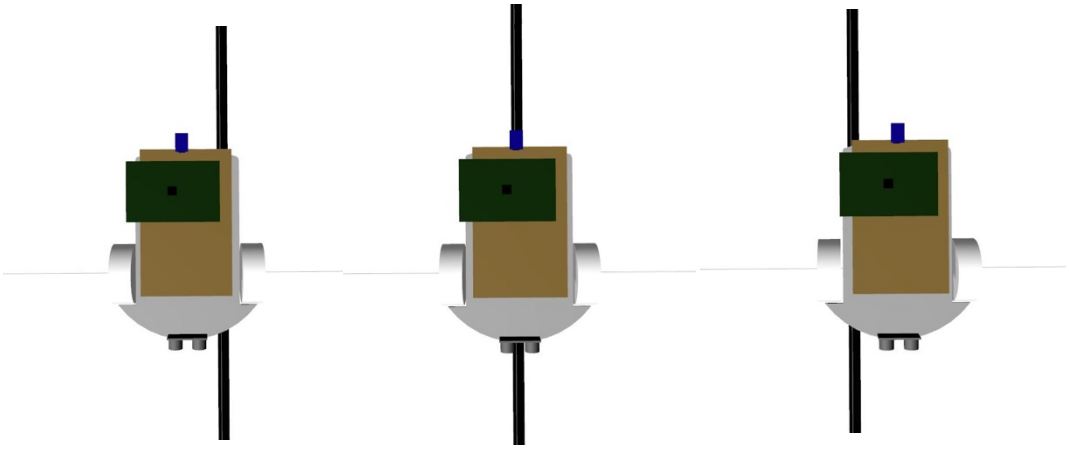


Fig. 5.2.2.1 Scheme of the experience developed to characterize the line sensor

This experience has been made perpendicular to the line axis because in the desired performance of the vehicle following the line to be achieved, the centre of the robot is situated on the line axis and its longitudinal axis is parallel.

The cameras output, which is the luminosity of its field of view, reaches the maximum 1, when the region in its field of view is completely white, and decreases when in its field of view starts appearing part of the black line. The results are shown in the next graphic with the difference between the left and right cameras output.

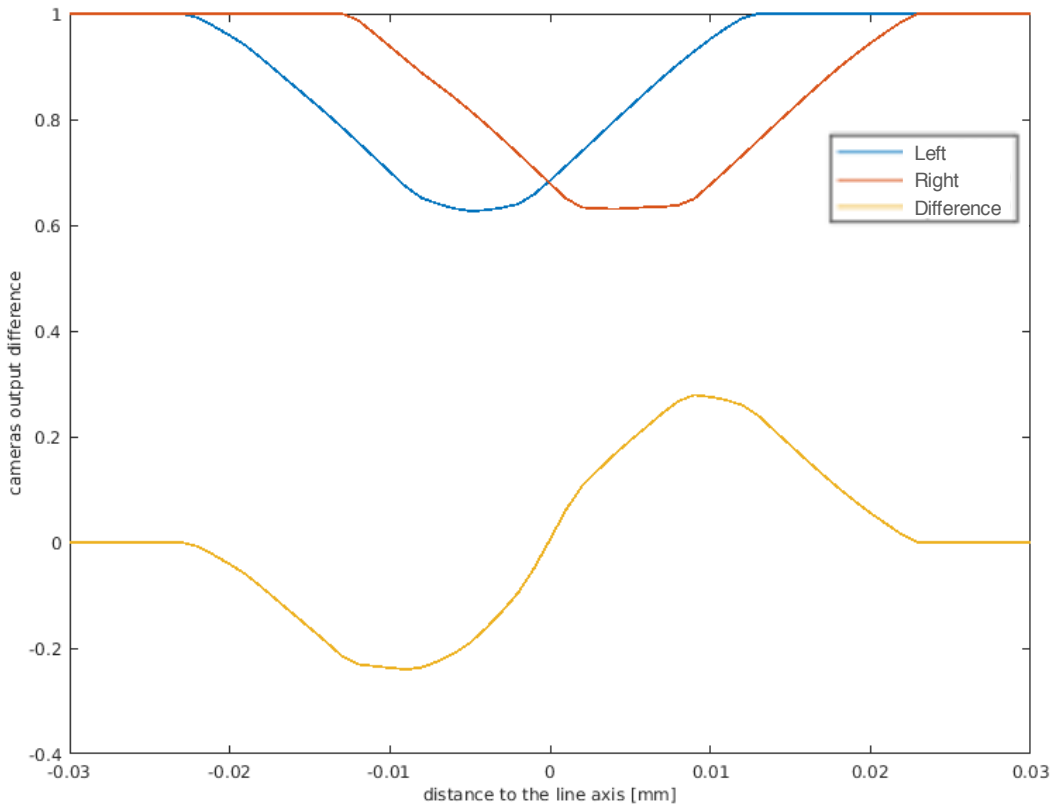


Figure 5.2.2.2. Graphic the cameras output versus the location of the vehicle respect the line axis

In this example, which is for a 1.2 rad angle of view, the luminosity perceived for the left and right cameras decreases and increases lineally as expected (the equation 1.1 is lineal, the values R, G, B decreases and increases lineally as black is [0,0,0] and white is [255,255,255] and the weight of white and black in the image too). There is a bottom flat part which are the values when the camera is in the range of [-0.002, 0.002] m positions where the output is the same as in all of them the whole line and the same weight of white is seen. There is a delay of 0.01 m between the left and the right cameras output, which is the distance between them in the robot. The difference between both outputs provides a correlation between the vehicles distance to the line axis and the cameras output.

To decide which angle of view to choose, an experience without shades and iterating for the different angles has been done. The angle of view describes field of vision of the camera, and is denoted with radians. An iteration from 0,5 rad to 2 rad with a step of 0, 5 rad has been carried out. The results are the followings:

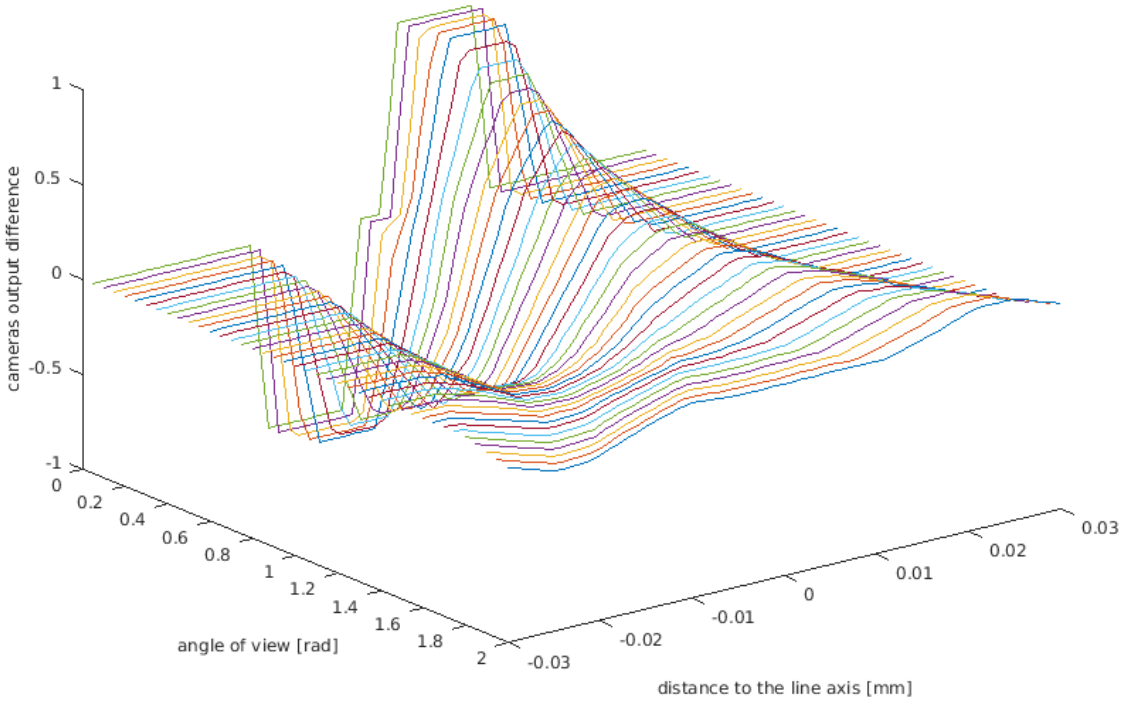


Fig. 5.2.2.3. Iteration for different angles of view, the cameras output difference versus the distance to the line axis.

As far as can be observed, the output response has a slope around x values near to 0, which decreases as the angle of view increases, until it reaches a 0 pending. The angle of view from which starts the dead zone (0 pending) can be geometrically determined with the camera position and the line width and corresponds when the angle reaches the opposite line boundary when the vehicle is centered on the line's axis.

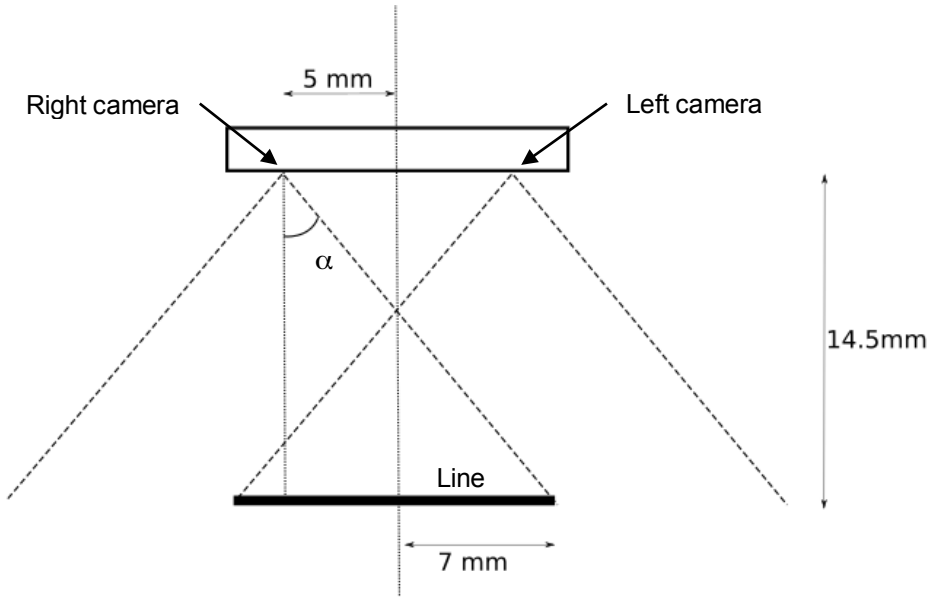


Fig. 5.2.2.4. Geometric representation of the cameras angle of view above the line

$$angle\ of\ view\ limit = 2 * \alpha = 2 * \tan^{-1}\frac{12}{14.5} = 1.38\ rad \qquad \text{(Eqn. 5.2.)}$$

From this angle of view on, the cameras image reaches all the line and more. We want to avoid these regions, as it does no allow correlating the position of the vehicle with the camera output. As concerns the narrower angles of view, it can be seen a dead zone in the middle, it means that both angles of view are on the line and both retrieve 0.

In order to choose the angle of view that more similarly reproduce the characterization of the vehicle line sensor, has been chosen this rang of angle from 0.75 rad to 1.4 rad, because lower values present abrupt changes on the top and bottom of the graphic response and higher values present the dead zone region. The possible values are the followings:

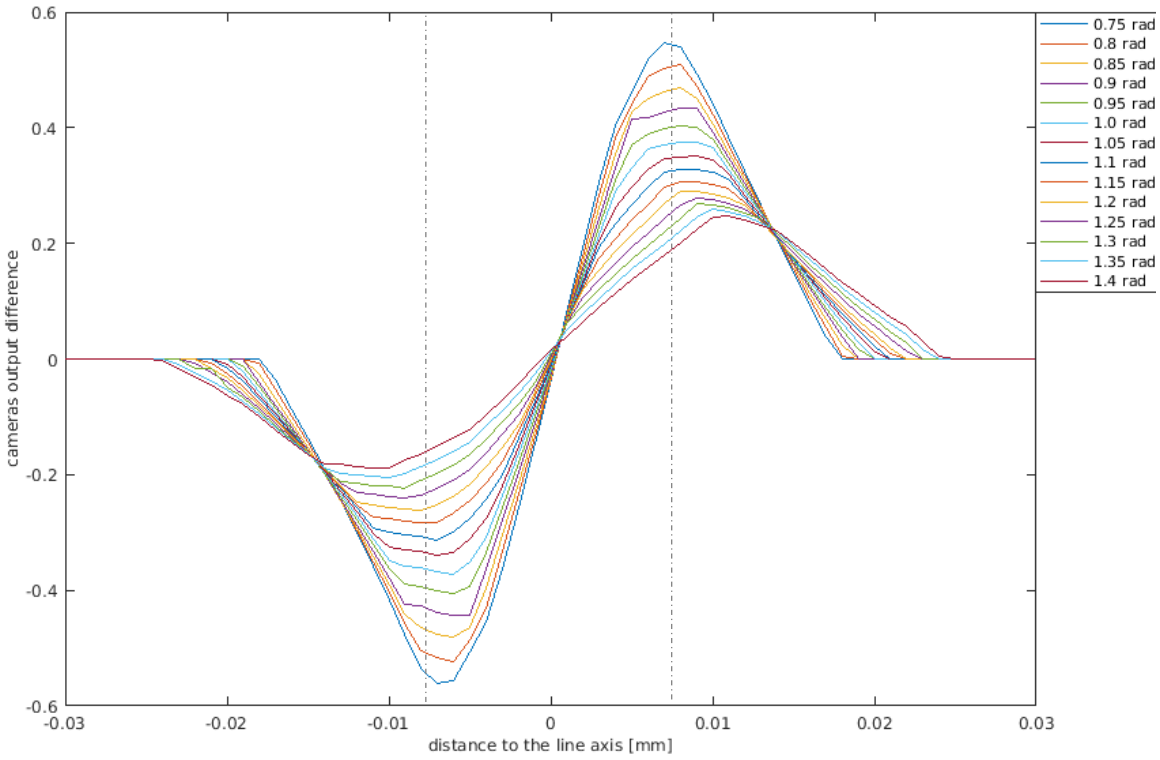


Figure 5.2.2.5 Graphic with all the possible angles of view with two auxiliar lines in -0.008m and 0.008m

It has been chosen 1.2 rad angle of view as it gives us a rang of values from -0.008 to 0.008m which is the rang used in the line sensor in the real vehicle.

5.2.3. Shadows

Although the shades have been elided in the previous experience, they cannot be neglected and a study of his affection must be made, the error that introduces to the d obtained. The same previous experience has been made to quantify the error, using the directional lights and changing its direction. The light direction is a 3D vector, where the z component has a major weight in front of x and y components as in scene to study, which is a laboratory room, the light comes from upwards. To develop the experience, it has only proved for different x and y components. Moreover, the chassis has been elided as it is transparent.

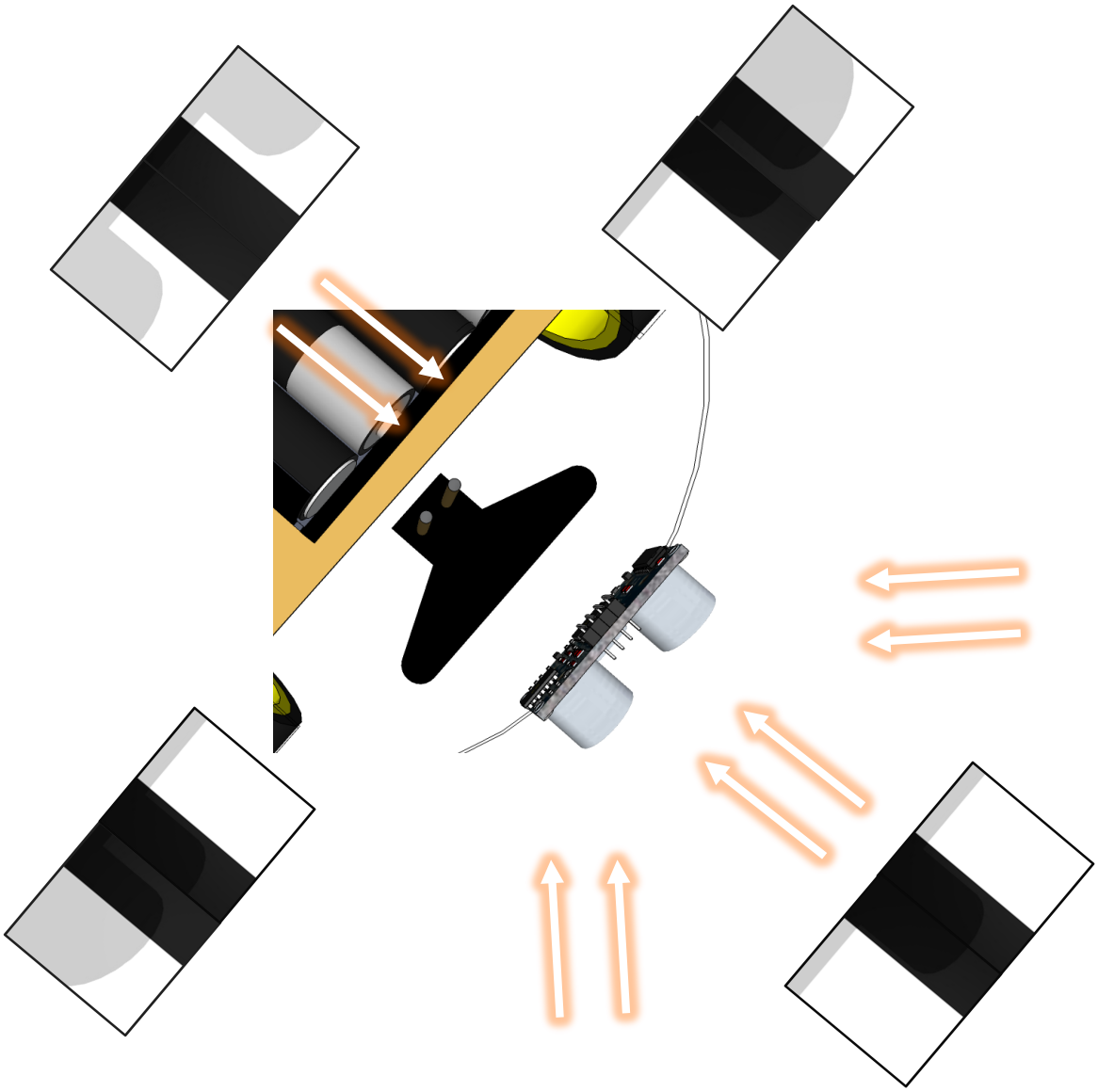


Fig. 5.2.3.1 Represetation of the x and y components of the light direction an the casted shadows on the cameras image

This representation tries to make understand the effects of the shadows on the image rendered by the camera. The objects that cast shadows on the image for a range of different light directions are the line sensor itself, if the light comes from backwards, and the ultrasound sensor, if the light comes from frontwards. The four images shown in the figure, are the four possibilities that the cameras may encounter. Backwards light does not modify its shadows if its direction oscillates in the z plane and the both cameras retrieve the same an equally brightness value as the grey weight on the image is the same. However, in the second case when the light does not come parallel to the vehicle longitudinal axis, the brightness retrieved is not the same.

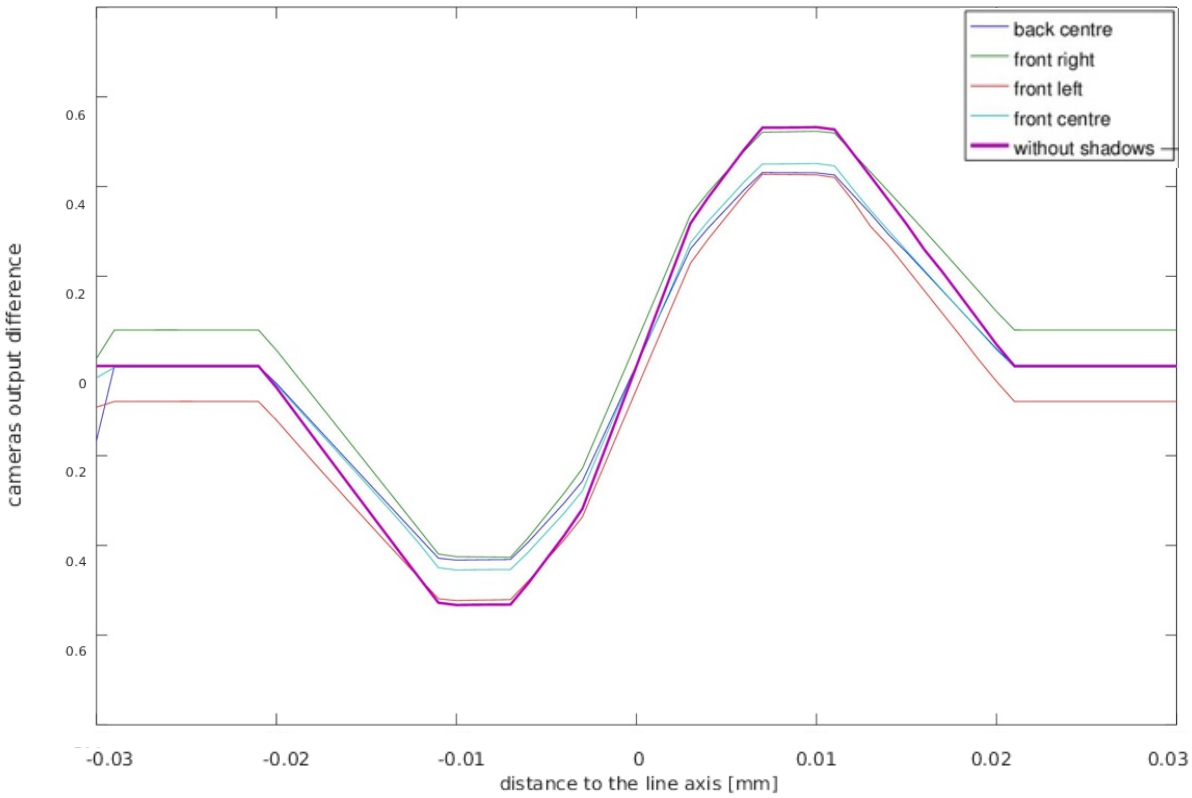


Figure 5.2.3.2. Results for the cameras output different in the four previous cases and one without shadows

In the case of the front right light, the output obtained is a bit higher because the right camera output detects more grey than the left camera and when the difference output is computed (left camera output – right camera output), the result is higher. Viceversa with the front left light. In the case of the front center and back center, they have similar behaviour but have lower values if there was any shadow.

5.3. Ultrasonic sensor

The ultrasound sensor in the vehicle is implemented with the HC-SR04 component. It has one ultrasound emitter, one ultrasound receiver and a control circuit. Depending on the distance to the nearest object in the ultrasound field, the sensor will send a proportionally pulse. Applying the speed of sound, we are able to know the distance to the object.

No characterization of the element has been done in the laboratory. The final purpose of the sensor is to detect near-by objects that can potentially collide with vehicle and Gazebo supports a basic behaviour of the ultrasound sensor and a detailed characteristics table is already distributed by different electronic stores. The most specific table supplied that we found is the one from *Cytron Technologies* (5).

In Gazebo, have been only specified the minimum range, the maximum range of the detectable object, the resolution and the radius of the cone base in its maximum range. All these features are displayed in the sensor table:

- Minimum range: 0.02 m
- Maximum range: 4 m
- Cone radius: 1.07m
- Resolution: 0.003 m

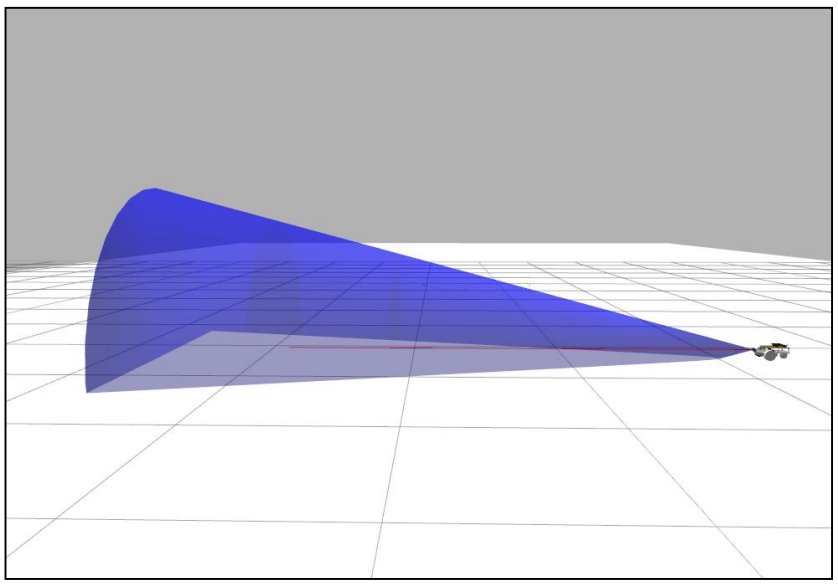


Figure 5.3.1. Ultrasonic sensor cone of effect

5.4. Vehicle's traction

The motion of the vehicle is a significant part in the modelling of the vehicle behaviour. The traction engine has been implemented with a Driver and two DC Motors, the driver regulates the speed of each wheel by sending a control signal to the DC Motors, which give torque to the wheels. In the transmission chain between the motor and the wheel, exist tangential forces that actuate against the movement of the wheel. Gazebo provides three force coefficients to accurate implement the tangential forces. In order to study the effects of these parameters over the wheel response, a test has been made for a command signal to the joint to rotate in a target velocity of 3 rad/s.

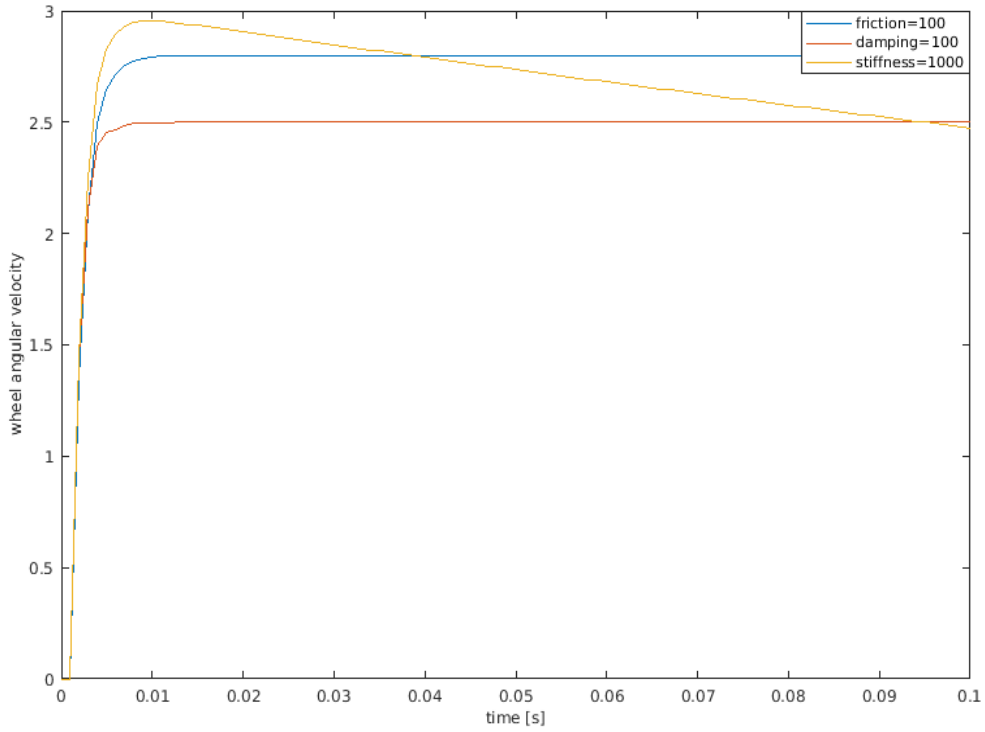


Fig. 5.4.1 Time response of the wheel in front of different force coefficients

The friction coefficient describes an opposite force to the traction torque applied to the wheels and is proportional to the normal force in the contact surface. The damping coefficient acts the same way as the friction coefficient, but it is proportional to the angular velocity instead. As a result, in both cases the joint is not able to reach the velocity target.

In the other hand, the stiffness coefficient produces a much different behaviour on the wheel response. It can be described as the rigidity of the joint and is proportional to the angular displacement.

Another study point has been the time response of a control signal implementing a PID controller, whose objective will be to reproduce the time response of the real vehicle. In the laboratory robot, the wheels have a first order response of 0.1s, which corresponds to Kp of value 50 n the simulation vehicle.

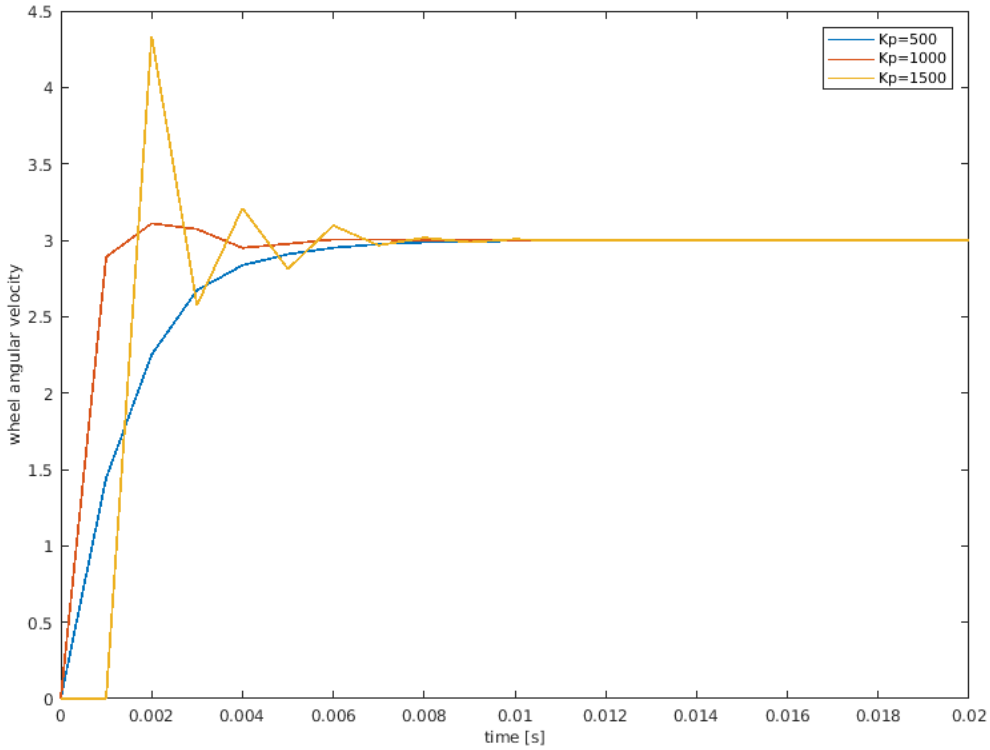


Figure 5.4.2. Time response of the wheel in front of different Kp

Finally, an angular velocity limit has been set to joints, which corresponds to the saturation velocity of the DC motors that has been calculated in the laboratory: 18,46 rad/s.

5.5. Inertia and mass

The inertia has an important role in the rotation movements, as it represents the mass distribution respect reference axes. Depending on the inertia respect the rotary axis, the resistance to the angular acceleration is high or low. The vehicle kinematics is not greatly affected by the inertia, because the angular accelerations of the vehicle are not very high, the maximum working lineal speed (v_1) is around 0.6 m/s. The effect of inertia in the rotation of the wheels is more significant and the working of the motors are affected. Nonetheless, the inertia of the wheels is not very high (the order of 10^{-4}) and is not supported by Gazebo as it sees them as a 0 and produces a computing error.

In case it is able to perform to higher working speeds or the robot encounters other scenarios where inertia has an effect on the vehicle response, it has been thought useful to compute the inertia. The tool used to compute the inertia matrix of the vehicle has been *Meshlab*, a free software mesh processor. It has been considered homogeneous density, which does not ideally reflects the mass distribution of the vehicle, but is useful as an approximation.

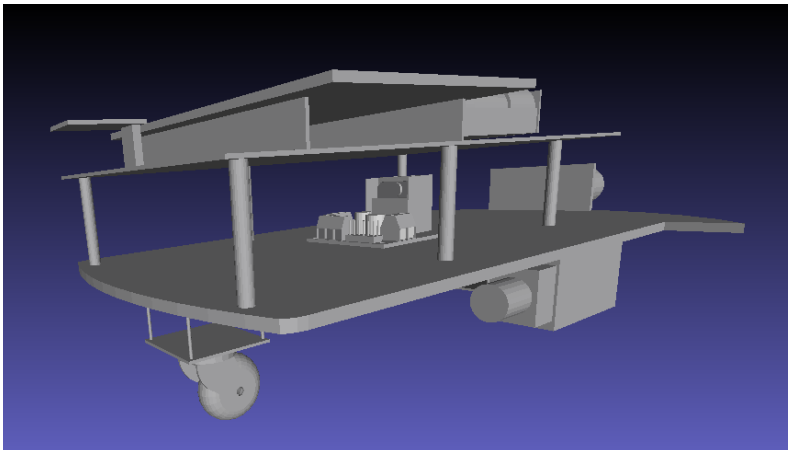


Fig. 5.5.1 Meshlab interface with the chassis mesh

The resultant inertia tensor is:

$$\begin{bmatrix} 0.001806 & -0.02510 & 0.06223 \\ 0.06452 & 0.07551 & 0.02314 \\ -0.00787 & 0.00594 & 0.02500 \end{bmatrix} [kg/m^2]$$

It has been calculated only the chassis inertia, as can be observed in the figure 5.5.1. The

inertia is computed in the description the link chassis in the model file.

The mass may have an influence in big accelerated movements, in case of the starting to move the vehicle or when it brakes. As it has been told before, because of the low performing speeds, the mass may not have a big influence. Nonetheless, the vehicles mass has been weighed in the laboratory, resulting a mass of 0.65 kg and has been described on the model.

5.6. Dynamics

The study of the dynamics is not the purpose of this project as in the control design it has been taken account only the kinematics variables of the vehicle, however they can not be neglected. To model the performance of the vehicle, the study is focused basically in the tangential forces, which, in other words define the contact interaction. The physics engine supports a wide collection of dynamic coefficients to define this behaviour.

To carry out the simulation test, it has been assumed and ideal performance of the joints with no frictions, stiffness nor damping. Nonetheless, in the real vehicle friction in the transmission chain has been detected. As it is observed in the previous chapter, this performance should be modelled with the explained parameters.

In the other hand, friction coefficient on the floor has been declared to enable the correct motion of the the two wheels and the caster wheel, so the vehicle does not slip. As has been told in the 5.1. chapter, the caster wheel does not affect in the vehicles kinematics and just give the third support to the chassis.

6. Designing the vehicle's simulation control

6.1. Control Architecture

The purpose of the control is the vehicle to trace a path which, in our case, is provided by a black line in a white surface. A sensing part is required so to supply the control algorithm with an estimate data of the position and orientation of the robot respect the line to be tracked, in our case, the distance of the optical sensor to the line axis. The optical sensor in the real vehicle consist of two photodetectors and, in the simulation, of two cameras. Moreover, internal sensors as the encoders of the vehicle to perceive the angular velocities of the wheels may contribute to design an internal speed control over the wheels. In our simulation environment, encoders are not needed as the velocity of the wheel can be retrieved from the program internal code.

Path tracking is directly related to the motion and steering of the vehicle, which involves speed control over the angular velocities of the wheels. In our case, the control algorithm works with constant linear velocity. The steering strategy (or the control law) will use the error between the current estimated vehicle position and the path to follow. Therefore, the inputs will be variables defining the location of the vehicle respect to the line and the outputs will be the steering commands. The control signals will be the working linear velocity (v1) and the kinematic constraint so the vehicle follows the line, which is the desired distance of the optical sensor to the line axis (d*) of 0 mm. The vehicle uses a feedback linearized control.

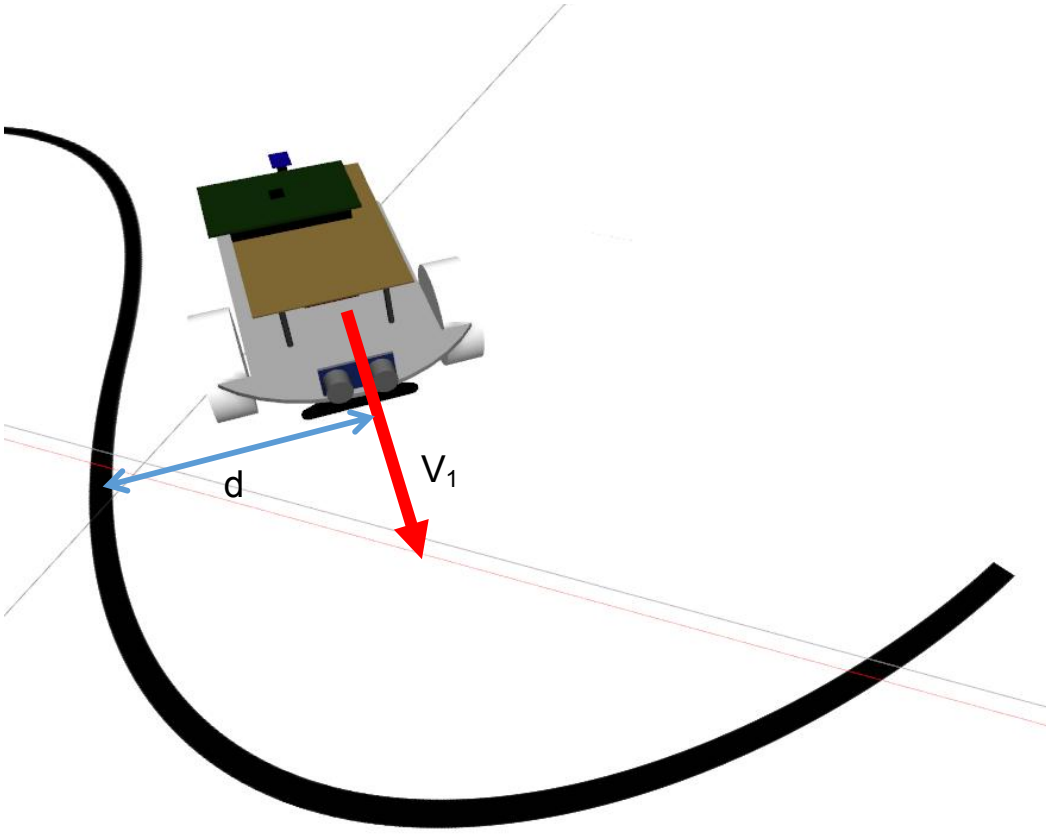


Fig. 6.1. Variables of the control algorithm

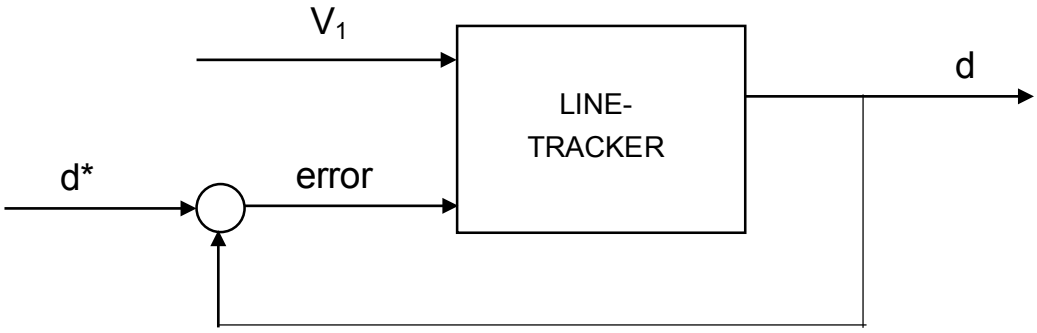


Fig. 6.2. Scheme of the general control

The control system can be divided in two subsystems: an internal one (joint control) that controls the joints rotation speed and an external one (trajectory control) that controls the entire vehicle and send the control signal to the joint control. Both present a control loop that provides feedback data. The internal one is a simple PID-based velocity control by actuating on the wheel's joints and is supplied by Gazebo. In the other hand, the external control is responsible for setting the wheels speed target with the aim of the vehicle following the path. Its feedback signal values describe the position of the vehicle respect the line axis, and is provided by the two camera sensors.

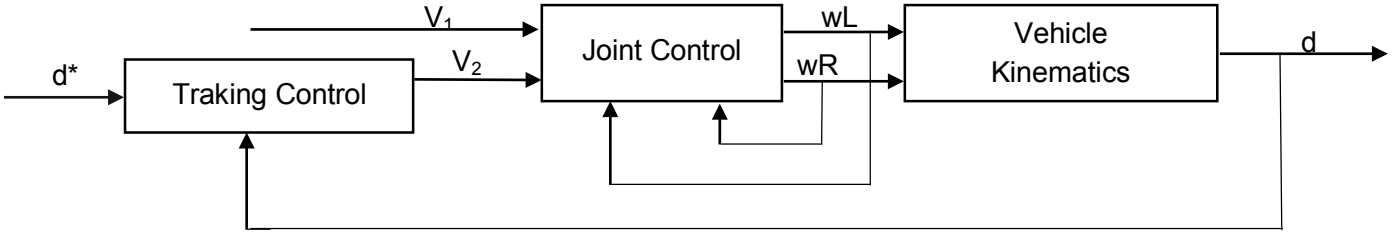


Fig. 6.3. The control scheme with the two subsystems

The v_2 is the control signal that corrects the vehicle's trajectory. The wL and wR are the left and right wheel speeds and d is the distance to the line axis. In the joint control the output data Wr and wL is computed with v_2 and v_1 using the vehicle kinematic relation.

The d is the parameter to compute the signal error in the control loop and has a significant effect on the tracking performance as may arise instability or oscillations related coming from navigation conditions as working speed, path to follow or turbulence on the ocular sensor.

6.2. Tracking Control

6.2.1. Vehicle's kinematics

To study the vehicle motion, it has been considerate only the kinematics of the vehicle, which can be assumed as a 2D model as the vehicle moves on a plane.

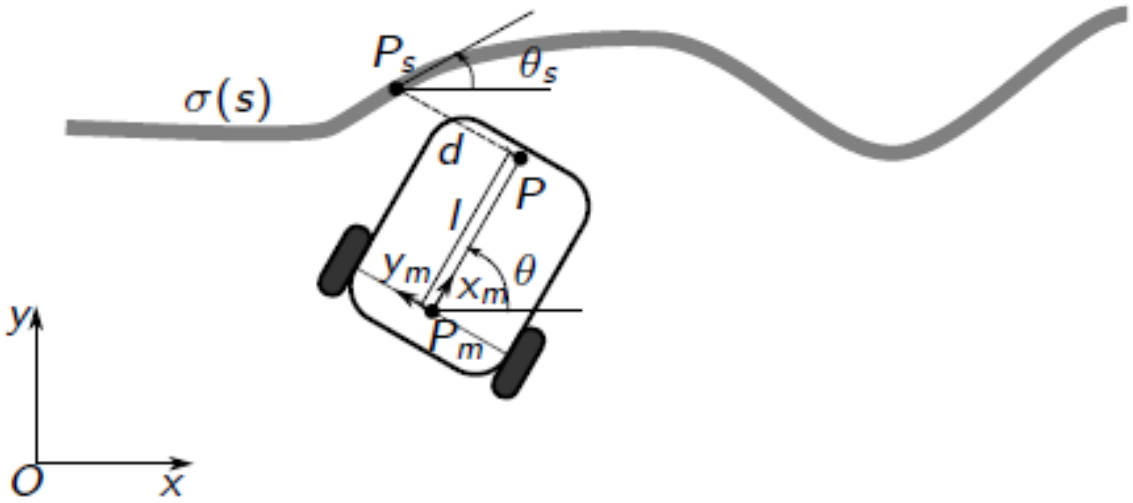


Fig. 6.2.1.1 Variables needed to describe the vehicle kinematics

The position and configuration of the unicycle-type mobile robot in world coordinates kinematic can studied with the point Pm, where the middle of the wheel axis is, and is given by the equations:

$$\dot{x} = cos(\theta)u_1$$

$$\dot{y} = sin(\theta)u_1 \qquad \text{(Eqn. 6.1)}$$

$$\theta = u_2$$

With:

$$u_1 = \frac{r}{2}(w_r + w_l);\ u_1 = \frac{r}{2R}(w_r + w_l) \quad \text{(Eqn. 6.2)}$$

Where r is each wheel's radius, R is the distance between the two wheels, and ω_r and w_l are the angular velocity of the right and left wheels respectively.

The problem translates in a point, P_s (where the optical sensor is placed), that tracks the path, ρ(q), with a certain speed, q = v. In other words, the tracking problem can be written in terms of the distance between points P_s and P_q defined by d = $\overline{P_S Pq}$.

The coordinates of the point P_S can be given by two equations:

$$P_S = Pm + R(\theta)\begin{pmatrix} l \\ d \end{pmatrix} = \begin{pmatrix} \sigma_x(s) \\ \sigma_y(s) \end{pmatrix} \quad \text{(Eqn. 6.3)}$$

Where $R(\theta)$ is the 2D coordinate rotation matrix:

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \quad \text{(Eqn. 6.4)}$$

Parameterising σ(q) with respect to q, differentiating *Eqn. 7.1* and using *Eqn. 7.3*, is possible to determine the motion equations in terms of d, q and θ

$$\dot{d} = lu_2 - tan(\theta - \theta q)(u_1 + du_2)$$

$$\dot{q} = \frac{(u_1 + du_2)}{cos(\theta - \theta q)} \quad \text{(Eqn. 6.5)}$$

$$\dot{\theta} = -u2$$

where $\frac{\partial \sigma x}{\partial s} = cos(\theta_s)$ and $\frac{\partial \sigma y}{\partial s} = sin(\theta_s)$.

Finally, defining a deviation angle θ_e (θ_e = θ − θ_s), the dynamics simplifies to :

$$\dot{d} = lu_2 - tan(\theta_e)(u_1 + du_2)$$

$$\dot{q} = \frac{(u_1 + du_2)}{cos(\theta_e)} \quad \text{(Eqn. 6.6)}$$

$$\dot{\theta_e} = -u_2 - \frac{c(q)}{\cos(\theta_e)}(u_1 + du_2)$$

where $c(q) = \frac{\partial \theta s}{\partial s}$ is the curvature of $\sigma(s)$.

Desired working trajectory

The conditions in order that the vehicle follows the trajectory are:

$$d*=0;\ \theta e* \sim 0; \tag{Eqn. 6.7}$$

Also it is required that the vehicle goes at a constant linear velocity

$$\dot{q}*=v; \tag{Eqn. 6.8}$$

Applying these conditions to the expression to *Eqn. 7.1*, the required control values u1 and u2, and the corresponding deviation angle are

$$u_1^* = v\sqrt{1 - l^2 c(s)^2}$$

$$u_2^* = -c(s) * v \tag{Eqn. 6.9}$$

$$\theta_e^* = arcsin(-c(s) \cdot l)$$

Considering u_1 in *Eqn. 7.9*, the maximum curvature constraint is:

$$c < c_{max} = \frac{1}{l} \tag{Eqn. 6.10}$$

Finally, let us assume that control u1 ensures that q* = v. Then, the remaining dynamics from *Eq. 7.6* yields:

$$\dot{d} = lu2 - vsin(\theta e)$$
$$\dot{\theta e} = -u2 - c(t)v. \tag{Eqn. 6.11}$$

6.3. Feedback signal

The optical sensor's goal is to give to the algorithm control and estimation of vehicles position. For this reason, a correlation between the distance to the line axis and the cameras output has been made. The purpose of using two cameras and neither one nor three is because it is desired to model the performance of the line sensor of the real vehicle, which has two photodetectors. Nonetheless, the study with the implementation with a different number of cameras may be of interest.

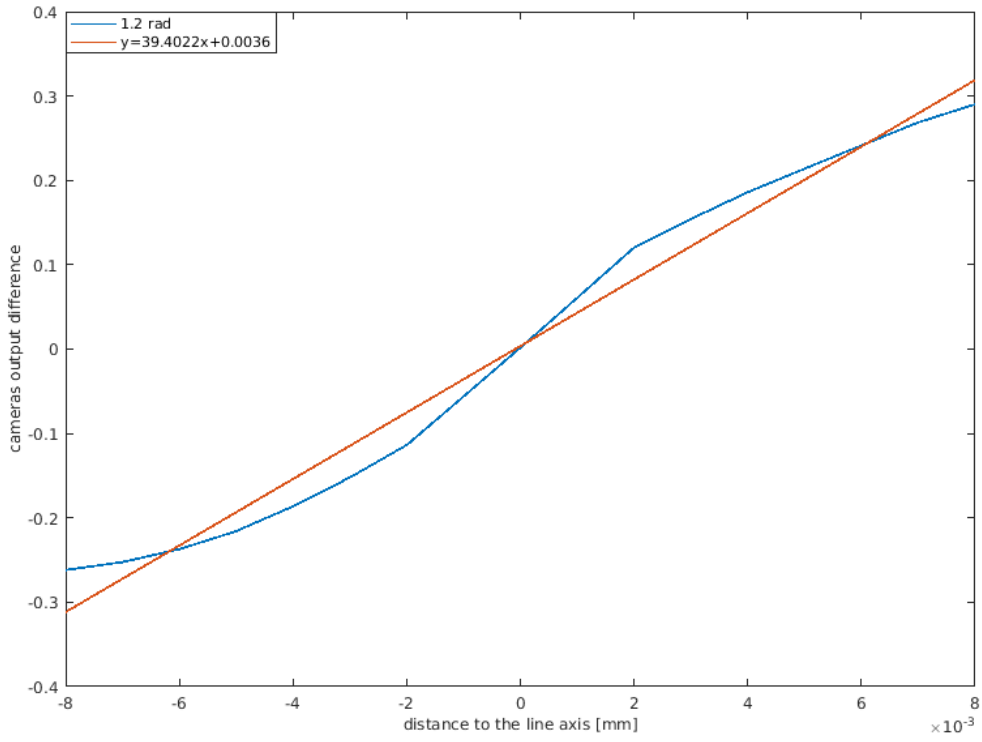


Figure 6.3.1. The almost lineal behaviour of response for x values from -0.008 m to 0.008

As it shown in chapter 5.2., the cameras output difference has a quasi lineal behaviour in the centre positions of the line [-0.008, 0.008] m, which has been modelled with a regression line:

$$y = 39.4022d + 0.0036 \quad \text{(Eqn. 6.12)}$$

As the desired value is d and y is the input value:

$$d = 0.0254y - 0.000091365 \quad \text{(Eqn. 6.13)}$$

In order to enlarge the x rang of values, Albert Costa in a previous Final Grade Project implemented the following logic:

$$if\ (ouput1 > output2\ \&\ output1 > output1_{maxdiff})$$

$$\{y\ = output1 + output2 + 2 * maxdiff\}$$

$$else\ if\ (ouput2 > output1\ \&\ output2 > output2_{mindiff})$$

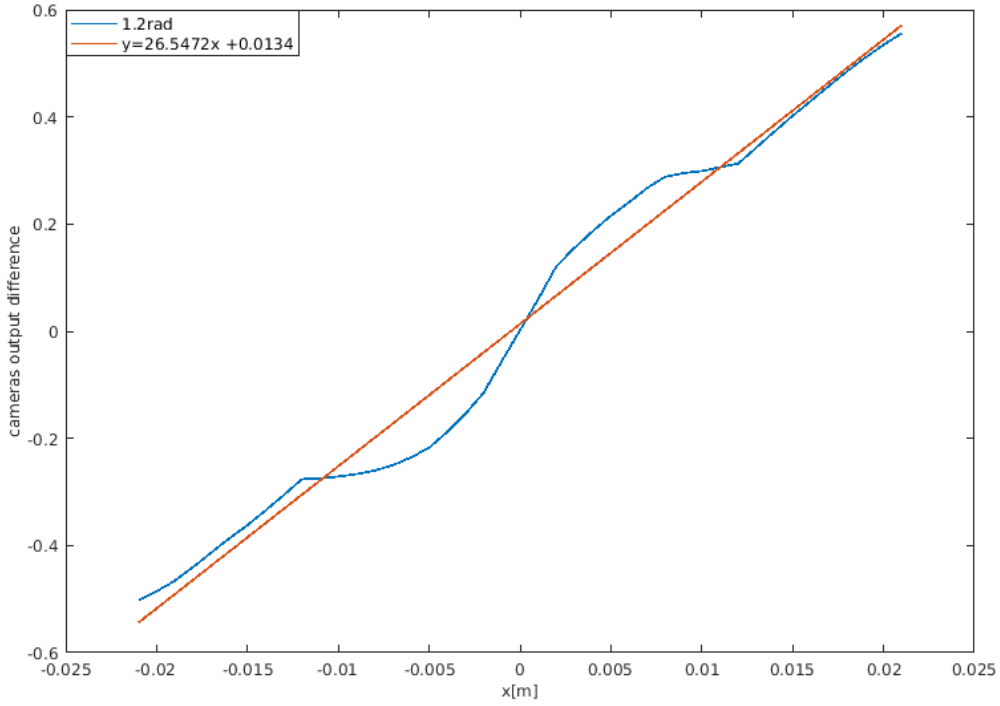
$$\{y\ = -(output1 + output2) - 2 * mindiff\}$$

$$else$$

$$\{y = output1 - output2\}$$

Where ouput1 = left camera output, output 2=right camera output, mindiff and maxdiff are the values at -0.008m and 0.008m.

Applying the conditions that characterize the regions of x values lower than -0.008m and higher than 0.008m, it is possible lo modify the y value.



The range of values is enlarged to [-0.021, 0.021] m. The regression is calculated again:

$$y = 26.5472x + 0.0134 \quad \text{(Eqn. 6.15)}$$

6.4. Pugins Architecture

A plugin is a chunk of code that is compiled as a shared library and inserted into the simulation. The plugin has direct access to all the functionality of Gazebo through the standard C++ classes. It let control almost any aspect of Gazebo. They have been used to implemented the control algorithm applied in the real vehicle. For this reason, have been necessary two sensor plugin to carry out the image processing and a model plugin to control the steering of the vehicle. The sensor plugins have two publisher nodes, which publish the average colour of the image frame in RGBA format under two topic called *Color_left* and *Color_right*. Plus, the sonar sensor has a default publisher node which publish the distance of the nearest object in its field of effect under the topic *SonarStamped.* The model's plugin receives the three topics through three subscriber nodes, which provide the input data to develop the vehicles control algorythm.

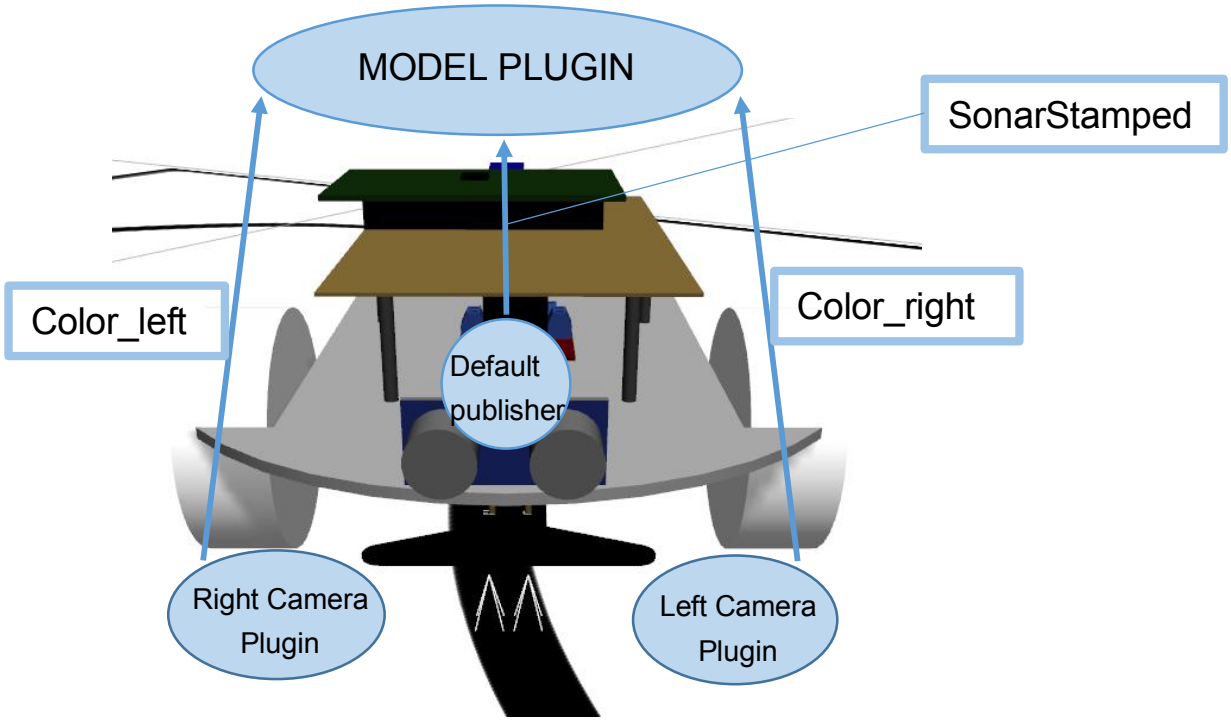


Fig 6.4. Diagram of plugins structure

6.5. Control algorithm

In the model plugin is where the tracking control is made. The control is based on a simple PID (Proportional Integral Derivative) controller, its main task is to reduce the error, in our case the distance to the line axis. The PID controller applies three basic mathematical functions as it names suggests to minimize the error. The distance is calculated applying the linearized equation to the difference of the two cameras outputs. The cameras outputs are received from the cameras plugins using the Gazebo communication explained in the previous chapter.

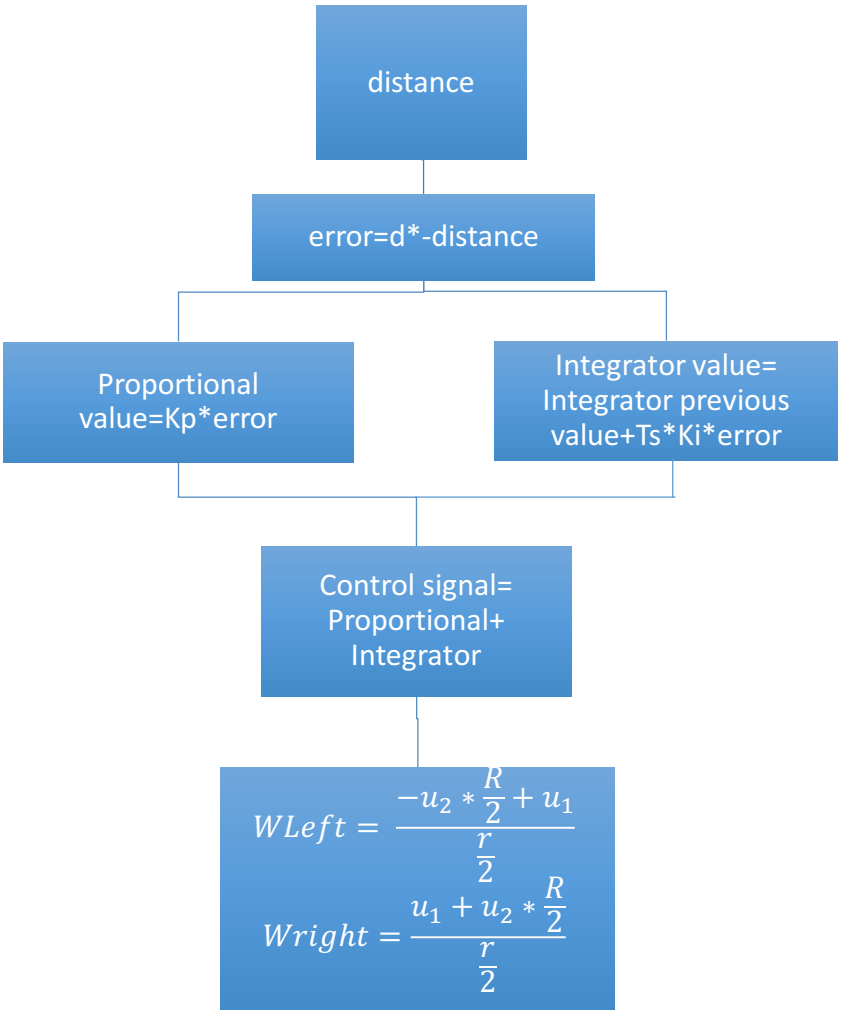


Fig. 6.5.1. Diagram of how the control algorythm is executed

The Wleft and the Wright are the command send to the joint control. In the experiencies made in the simulation it has been tested an ideal behaviour of the wheels with no delays. However, the delay can be set using a PID controller explained in chapter 5.4.

6.6. Step Time

In order to set the step time of the control system, two parameters are employed: the real time factor and the update rate of the cameras and the ultrasonic sensor. The real time factor of the simulator can be described as:

$$Real\ time\ factor = \frac{Simulation\ time[s]}{Real\ time\ [s]} \qquad \text{(Eqn. 6.16)}$$

That is to say, if the the real time factor is smaller than 1, the simulation will perform slower respect to the real time and if its bigger, it will perform faster.
To update rate of the cameras is the times per second that the cameras retrieve an image from the ground, which is able to do every time the scene is rendered (frames per second). In other words, the frames per second of the simulation graphical interface is the bottleneck of the camera's frequency. It has been tried to accelerate at the maximum the frames update of the

simulator, which could not be explicitly as Gazebo does not offers this functionality, but it has been noticed that if the resolution of the camera was reduced (lower number of bits), the cost-time to render the scene decreases, increasing the frames per second. The point is decreasing the image quality without losing the cameras functionality. To study the optimal image solution, the experience of the moving the vehicle through a straight black line has been made:

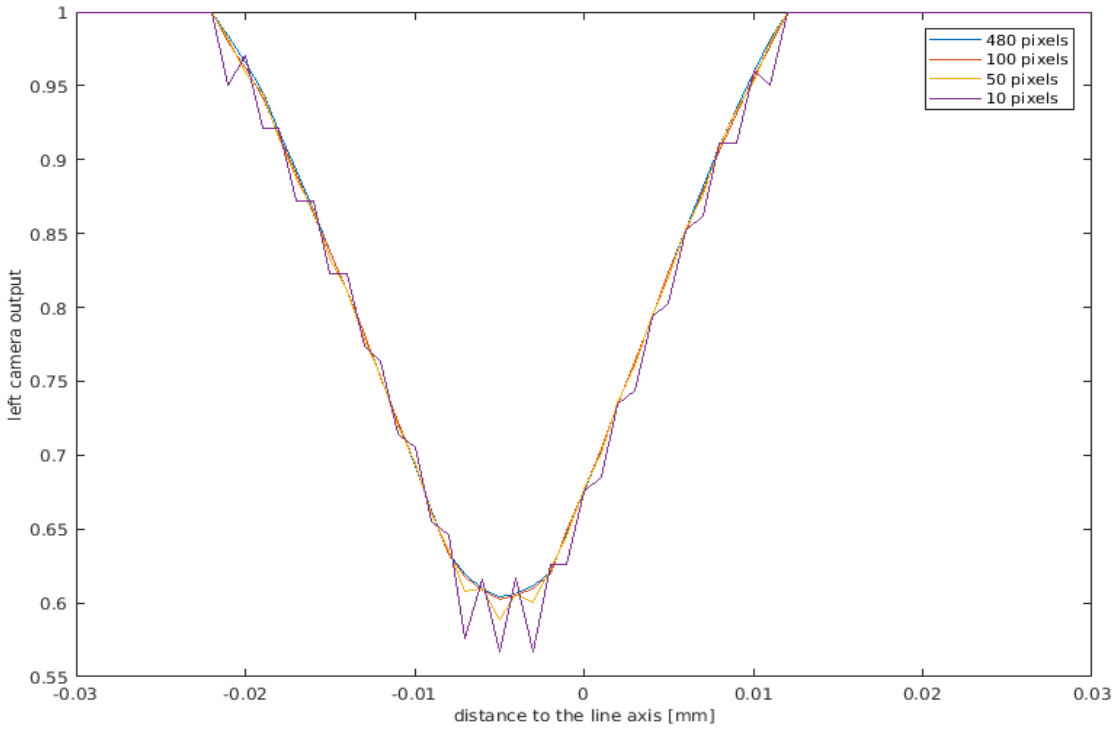


Figure 6.6.1 Brightness detection in function of the image quality

In the figure, has just been denoted only the left camera's output as both cameras behaviour is the same. It has been observed that for image resolutions lower than 100x100 pixels, the image obtained is blurred and the value of brightness obtained is not the desired. In this order of 100x100 pixels resolution, the sensor was able to perform at 300Hz more or less (the scene was rendered 300 times per second).

Once the real time factor and the update rate are described, playing with its values it is possible to have the cameras working in the desired step time. For example, if the camera frequency has to be 600, the simulation may be done at 100x100 pixel resolution and 0.5 real time factor.

$$600\ Hz = 300\ frames\ per\ real\ second * \frac{1\ simulation\ second}{0.5\ real\ second} = 600\ frames\ per\ simulation\ second \quad \text{(Eqn. 6.17)}$$

Concerning the sonar sensor update rate and the controller update rate, its bottleneck is higher and is defined in the own characteristics of the physics engine, which can be explicitly set in

the internal code of Gazebo. The default update rate is of 1000Hz and if it is increased, it has to be considered not to damage the correct working of the simulator. However, in both cases to have them working in the desired frequency without modifying the physics engine update rate. The first one can be set in SDF file description and the second one in the inner code of the model plugin.

In any case, to correctly reproduce the timing in the vehicle; the controller, the sonar sensor and the camera sensors have to work at the same frequency.

7. Simulation results

To prove the usability of the model and the simulation environment, some experiences have been made. The parameters with which the control study has been carried out are the followings: step time, Kp, Ki and lineal velocity. For instance, the range of the camera chosen has been [-0.008, 0.008] m and a resolution of 100x100 pixels. Has been studied two types of circuits.

7.1. Straight Line with a P controller

The initial conditions of the vehicle have been: static and positioned parallel to the line axis and in the middle of it. It has been tested for 0.04, 0.06, 0.08, 0.1s step time and for velocities of 0.01m/s, 0,1 m/s and 0,5 m/s. The Kps used have been 5, 50,100 ,150. The variables observed have been the estimated distance perceived by the line sensor, the x and y coordinates of the line sensor and angular velocity of the wheels. In all the cases, the response of the distance and the angular velocity observed has been of second order. This makes sense as the vehicle angular velocity is proportional to the estimated distance. The following is an example for ts=0.06, Kp=100 and v_1=0.1m/s:

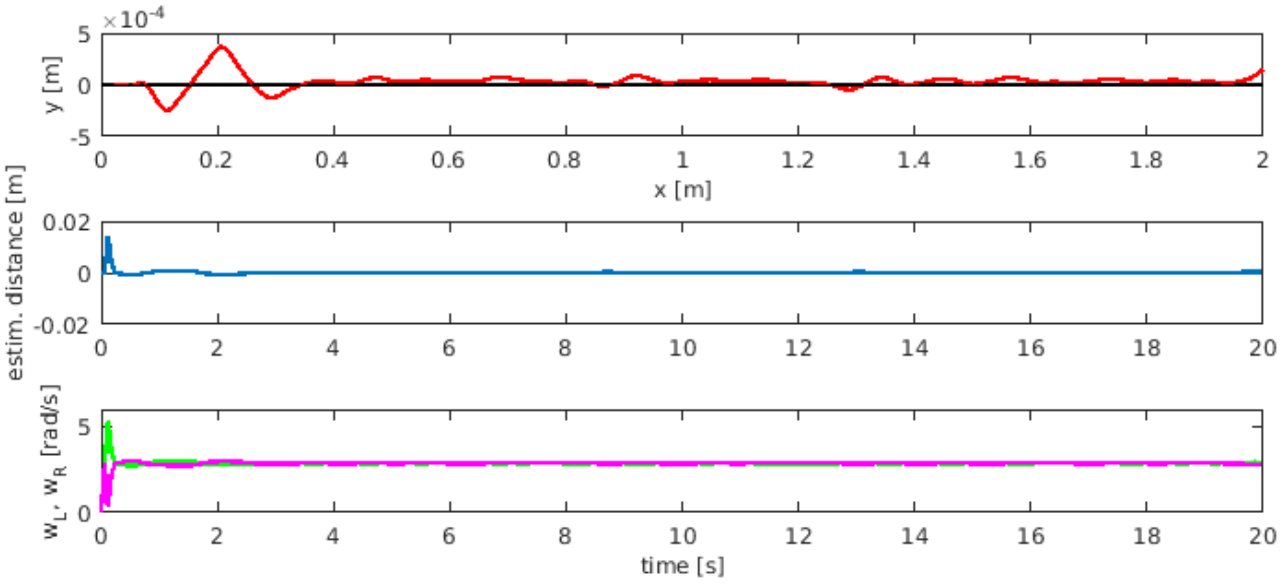


Fig. 7.1.1 Simulation test for Kp 100, ts 0.06 s and v1 0.1 m/s

The following figures show the simulation results keeping constant one of the three parameters and modifying the others.

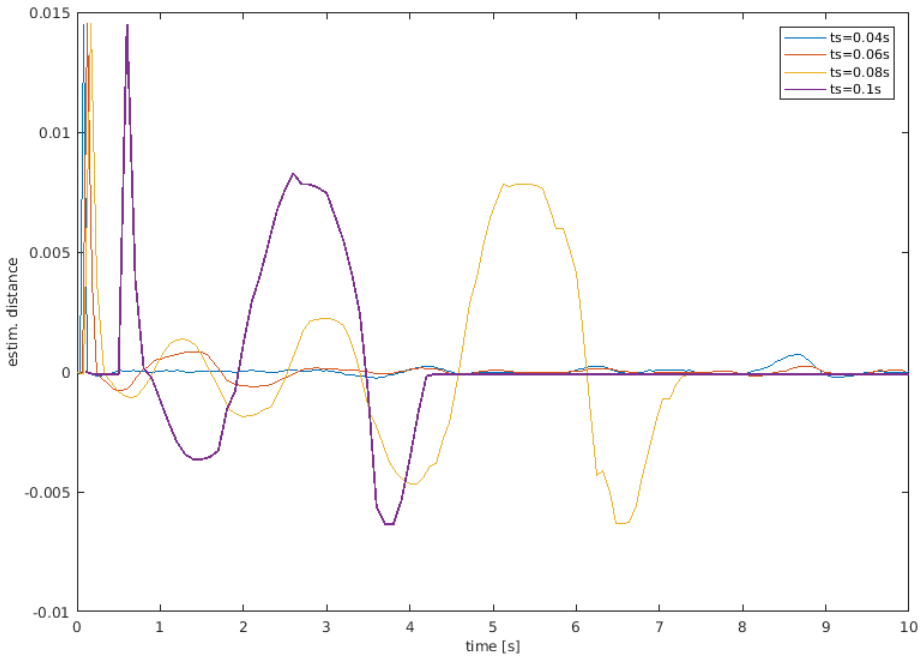


Fig. 7.1.2. Kp is 100 and v1 is 0,1 m/s, different step times

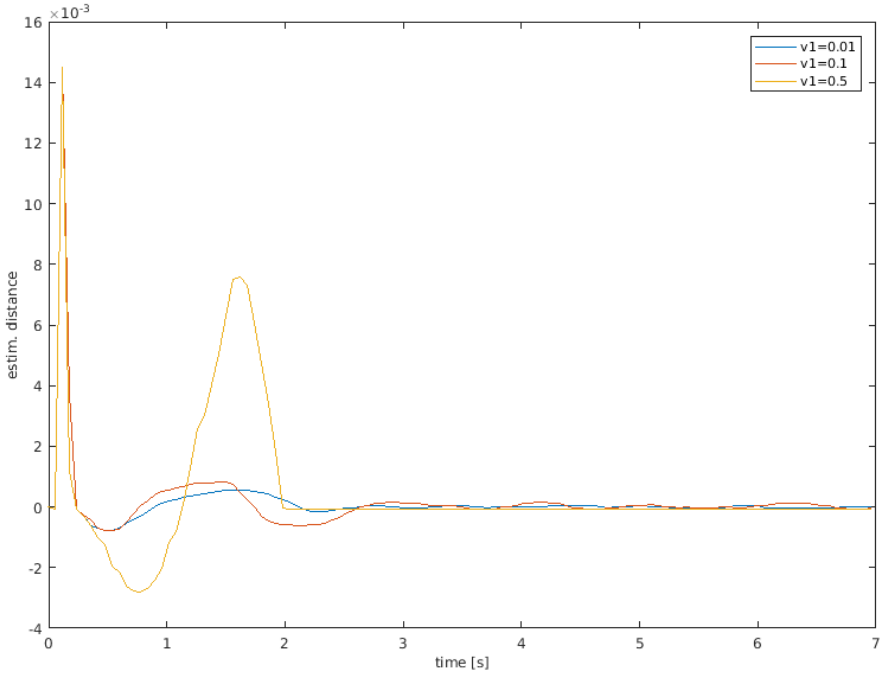


Fig. 7.1.3. Kp is 100 and ts is 0,06s, different v1.

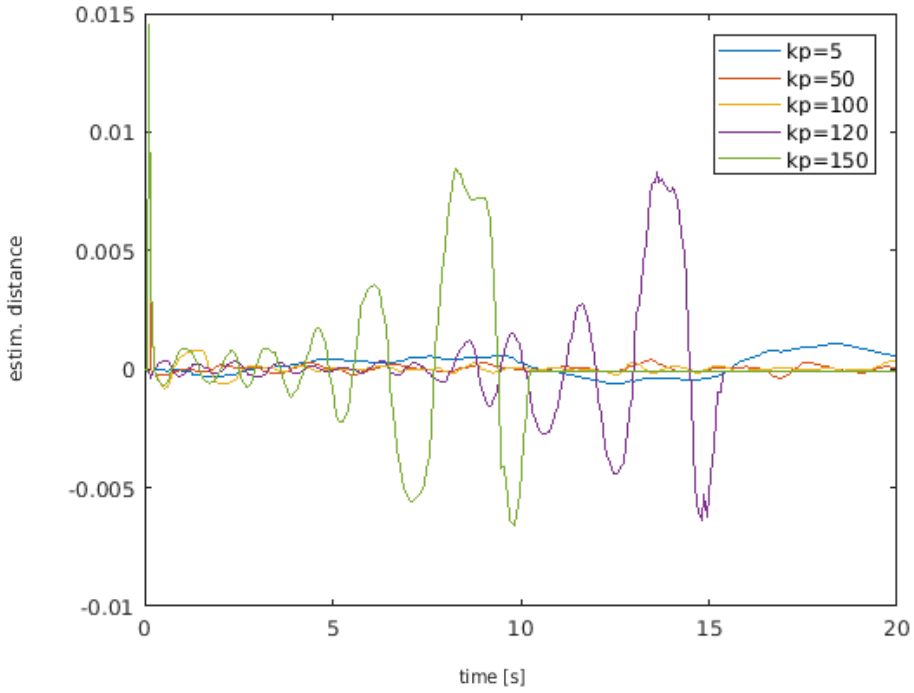


Figure 7.1.4. ts is 0,06s and v1 is 0,1 m/s, different Kp

7.2. Circular circuit with P controller

The initial conditions of the vehicle have been: static and positioned tangent to the line axis with the line sensor in the middle of the line. Working at the previous steps time in all the cases, the system became unstable at the speed of 0.1 m/s. So the step time was decreased. It has been tested for values of 0.01s and 0.002 s, which in the second case, made the simulation ran slowly as it was necessary 500Hz and the camera runs at 300Hz. The results are the followings for a Kp=50.

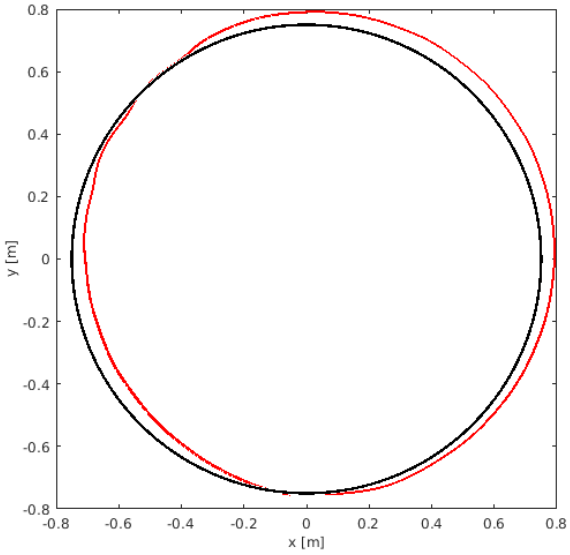


Fig. 7.2.1 Positions of the line sensor and the path to follow

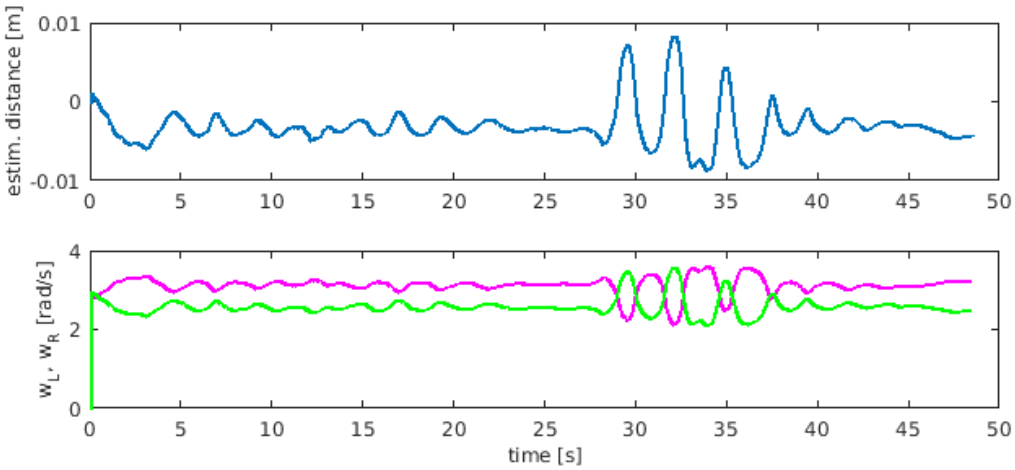


Figure 7.2.2. The vehicle and the wheels response

As it can be observed, the system is asymptotically stable till it reaches 200 grades of circumference, where some turbulence appears. In this case the turbulence gets near to the 0.008m distance from the line axis, which is the limit of the rang of values where the line sensor gives a correct feedback of the position of the vehicle. For higher steps times, the system became unstable in this region. It is a second order response.

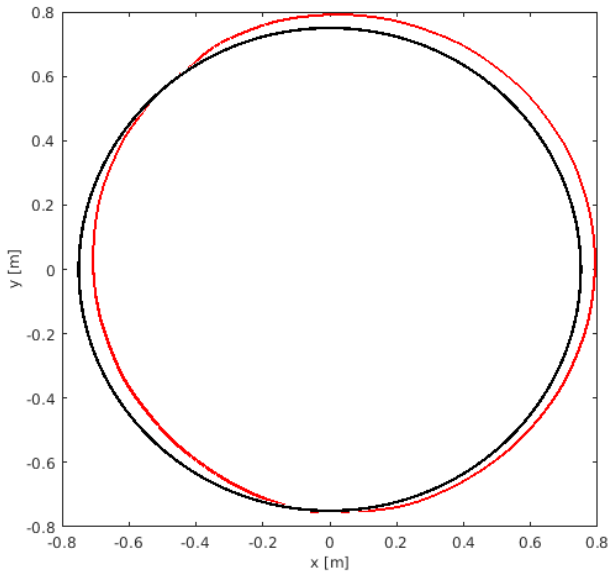


Fig. 7.2.3 Positions of the line sensor and the path to follow

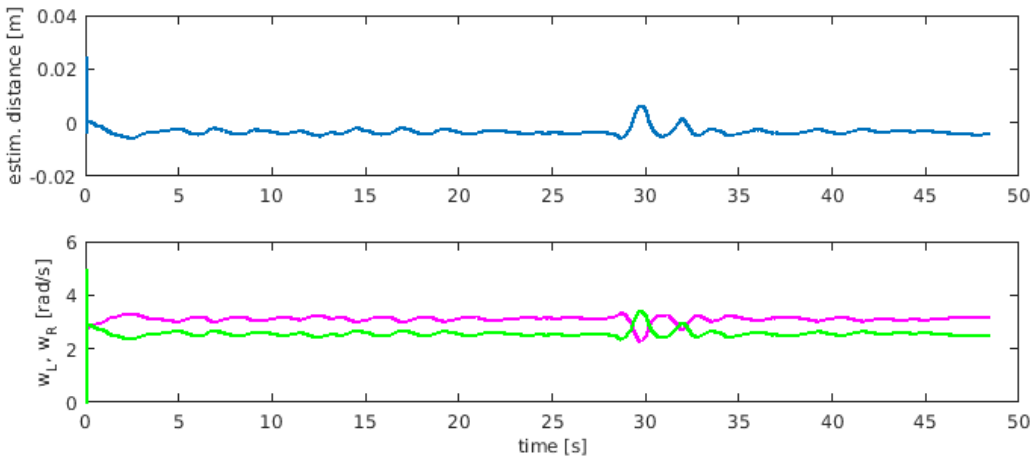


Fig. 7.2.4. The vehicle and the wheels response

As it can be observed, for a lower step time the vehicle seems to be stable. In the same region, which corresponds to the moment the vehicle transition from following the circle in the exterior side to the interior side, appears turbulence. However, the vehicle turns to stable after a while. The response is too of second order.

7.3. Circular circuit with PI controller

The previous experience reflect that an Integrator is not necessary. However, it has been of interest see if its implementation affects on the previous results. It has been studied for a Kp=50, Ki=0.001, ts=0.01s and v1=0.1m/s.

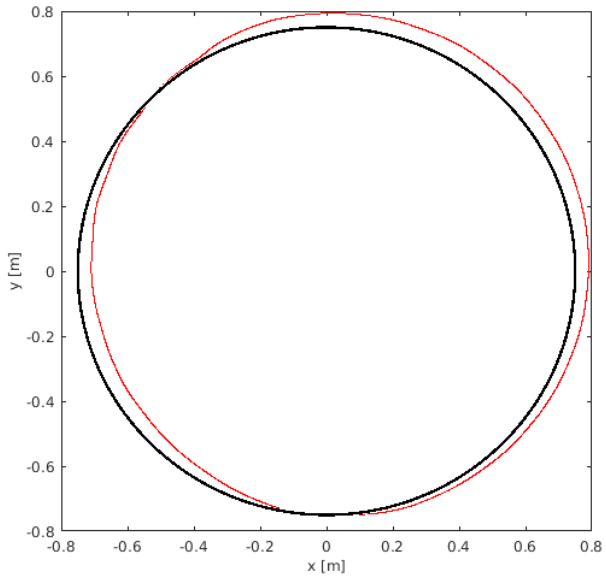


Fig. 3.1.Positions of the line sensor and the path to follow

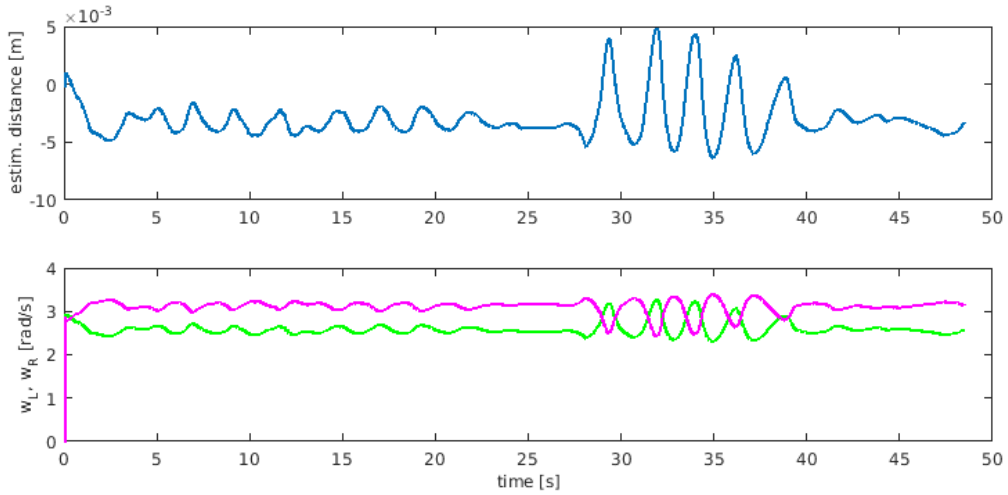


Figure 3.2 The vehicle and the wheels response

As far as can be observed, it has the same oscillations as the proportional controller but the amplitude is lower.

Conclusions

The purpose of this study was to develop a reliable environment for simulation and control of mobile robots using Gazebo software. The simulation results prove the usability of the model and the environment to reproduce the performance of a line tracker vehicle.

The core of the work has been the design of the control code using two camera sensor plugin and a model plugin where the control strategy is applied. It has been very useful the wide range of features supplied by Gazebo to try to simulate as more accurate as possible the line tracker robot.

The principal handicap has been Gazebo does not have a user-friendly interface to model your robots and is basically all programmed in its core code, which means that the user has to get used to the the programming structure, which is based on an object-oriented application.

Besides, the image processing bottleneck has been denoted as an important problem to carry out real-time simulation, as the cameras are not able to operate in high frequencies.

To conclude, in further developments would be recommended to implement more realistic vehicle traction with delays and friction in the transmission chain and to create more user-friendly interface so as to ease the simulation preparation and setting, as well as, an executable to plot the desired variable of the scene with more rapidness.

Acknowledgements

I would like to express my gratitude to my tutors Arnau Dòria and Jan Rosell for their help along the project and to give me the opportunity to work in institute of Management and Control in the Robotics Laboratory, and also to Nestor García to give me a hand in any kind of doubt that came across the development of the project.

Bibliografy

Referències bibliogràfiques

[1] Prats Matinho, Ivan. *Control design and implementation for a line tracker vehicle*. End of Degree Project of Physics Engineering(2016)

[2] Riera Seguí, Antoni. *Disseny I implementació d'un system de comunicacions WiFi per una xarax de vehicle autònoms.* End of Degree Project UPC(2016)

[3] Yulin Zhang, Daehie Hong, Jae H. Chung, and Steven A. Velinsky, *Dynamic Model Based Robust Tracking Control of a Differentially Steered Wheeled Mobile Robot.* Proc of the American Control Conference (1998).

[4] Kenta Takaya, Toshinori Asai, Valeri Kroumov, Florentin Smarandache, *Simulation environment for mobile robots testing using ROS and GAZEBO*. 20th International Conference on System Theory, Control and Computing. (2016)

[5] Thomio Watanabe, Gustavo Nenes, Tiago Trocoli, Marcos Reis, Jan Albiez *. *The rock-Gazebo integration and Real-Time AUV Simulation.* Latin American Robots Symposium (2015).

[6] Costa Ruiz, Albert. *Design of controllers and its implementation fro a line tracker vehicle. .* End of Degree Project UPC(2017).

Annex

I. Mesh Design

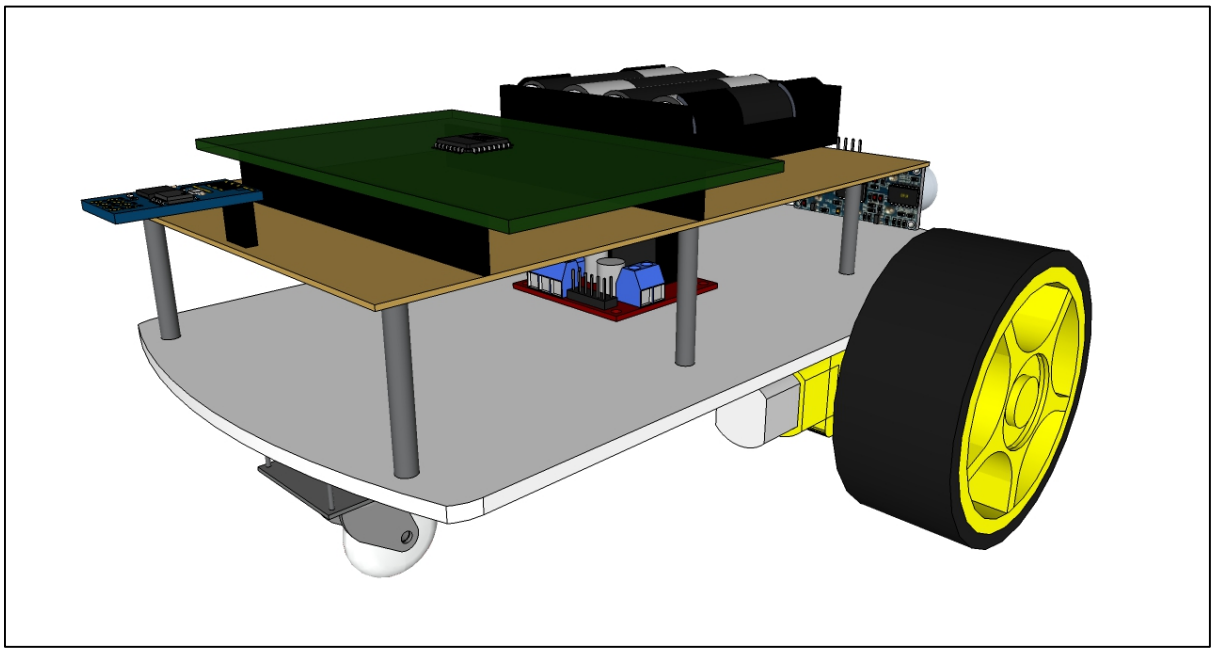


Fig. I.1 Front perspective view

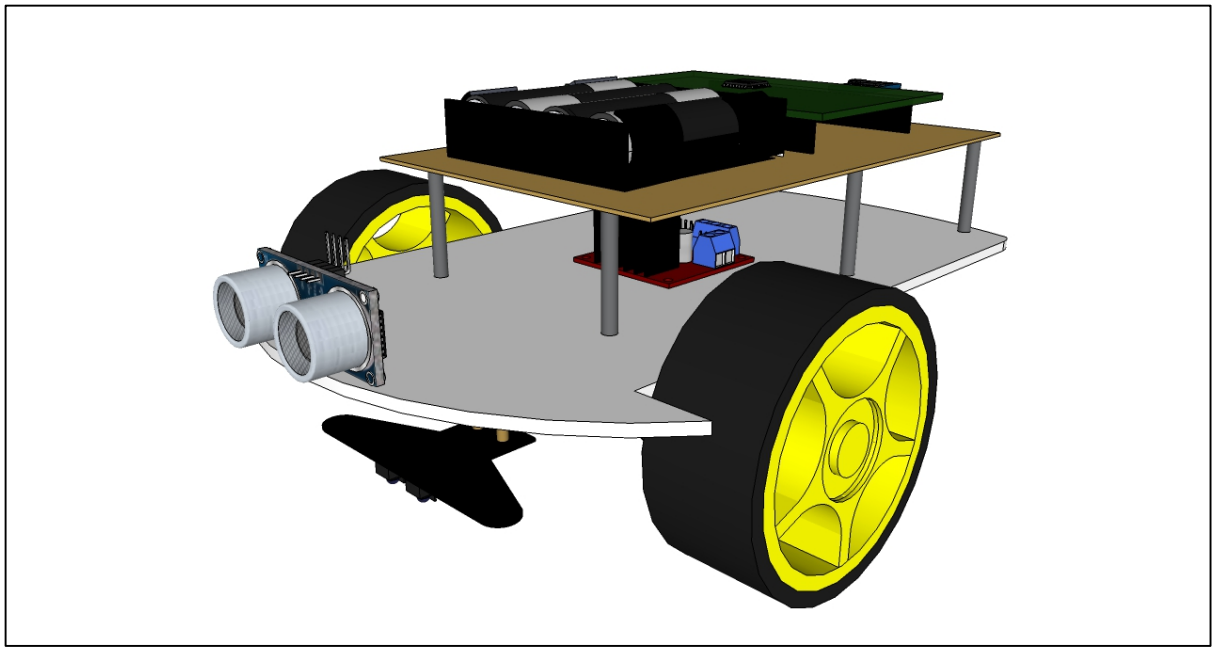


Fig. I.2. Back perspective view

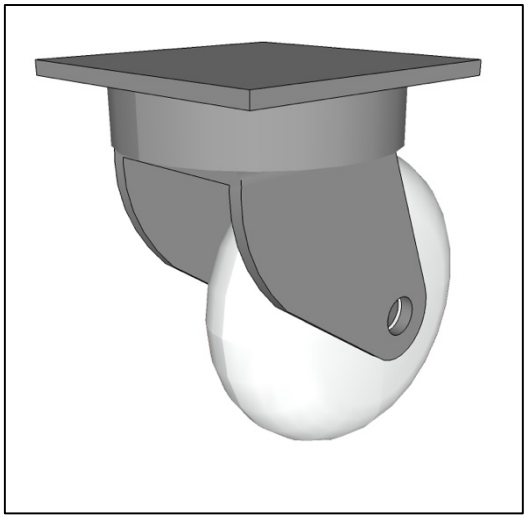


Fig. I.3. Passive caster wheel

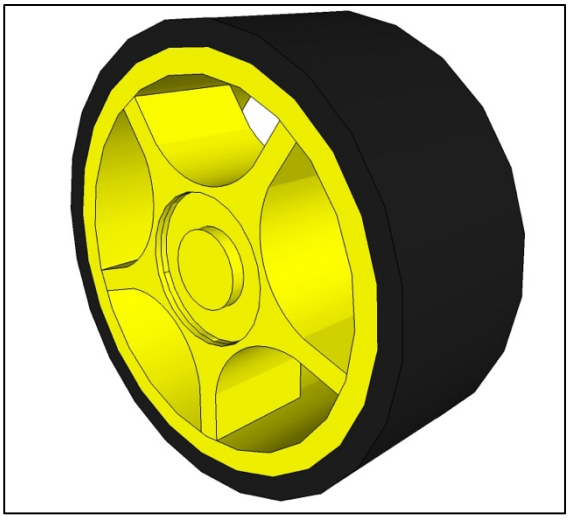


Fig. I.4. Traction wheel

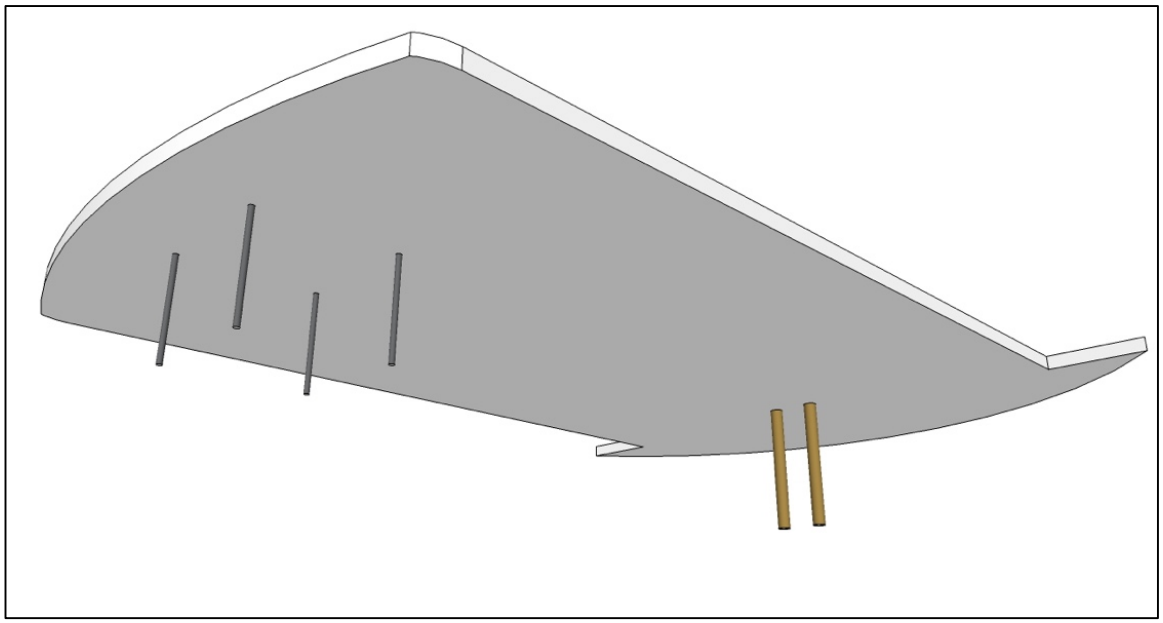


Fig. I.5. Chassis

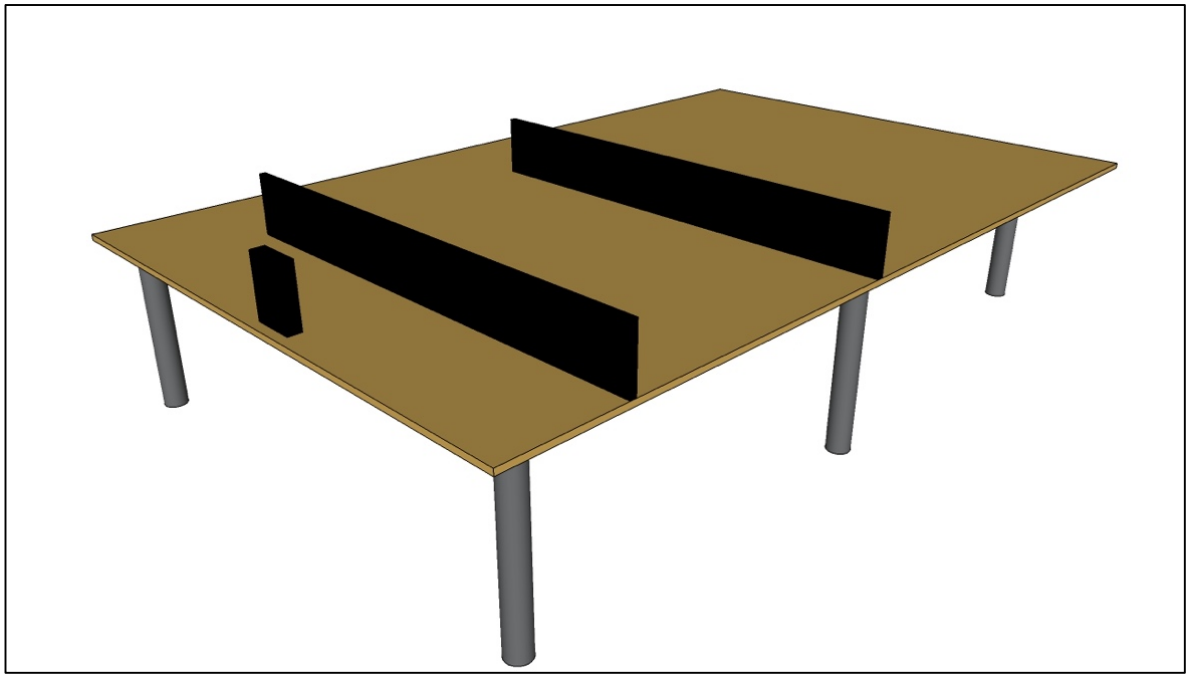


Fig. I.6. Base support for the microcontroller protoboard, the battery holder and the Wifi module

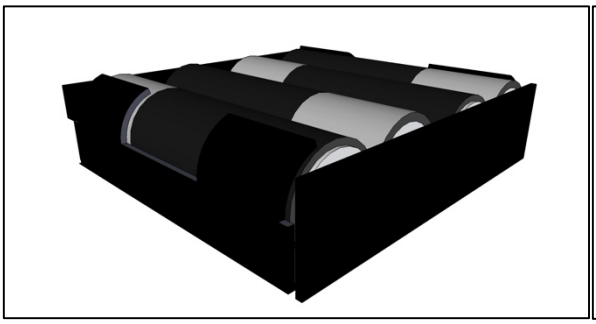


Fig. I.7. Battery holder

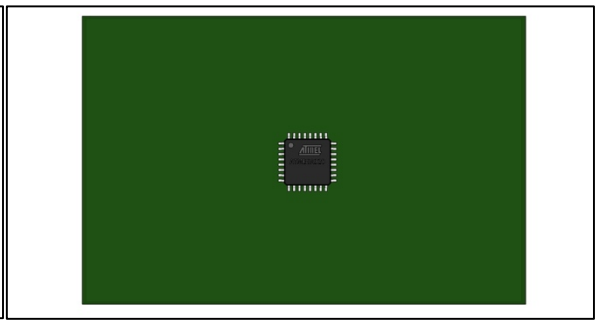


Fig. I.8. Microcontroller protoboard

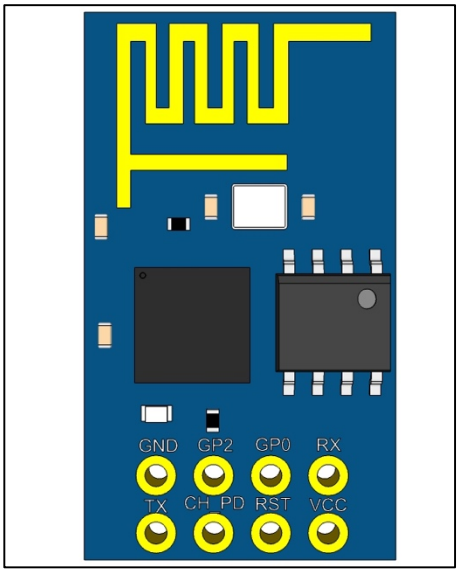


Fig. I.9. Wifi module

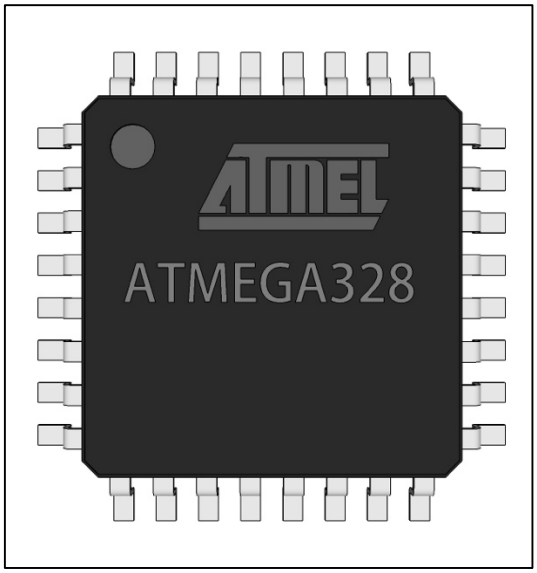


Fig. I.10. Microcontroller ATMEGA328

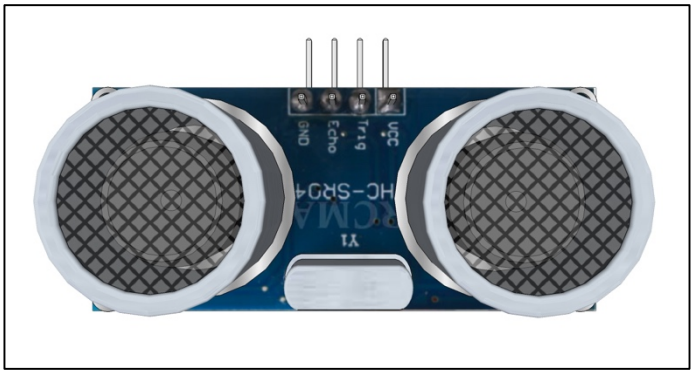


Fig. I.11. Ultrasonic sensor HC-SR04

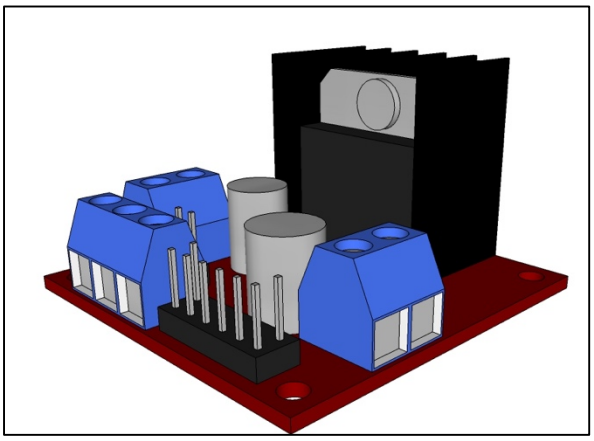


Fig. I.12. Driver L298N

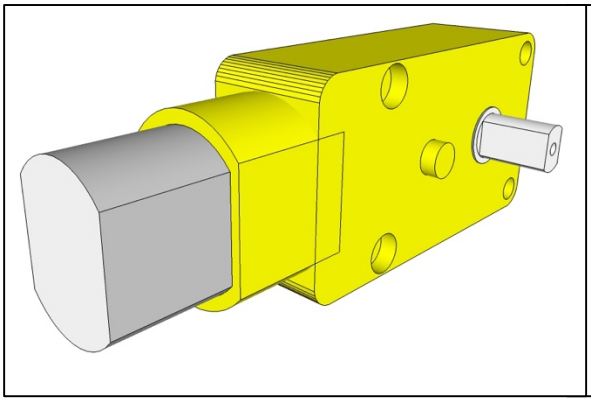


Fig. I.13. Motor DC

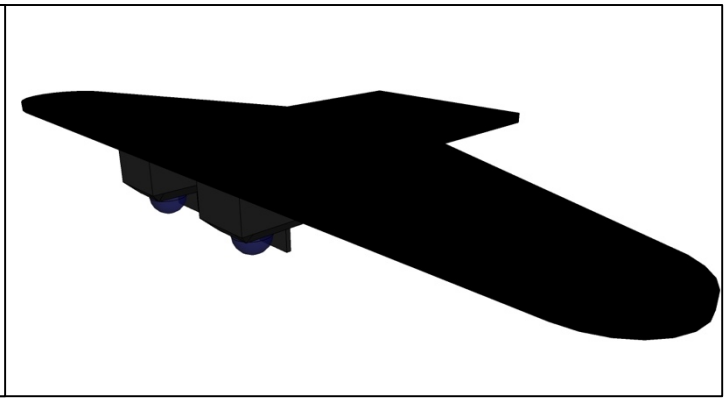


Fig. I.14. Line sensor LRE-F22

II.Code

```
#ifndef _TESTSENSOR_HH_
#define _TESTSENSOR_HH_

#include <iostream>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/math/gzmath.hh>
#include <thread>
#include <ros/ros.h>
#include <ros/callback_queue.h>
#include <ros/subscribe_options.h>
#include <std_msgs/Float32.h>
#include <gazebo/msgs/msgs.hh>
#include <fstream>
#include <string>
#include <functional>
#include <mutex>
#include <gazebo/physics/physics.hh>
#include <gazebo/physics/World.hh>
#include "gazebo/physics/PhysicsTypes.hh"


namespace gazebo
{

  struct TestPrivate;
  class TestSensor: public ModelPlugin
  {
    public: TestSensor();

    /// \brief Destructor

    public: virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf);

        public: virtual void cb1(ConstColorPtr &_msg1);
        public: virtual void cb2(ConstColorPtr &_msg2);
        public: virtual void cb4(ConstSonarStampedPtr &_msg4);

    public: virtual void OnUpdate();

        private: void OnMsg(ConstVector3dPtr &_msg);

        public: void OnRosMsg(const std_msgs::Float32ConstPtr &_msg);

        private: void QueueThread();

        private: math::Pose pose;
        private: double idx;
        private: physics::ModelPtr model;
        //private: event::ConnectionPtr updateConnection;
        private: transport::NodePtr node1;
        private: transport::NodePtr node2;
        private: transport::NodePtr node3;
        private: transport::NodePtr node4;
        private: transport::SubscriberPtr sub1;
        private: transport::SubscriberPtr sub2;
        private: transport::SubscriberPtr sub3;
        private: transport::SubscriberPtr sub4;
        private: float brightness1;
        private: float brightness2;
        private: long saveCount;
        private: bool condition1;
        private: bool condition2;
        private: float array1[40]={};
        private: float array2 [40]={};
        private: float u1;
        private: float Kp;
        private: float Ki;
        private: float distance;
```

```
        private: physics::JointPtr joint1;
        private: physics::JointPtr joint2;
        private: common::PID pid;
        private: transport::NodePtr node;
        private: float Kielement;
        private: int a;
    private: std::ofstream file1;
    private: std::ofstream file2;
    private: std::ofstream file3;
    private: std::ofstream file4;
    private: std::ofstream file5;
    protected: gazebo::physics::WorldPtr world;
    protected: common::Time lastUpdateTime;

        /// \brief A ROS subscriber
    private: ros::Publisher rosPub;

        /// \brief A node use for ROS transport
        private: std::unique_ptr<ros::NodeHandle> rosNode;

        private:  std_msgs::Float32 msg;
    private: std::unique_ptr<TestPrivate> dataPtr;

        /// \brief A ROS callbackqueue that helps process messages
        /*private: ros::CallbackQueue rosQueue;

        /// \brief A thread the keeps running the rosQueue
        private: std::thread rosQueueThread;*/

  };
}
#endif
```

```
#include "TestSensor.hh"

using namespace gazebo;

// Tell Gazebo about this plugin, so that Gazebo can call Load on this plugin.

GZ_REGISTER_MODEL_PLUGIN(TestSensor)

namespace gazebo
{

  struct TestPrivate
  {

    public: event::ConnectionPtr updateConnection;

    /// \brief Pointer to the model.
    public: float brightness2;
    public: float brightness1;

    /// \brief Update mutex.
    public: std::mutex mutex;
    };
}

TestSensor::TestSensor() : pose(0.0,-0.0,0.0,0.0,0.0,0.0), idx(-0.025), node1 (new
transport::Node()), node2 (new transport::Node()), node3 (new transport::Node()), node4
(new transport::Node()), saveCount(0), u1(0.05), Kp (50), Ki(0.00), a(1),
Kielement(0),distance(0),dataPtr(new TestPrivate)
{

    this->node1->Init();
    this->node2->Init();
    this->node3->Init();
    this->node4->Init();
    this->file1.open ("save1.txt");
    this->file2.open ("save2.txt");
    this->file3.open ("save3.txt");
    this->file4.open ("save4.txt");
    this->file5.open ("save5.txt");
}

void TestSensor::cb1(ConstColorPtr &_msg1)
{
    this->dataPtr->brightness1=(0.2126*(_msg1->r()) + 0.7152*(_msg1->g()) +
0.0722*(_msg1->b()));

    //gzmsg << this->brightness1;

    //std::cout << std::endl;

}

void TestSensor::cb2(ConstColorPtr &_msg2)
{
    this->dataPtr->brightness2=(0.2126*(_msg2->r()) + 0.7152*(_msg2->g()) +
0.0722*(_msg2->b()));

    //gzmsg << this->brightness2;

    //std::cout << std::endl;


    /*if(this->idx <= 0.020 )
    {
        //this->array1[this->saveCount]=this->brightness1;

        //this->array2[this->saveCount]=this->brightness2;
```

```
        //gzmsg << this->array1[this->saveCount];

        //gzmsg << this->saveCount;

        this->model->SetWorldPose(pose,true,true);

        ++(this->saveCount);

        this->idx=(this->idx)+0.001;

        this->pose.Set(0.0,double(idx),0.0,0.0,0.0,0.0);
    }

    //else

    {
        std::cout << '[';
        for(int i=0 ; i< 45 ; i=i+1)
        {
            std::cout << this->array1[i] << ",";
        }
        std::cout << ']'<< ';' << std::endl;
        std::cout << '[';
        for(int y=0 ; y< 45 ; y=y+1)
        {
            std::cout << this->array2[y] << ',';
        }
        std::cout << ']'<< ';'<< std::endl;

    //this->sub1->Unsubscribe();
    //this->sub2->Unsubscribe();

    }*/

}

void TestSensor::cb4(ConstSonarStampedPtr &_msg4)
{
    double range=_msg4->sonar().range();
    if(range<0.1)
        {
            this->a=0;
        }
    else
        {
        this->a=1;
        }

}



void TestSensor::OnMsg(ConstVector3dPtr &_msg)
{

    this->u1=_msg->x();

    if (_msg->y()!=0.0)
    {
            this->Kp=_msg->y();
    }
    if (_msg->z()!=0.0)
    {
            this->Ki=_msg->z();
    }
}

void TestSensor::Load(physics::ModelPtr _model, sdf::ElementPtr _sdf)
```

```
{

    //Initialize ros, if it has not already bee initialized.
    if (!ros::isInitialized())
    {
        int argc = 0;
        char **argv = NULL;
        ros::init(argc, argv, "gazebo_client",
              ros::init_options::NoSigintHandler);
    }

    this->rosNode.reset(new ros::NodeHandle("gazebo_client"));

    /*// Create our ROS node. This acts in a similar manner to
    // the Gazebo node
    this->rosNode.reset(new ros::NodeHandle("gazebo_client"));

    // Create a named topic, and subscribe to it

    ros::SubscribeOptions so =
        ros::SubscribeOptions::create<std_msgs::Float32>(
              "/" + this->model->GetName() + "/vel_cmd",
              1,
              boost::bind(&TestSensor::OnRosMsg, this, _1),
              ros::VoidPtr(), &this->rosQueue);
    this->rosSub = this->rosNode->subscribe(so);
    // Spin up the queue helper thread.
    this->rosQueueThread =
        std::thread(std::bind(&TestSensor::QueueThread, this));*/

    rosPub = rosNode->advertise<std_msgs::Float32>("distance", 1000);
  this->lastUpdateTime = common::Time(0.0);

    this->model=_model;

    this->joint1 = this->model->GetJoints()[0];
  this->joint2 = this->model->GetJoints()[1];

    //this->pose.Set(-0.04616,-0.08303,0.0,0.0,0.0,1.06);
    this->pose.Set(-0.095,0,0.0,0.0,0.0,0);
    this->model->SetWorldPose(pose,true,true);

  this->world = physics::get_world("default");
    this->file1 <<0;
    this->file1 << ',';
    this->file2 <<0;
    this->file2 << ',';
    this->file3 <<0;
    this->file3 << ',';
    this->file4 <<0;
    this->file4 << ',';
    this->file5 <<0;
    this->file5 << ',';




    common::Time::MSleep(10000);

    //this->pose.Set(0.0,double(idx),0.0,0.0,0.0,0.0);

    this->sub1 = this->node1->Subscribe("~/Color_left", &TestSensor::cb1, this);

    this->sub2 = this->node2->Subscribe("~/Color_right", &TestSensor::cb2, this);

    this->sub3 = this->node3->Subscribe("~/Message", &TestSensor::OnMsg, this);

    this->sub4 = this->node4->Subscribe("~/LineTracker/line_tracker/chassis/ultrasound/
sonar", &TestSensor::cb4, this);
```

```
  this->pid = common::PID(this->Kp, this->Ki, 0);

    this->model->GetJointController()->SetVelocityPID(
     this->joint1->GetScopedName(), this->pid);
    this->model->GetJointController()->SetVelocityPID(
     this->joint2->GetScopedName(), this->pid);

    this->joint1->SetVelocityLimit(0,18.46);
    this->joint2->SetVelocityLimit(0,18.46);

    this->dataPtr->updateConnection = event::Events::ConnectWorldUpdateBegin(
         std::bind(&TestSensor::OnUpdate, this));


}

    /// \brief Handle an incoming message from ROS
    /// \param[in] _msg A float value that is used to set the velocity
    /// of the Velodyne.
    /*void TestSensor::OnRosMsg(const std_msgs::Float32ConstPtr &_msg)
    {
            this->model->GetJointController()->SetVelocityTarget(
         this->joint1->GetScopedName(), _msg->data);
    this->model->GetJointController()->SetVelocityTarget(
         this->joint2->GetScopedName(), _msg->data);untitled.
    }

    /// \brief ROS helper function that processes messages
    void TestSensor::QueueThread()
    {
        static const double timeout = 0.01;
        while (this->rosNode->ok())
        {
          this->rosQueue.callAvailable(ros::WallDuration(timeout));
        }
    }*/

void TestSensor::OnUpdate()
{
  // Move the model.
  std::lock_guard<std::mutex> lock(this->dataPtr->mutex);


  if (this->world->GetSimTime() - this->lastUpdateTime >= 0.06)
    {

    float y=(this->dataPtr->brightness1)-(this->dataPtr->brightness2);
    this->distance= -0.000091365+0.0254*y;
    float error =-1*this->distance;
    float Kpelement=(this->Kp)*error;
    this->Kielement=this->Kielement+0.01*this->Ki*error;
    float u2= Kpelement+this->Kielement;
    float wLref=(u2*0.06*-1.0+(this->u1))*(this->a)/float(0.035);
    float wRref=((this->u1)+u2*0.06)*(this->a)/float(0.035);

    this->joint1->SetVelocity(0, wLref);

    this->joint2->SetVelocity(0, wRref);

    this->lastUpdateTime = this->world->GetSimTime();

    this->msg.data = distance;
  this->rosPub.publish(this->msg);

    if( this->world->GetSimTime()<240)
            {

        this->file1 << (this->model->GetWorldPose().pos.x)+cos(this->model-
```

```
>GetWorldPose().rot.z)*(0.045);
            this->file1 <<',';

        this->file2 << (this->model->GetWorldPose().pos.y)+sin(this->model-
>GetWorldPose().rot.z)*(0.045);
            this->file2 <<',';

          this->file3 << this->distance;
            this->file3 <<',';

          this->file4 << this->joint1->GetVelocity(0);
            this->file4 <<',';

          this->file5 << this->joint2->GetVelocity(0);
            this->file5 <<',';

            }

    else
    {
        this->file1 << (this->model->GetWorldPose().pos.x)+cos(this->model-
>GetWorldPose().rot.z)*(0.045);
        this->file2 << (this->model->GetWorldPose().pos.y)+sin(this->model-
>GetWorldPose().rot.z)*(0.045);
          this->file3 << this->distance;
          this->file4 << this->joint1->GetVelocity(0);
          this->file5 << this->joint2->GetVelocity(0);
            this->file1.close();
            this->file2.close();
            this->file3.close();
            this->file4.close();
            this->file5.close();
    }
    this->saveCount=this->saveCount+1;
    }


}
```

```
/*
 * Copyright (C) 2012 Open Source Robotics Foundation
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
*/
#include <gazebo/gazebo.hh>
#include <gazebo/plugins/CameraPlugin.hh>
#include <gazebo/common/Image.hh>
#include <ros/ros.h>
#include <ros/callback_queue.h>
#include <ros/subscribe_options.h>
#include <std_msgs/Float32.h>
#include <string>


namespace gazebo
{
  class CameraDump1 : public CameraPlugin
  {
    public: CameraDump1() : CameraPlugin(), saveCount(0), node (new transport::Node())
        {
            this->node->Init();
            this->pub= node->Advertise<msgs::Color>("~/Color_left");
        }
    public: void Load(sensors::SensorPtr _parent, sdf::ElementPtr _sdf)
    {
      // Don't forget to load the camera plugin
      CameraPlugin::Load(_parent, _sdf);
            this->rosNode3.reset(new ros::NodeHandle("gazebo_client"));
            this->rosPub3 = rosNode3->advertise<std_msgs::Float32>("color1", 1000);

    }

    // Update the controller
    public: void OnNewFrame(const unsigned char *_image,
        unsigned int _width, unsigned int _height, unsigned int _depth,
        const std::string &_format)
    {

        this->render.SetFromData(_image, _width, _height, this-
>render.ConvertPixelFormat(_format));
        //this->brightness=(0.2126*(this->render.GetAvgColor().r) + 0.7152*(this-
>render.GetAvgColor().g) + 0.0722*(this->render.GetAvgColor().b));
        msgs::Set(&output,this->render.GetAvgColor());
        //std::cout << "Waiting for connection1 " << std::endl;
        this->pub->WaitForConnection();
        this->pub->Publish(output);
        this->msg3.data = 213.1;
    this->rosPub3.publish(this->msg3);



          /*if (this->saveCount < 10)
      {
                gzmsg<< this->brightness;
        this->parentSensor->Camera()->SaveFrame(
            _image, _width, _height, _depth, _format, tmp);
        gzmsg << "Saving frame [" << this->saveCount
```

```
              << "] as [" << tmp << "]\n";
        this->saveCount++;
      }*/
    }

    private: int saveCount;
        private: common::Image render;
        private: float brightness;
        private: transport::NodePtr node;
        private: transport::PublisherPtr pub;
        private: msgs::Color output;
        private: std::unique_ptr<ros::NodeHandle> rosNode3;
    private: ros::Publisher rosPub3;
        private: std_msgs::Float32 msg3;
  };

  // Register this plugin with the simulator
  GZ_REGISTER_SENSOR_PLUGIN(CameraDump1)
}
```

```
/*
 * Copyright (C) 2012 Open Source Robotics Foundation
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
*/
#include <gazebo/gazebo.hh>
#include <gazebo/plugins/CameraPlugin.hh>
#include <gazebo/common/Image.hh>

namespace gazebo
{
  class CameraDump2 : public CameraPlugin
  {
    public: CameraDump2() : CameraPlugin(), saveCount(0), node (new transport::Node())
        {
            this->node->Init();
            this->pub= node->Advertise<msgs::Color>("~/Color_right");
        }
    public: void Load(sensors::SensorPtr _parent, sdf::ElementPtr _sdf)
    {
      // Don't forget to load the camera plugin
      CameraPlugin::Load(_parent, _sdf);

    }

    // Update the controller
    public: void OnNewFrame(const unsigned char *_image,
        unsigned int _width, unsigned int _height, unsigned int _depth,
        const std::string &_format)
    {
      char tmp[1024];
      snprintf(tmp, sizeof(tmp), "/tmp/%s-%04d.jpg",
          this->parentSensor->Camera()->Name().c_str(), this->saveCount);


        this->render.SetFromData(_image, _width, _height, this-
>render.ConvertPixelFormat(_format));
        //this->brightness=(0.2126*(this->render.GetAvgColor().r) + 0.7152*(this-
>render.GetAvgColor().g) + 0.0722*(this->render.GetAvgColor().b));
        msgs::Set(&output,this->render.GetAvgColor());
        //std::cout << "Waiting for connection2 " << std::endl;
        this->pub->WaitForConnection();
        this->pub->Publish(output);



          /*if (this->saveCount < 10)
      {
                gzmsg<< this->brightness;
        this->parentSensor->Camera()->SaveFrame(
            _image, _width, _height, _depth, _format, tmp);
        gzmsg << "Saving frame [" << this->saveCount
              << "] as [" << tmp << "]\n";
        this->saveCount++;
      }*/
    }

    private: int saveCount;
```

```
        private: common::Image render;
        private: float brightness;
        private: transport::NodePtr node;
        private: transport::PublisherPtr pub;
        private: msgs::Color output;
  };

  // Register this plugin with the simulator
  GZ_REGISTER_SENSOR_PLUGIN(CameraDump2)
}
```

```
<?xml version='1.0'?>
<sdf version='1.4'>
  <model name="line_tracker">
        <static>false</static>//The vehicle is dynamic
                <link name='chassis'>
                  <inertia>
                    <mass>0.65</mass><!--inertia-->
                    <inertial>
                      <ixx>0.001806</ixx>
                      <iyy>0.07551</iyy>
                      <izz>0.02500</izz>
                      <ixz>0.06233</ixz>
                      <ixy>0.6452</ixy>
                      <iyz>0.00594</iyz>
                        <pose>0 0 0.0485 0 0 0</pose>>><!--origin of the chassis,
situated in the middle-->
                        <collision name="chassis_collision">
                                <pose>0 0 0.0165 0 0 0</pose>
                                <geometry>
                                        <box>
                                                <size>0.16 0.12 0.036</size><!--geometry
used by the physiscs engine-->
                                        </box>
                                </geometry>
                        </collision>
                        <visual name='chassis_visual'>
                                <cast_shadows>false</cast_shadows>
                                <pose>0 0 -0.0485 0 0 0</pose>
                                <geometry>
                                        <mesh><uri>model://line_tracker/meshes/
gazebo5.dae</uri></mesh><!--mesh in the visual part-->
                                </geometry>
                        </visual>
                  <collision name='caster_collision'>
                    <pose>-0.0775 0 -0.025 0 0 0</pose>
                    <geometry>
                                        <sphere>
                                                <radius>.0235</radius><!--caster wheel-->
                      </sphere>
                    </geometry>
                  </collision>
                  <visual name='caster_visual'>
                    <pose>-0.0775 0 -0.025 0 0 0</pose>
                    <geometry>
                      <sphere>
                        <radius>.0235</radius><!--caster wheel visual-->
                      </sphere>
                    </geometry>
                  </visual>
                        <collision name='motor_left'>
                        <pose>0.0225 .05 -.0125 0 0 0</pose>
                                <geometry>
                                        <box>
                                        <size>0.065 0.02 0.022</size>
                                </box>
                                </geometry>
                        </collision>
                        <collision name='motor_right'>
                        <pose>0.0225 -.05 -.0125 0 0 0</pose>
                                <geometry>
                                        <box>
                                        <size>0.065 0.02 0.022</size>
                                </box>
                                </geometry>
                        </collision>
                        <sensor type='camera' name='left_camera'>
                                <pose> 0.095 0.005 -0.034 0 1.5707 0</pose>
                                <visualize>true</visualize>
                                <topic>left_camera</topic>
```

```
                                <always_on>1</always_on>
                                <update_rate>17</update_rate>
                                <camera>
                                        <horizontal_fov>1.2</horizontal_fov>
                                        <image>
                                                <width>480</width>
                                                <height>480</height><!--image resolution--
>
                                                <format>RGB_INT8</format>
                                        </image>
                                        <clip>
                                                <near>0.001</near>
                                                <far>0.03</far>
                                                <!--Hypotesis there is no object between
the camera an the line -->
                                        </clip>
                                </camera>
                                <plugin name='camera_dump_1'
filename='libcamera_dump_1.so'/><!--sensor plugin-->
                        </sensor>
                        <sensor type='camera' name='right_camera'>
                                <pose> 0.095 -0.005 -0.034 0 1.5707 0</pose>
                                <visualize>true</visualize>
                                <topic>right_camera</topic>
                                <always_on>1</always_on>
                                <update_rate>17</update_rate>
                                <camera>
                                        <horizontal_fov>1.2</horizontal_fov>
                                        <image>
                                                <width>480</width>
                                                <height>480</height><!--image reolution-->
                                                <format>RGB_INT8</format>
                                        </image>
                                        <clip>
                                                <near>0.001</near>
                                                <far>0.03</far>
                                                <!--Hypotesis there is no object between
the camera an the line -->
                                        </clip>
                                </camera>
                                <plugin name='camera_dump_2'
filename='libcamera_dump_2.so'/><!--sensor plugin-->
                        </sensor>
                        <sensor type='sonar' name='ultrasound'>
                                <pose> 0.116 0.00 0.0115 0 -1.5707 0</pose>
              <always_on>1</always_on>
              <update_rate>1000</update_rate>
              <visualize>false</visualize>
                    <sonar>
                                        <min>0.02</min><!--ultrasound characteristics of
the supplier table-->
                                        <max>4</max>
                                        <radius>1.07</radius>
                    </sonar>
      </sensor>
                </link>
                <link name='right_wheel'><!--link for the right wheel-->
                        <pose>.045 -.075 .035 0 1.5707 1.5707</pose>
                        <collision name='collision'>
                                <geometry>
                                        <cylinder>
                                                <radius>.035</radius>
                                                <length>.025</length>
                                        </cylinder>
                                </geometry>
                        </collision>
                        <visual name='visual'>
                                <geometry>
                                        <cylinder>
```

```
                                        <radius>.035</radius>
                                        <length>.025</length>
                                </cylinder>
                        </geometry>
                </visual>
        </link>
        <link name='left_wheel'><!--link for a left wheel-->
                <pose>.045 .075 .035 0 1.5707 1.5707</pose>
                <collision name='collision'>
                        <geometry>
                                <cylinder>
                                        <radius>.035</radius>
                                        <length>.025</length>
                                </cylinder>
                        </geometry>
                </collision>
                <visual name='visual'>
                        <geometry>
                                <cylinder>
                                        <radius>.035</radius>
                                        <length>.025</length>
                                </cylinder>
                        </geometry>
                </visual>
        </link>
        <joint type='revolute' name='left_wheel_hinge'><!--joints to articulate
the chassis and the wheels-->
                <pose>0 0 -.0125 0 0 0</pose>
                <child>left_wheel</child>
                <parent>chassis</parent>
                <axis>
                        <xyz>0 1 0</xyz>
                </axis>
        </joint>
        <joint type='revolute' name='right_wheel_hinge'>
                <pose>0 0 .0125 0 0 0</pose>
                <child>right_wheel</child>
                <parent>chassis</parent>
                <axis>
                        <xyz>0 1 0</xyz>
                </axis>
        </joint>
  </model>
</sdf>
```

```
<?xml version="1.0"?>
<sdf version="1.4">
  <model name="ground3">
    <static>true</static>
    <link name="link">
      <collision name="collision">
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>100 100</size>
          </plane>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>100</mu>
              <mu2>50</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
      <visual name="visual">
                                <pose>1.74365 -0.005 0 0 0 0</pose>
        <cast_shadows>false</cast_shadows>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>3.500 0.500</size>
          </plane>
        </geometry>
              <material>
                <script>
                  <uri>model://ground3/materials/scripts</uri>
                  <uri>model://ground3/materials/textures</uri>
                  <name>ground3/Image1</name>
                </script>
              </material>
      </visual>
                        <visual name='visual1'>
                                <pose>3.996825 4.1225 0 0 0 0</pose>
        <cast_shadows>false</cast_shadows>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>8.00635 7.755</size>
          </plane>
        </geometry>
              <material>
                <script>
                  <uri>model://ground3/materials/scripts</uri>
                  <uri>model://ground3/materials/textures</uri>
                  <name>ground3/Image2</name>
                </script>
              </material>
                        </visual>
                        <visual name='visual2'>
                                <pose>3.996825 -4.1275 0 0 0 0</pose>
        <cast_shadows>false</cast_shadows>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>8.00635 7.745</size>
          </plane>
        </geometry>
              <material>
                <script>
                  <uri>model://ground3/materials/scripts</uri>
                  <uri>model://ground3/materials/textures</uri>
                  <name>ground3/Image2</name>
```

```
              </script>
            </material>
                      </visual>
                      <visual name='visual3'>
                              <pose>-4.003175 0 0 0 0 0</pose>
        <cast_shadows>false</cast_shadows>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>7.99365 16</size>
          </plane>
        </geometry>
              <material>
                <script>
                  <uri>model://ground3/materials/scripts</uri>
                  <uri>model://ground3/materials/textures</uri>
                  <name>ground3/Image2</name>
                </script>
              </material>
                      </visual>
                      <visual name='visual4'>
                              <pose>5.740475 -0.005 0 0 0 0</pose>
        <cast_shadows>false</cast_shadows>
        <geometry>
          <plane>
            <normal>0 0 1</normal>
            <size>4.49365 0.5</size>
          </plane>
        </geometry>
              <material>
                <script>
                  <uri>model://ground3/materials/scripts</uri>
                  <uri>model://ground3/materials/textures</uri>
                  <name>ground3/Image2</name>
                </script>
              </material>
                      </visual>
    </link>
  </model>
</sdf>
```