POLITECNICO
MILANO 1863

Extended project work

A Gazebo car simulator, analysis and comparison with a single-track model

Colli Stefano, Pagani Mattia, Panelli Erica

"Control of Mobile Robots" Course

A.A. 2022/2023

Abstract

Contents

Chapter 1

Notes on Installation and Launch

1.1 Installation

1.1.1 Downloading material

The project is based on the material of original MIT racecar. That's it, we have generated a new ROS environment copying MIT repository packages. In particular the following packages have been downloaded:

- ackermann_msgs
- racecar
- racecar_gazebo

They can be found at the link: `https://github.com/mit-racecar`

1.1.2 Additional packages to be installed

To be able to compile the project it is necessary to download two internal ROS packages which will be used by the racecar ones. Launch the following commands:

```
sudo apt install ros-noetic-ros-control
sudo apt install ros-noetic-ros-controllers
```

Otherwise an error will be thrown when `catkin_make` command is called.

1.1.3 Additional modifications

In some cases, to avoid conflicts, it's required to change Python environment to version 3 in each file of the original packages. In particular, if Python environment is set to 3, modifications are needed for `joy_teleop.py` file:

- Row 277: replace ',' with 'as'
- Row 282: replace `iteritems` with `items`

1.2 Launch

1.2.1 Original Project

In order to launch original project, once it's compiled following ROS guide, following steps should be followed:

- (run) `roscore`
- (run) `keyboard_teleop.py`
- (run) `racecar_gazebo racecar.tunnel`

If there are no errors the user should be able to see the racecar in a Gazebo environment. WASD keys on keyboard produce car moves.

Chapter 2

General Project Structure

2.1 Catkin Workspace Directories

2.1.1 Original MIT Racecar Packages

ackermann_cmd_mux (racecar folder)	...
ackermann_msgs	Contains definitions of **AckermannDrive** and **AckermannDriveStamped** messages, used by the racecar to compute movements.
racecar (racecar folder)	Directory which contains
racecar_control (racecar_gazebo folder)	Contains launch files to load controllers used to manage the motors of the racecar. Also load nodes which dispatch messages to controllers.
racecar_description (racecar_gazebo folder)	Contains a description of the racecar, in terms of models, meshes ecc... It will be used by Gazebo to represent it.
racecar_gazebo (racecar_gazebo folder)	Mainly contains launch scripts used to load all necessary nodes, worlds and other components to open a Gazebo instance with a controllable car.

2.1.2 Added Packages

car_control	Contains node which performs the linearization of the nonlinear bycicle **dynamic** model. It' receives desired velocities from trajectory tracker and sends Ackermann commands to the racecar.
car_kinematic_control	Contains node which performs the exact linearization of the nonlinear bycicle **kinematic** model. It' receives desired velocities from trajectory tracker and sends Ackermann commands to the racecar.
trajectory_tracker	Generates (or receives in input) a desired trajectory and actual car positions, than compute desired velocities to be sent to controllers.

2.1.3 Added Plugin

gazebo_ros_pacejka	Plugin used to replace inner wheel friction phisical model of ROS with a custom one.

Chapter 3

MIT Racecar Model

Before entering in the detail of the RACECAR model description we want to give a small introduction about the Gazebo simulator environment and the URDF/SDF description standard.

3.1 Gazebo Simulator

Gazebo is an open-source 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments.
It integrates the ODE physics engine to provide a high fidelity physics simulation, OpenGL rendering to have visually appealing scenarios, and it supports code for sensor simulation and actuator control

3.1.1 ODE

The ODE (*Open Dynamics Engine*) is a physics engine for simulating articulated rigid body dynamics. It is fast, flexible, robust and has built-in collision detection.
An articulated structure is created when rigid bodies of various shapes are connected together with joints of various kinds.
ODE is designed to be used in interactive and real-time simulations. It has hard contacts, this means that a special non-penetration constraint is used whenever two bodies collide.

ODE uses the notion of *Coulomb Friction*, which is the most common friction model.
The simplified friction law is:

$$F_c = \begin{cases} \mu N \cdot sign(v) & \text{if } |v| > 0 \\ \min(|F_{app}|,\ \mu N) \cdot sign(F_{app}) & \text{if } v = 0 \end{cases}$$

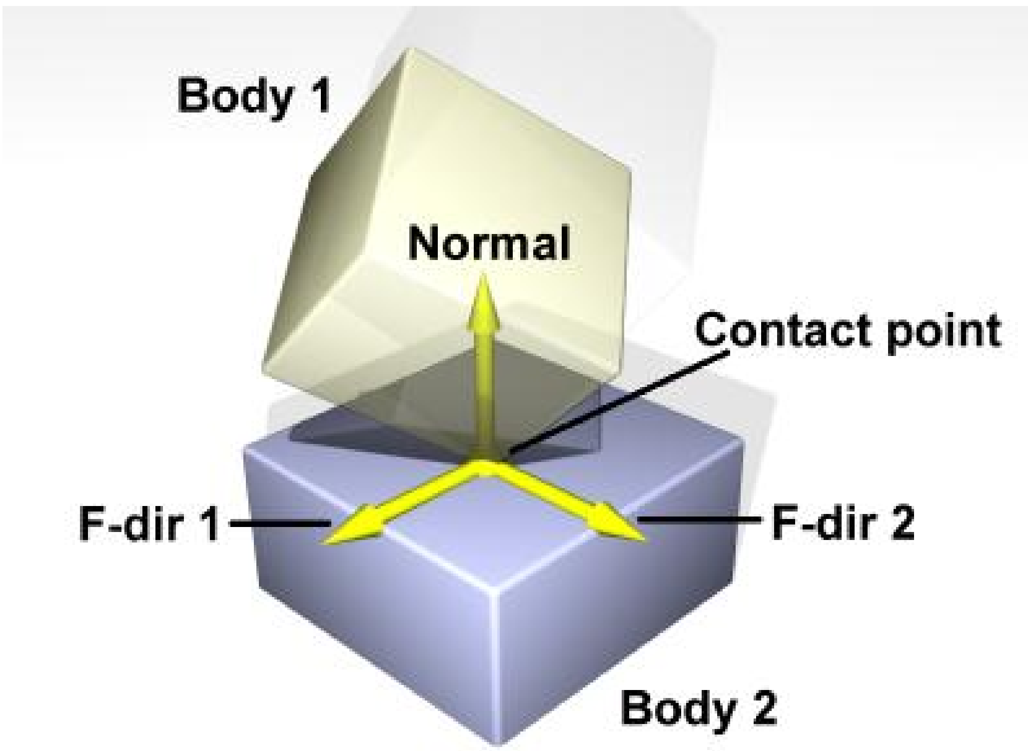


Figure 3.1: Body interaction causing friction

Where F_c is the Coulomb friction force, v is the sliding speed, μ is the coefficient of friction, N is the normal contact force and F_{app} is the applied force on the body.

3.2 URDF and SDF

URDF (*Unified Robot Description Format*) is an XML format for representing a robot model used by ROS.
A number of different packages and components compose URDF:

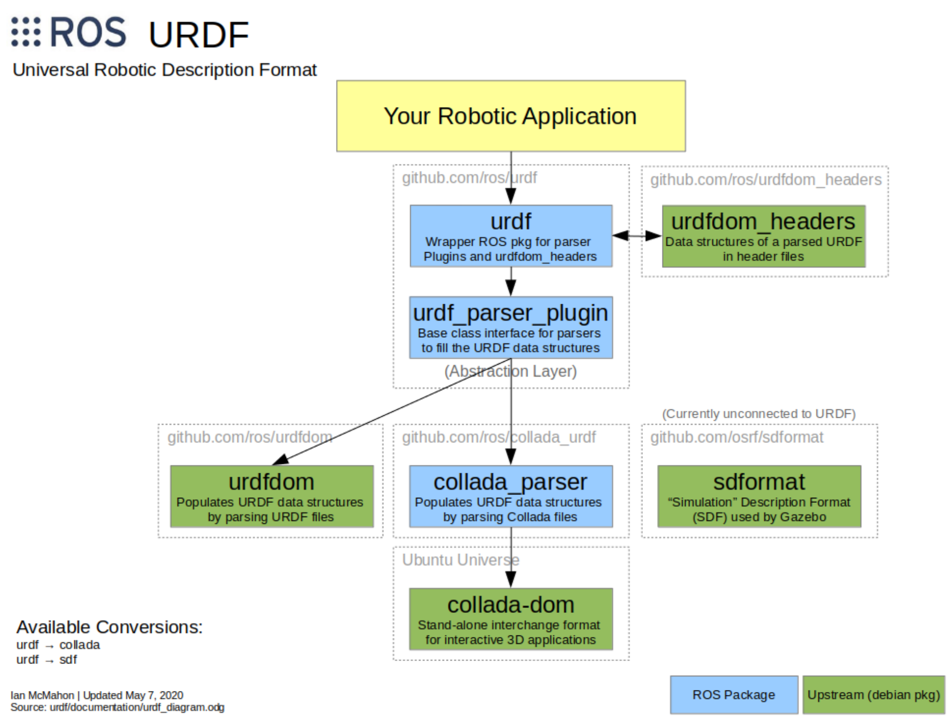


Figure 3.2: URDF description

URDF can only specify the kinematic and dynamic properties of a single robot in isolation, it cannot specify the pose of the robot itself with respect to the world it is placed in.
URDF is not a universal description format since it cannot specify joint loops (parallel linkages), and it lacks the possibility of specifying friction and other dynamic properties.

For the reasons above, Gazebo uses another XML format called SDF (*Simulation Description Format*) which is a complete description for everything from the world level down to the robot level.
It has been developed also a macro language called XACRO (*XML Macro*) to make it easier to maintain the robot description files, increase their readability, and to avoid duplication of code in the robot description files. In our case, the model is described using XACRO files, and then it is converted to SDF by the *gazebo_ros* package. Finally, it is spawned it in the Gazebo simulated world.
The basic building blocks for describing a robot using URDF/SDF are *links* and *joints*.

3.2.1 Links

In URDF/SDF, a *Link*[1] element describes a rigid body with an inertia, visual features, and collision properties.

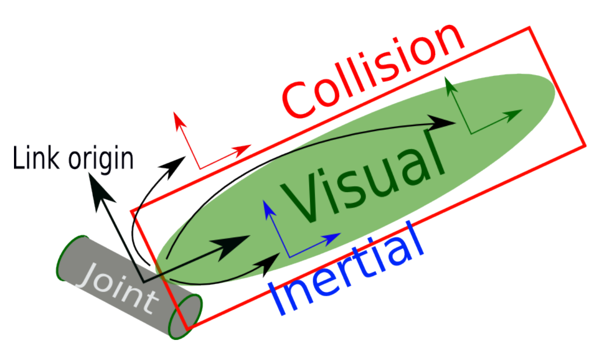


Figure 3.3: Link schematic

Each *link* element is composed by three main parts (encapsulated in their respective XML tags):

- *Inertial* in which we specify the link mass, position of its center of mass, and its central inertia properties;
- *Visual* in which we describe the shape of the object (e.g. box, cylinder) for visualization purposes. We can also use multiple instances of

[1] Visit `https://wiki.ros.org/urdf/XML/link` for a detailed description

<visual> tags for the same link to define the shape in a constructive way.

- *Collision* in which we list the collision properties of a link, which are usually different from the visual ones. In fact, simpler collision models are often used to reduce computation time.
  Also in this case, we can also use multiple instances of <collision> tags for the same link to define the desired shape in a constructive way.

3.2.2 Joints

The *joint*[2] element describes the kinematics, dynamics and safety limits of a joint connecting together two link objects.

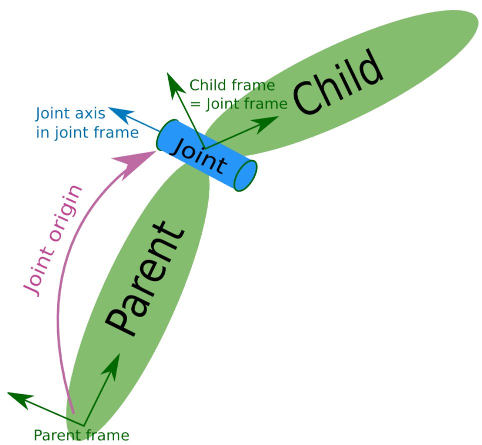


Figure 3.4: Joint schematic

Using the attribute *type* we can specify the type of the joint as one of the following:

- **revolute** — a hinge joint that rotates along the provided axis and has a limited range specified by the upper and lower limits;
- **continuous** — a continuous hinge joint that rotates around the provided axis and has no upper and lower limits;

[2]Visit `https://wiki.ros.org/urdf/XML/joint` for a detailed description

- **prismatic** — a sliding joint that slides along the axis, and has a limited range specified by the upper and lower limits;
- **fixed** — this is not really a joint because it cannot move since all degrees of freedom are locked. This type of joint is used to rigidly attach links together;
- **floating** — this joint allows motion for all 6 degrees of freedom;
- **planar** — this joint allows motion in a plane perpendicular to provided the axis.

Using <parent> and <child> XML elements, we can define which links (rigid bodies) are attached together through the joint.

3.2.3 URDF in Gazebo

In order to use a robot described using URDF within Gazebo, it is mandatory that all links have an *<inertial>* element.
Optionally, we can introduce additional information appending *<gazebo>* elements to the links (e.g., to add sensor plugins to convert colors to Gazebo format, ...), or to the joints (e.g., to set proper damping dynamics, add actuator control plugins, ...).[3]

3.3 RACECAR model description

In this section we are going to explain the structure of the *RACECAR project* developed by MIT, by focusing on the model the robot and on its simulation made by using Gazebo.
The robot is composed by nine links: a chassis, four wheels, two steering hinges, one Hokuyo LIDAR sensor and one camera; and by eight joints: four for the wheels, two for the steering mechanism, and two for connecting the camera and the LIDAR to the car.

[3] Refer to `http://classic.gazebosim.org/tutorials?tut=ros_urdf` for a detailed guide on the conversion

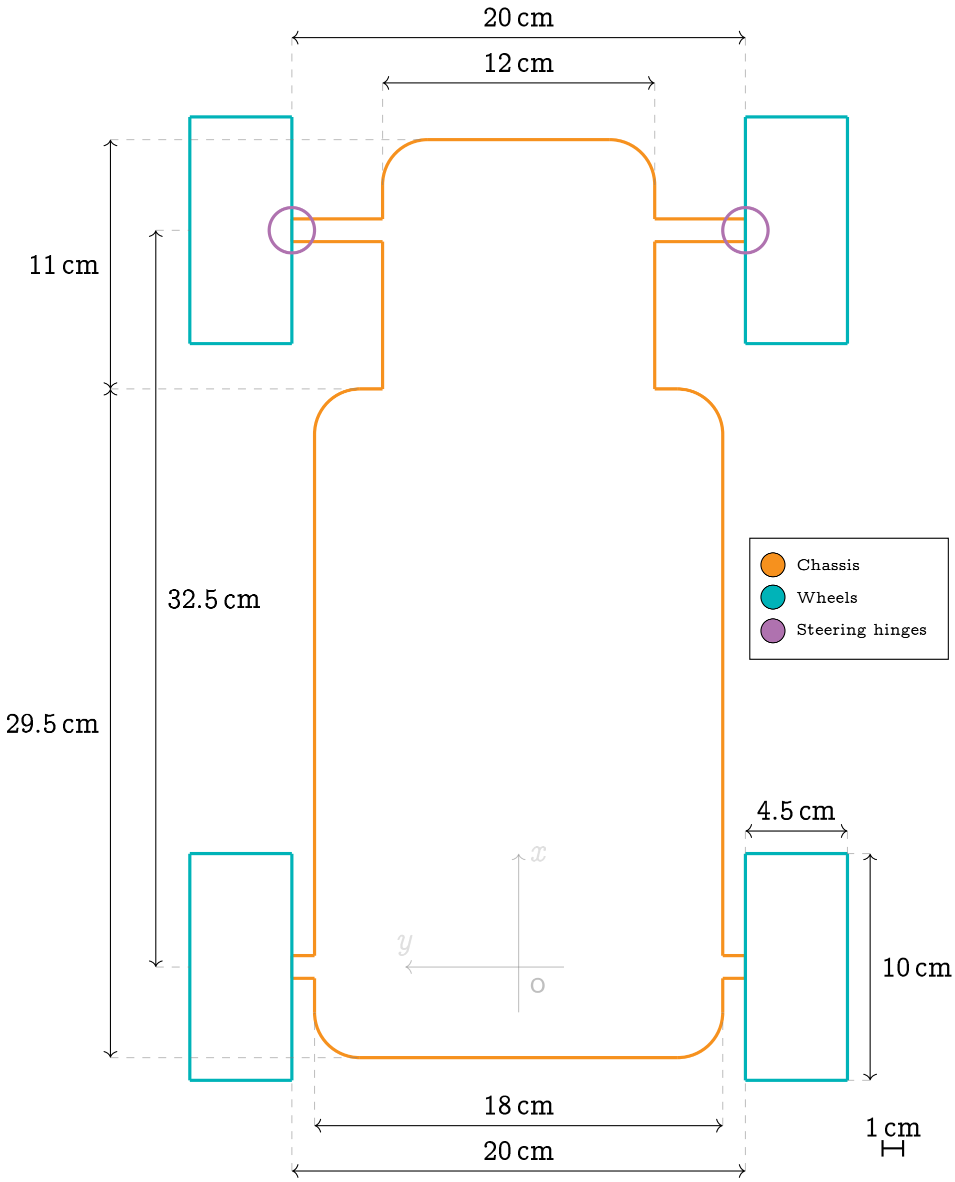


Figure 3.5: RACECAR links schematic

3.3.1 RACECAR links

Chassis

The robot chassis weights 4 kg and has an irregular shape (see schematic in 3.5); it is placed 5 cm above the ground (0.05 m offset on the z axis). Since there is no <collision> element, all collisions with the chassis are ignored.
The inertial matrix is:

$$\begin{bmatrix} 0.010609 & 0 & 0 \\ 0 & 0.050409 & 0 \\ 0 & 0 & 0.05865 \end{bmatrix}$$

It is a frictionless surface ($\mu_1 = \mu_2 = 0$), but has spring constant equals to 1×10^7 N/m and a damping constant of 1.0 kg/s.

Wheels

Each one of the four wheels, weighting 0.34 kg, has a cylindrical shape with rounded borders defined by a custom STL (*STereo Lithography interface format*) file.
The inertial matrix is reported here:

$$\begin{bmatrix} 0.00026046 & 0 & 0 \\ 0 & 0.00026046 & 0 \\ 0 & 0 & 0.00041226 \end{bmatrix}$$

Each wheel has also two friction coefficients, μ_1 and μ_2, both set to 1; these coefficients are referring to the principal contact directions along the contact surface. Furthermore, we have a spring constant of 1×10^7 N/m and a damping constant of 1.0 kg/s.
Since we are interested in the ground tire interaction, it is necessary to specify the first friction direction in the local reference frame along which frictional force is applied. Such vector must be of unit length and perpendicular to the contact normal, it is typically tangential to the contact surface: rear wheels have a friction direction vector equal to $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$, instead front wheels have a friction direction vector equal to $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$.
Each wheel has its own collision space, it is delimited by a cylinder 4.5 cm long and having a radius of 5 cm. The origin of the collision spaces for right wheels are shifted by $\begin{bmatrix} 0 & 0 & 0.0225 \end{bmatrix}$, instead the origin of the collision

spaces for the left wheels are shifted by $\begin{bmatrix} 0 & 0 & -0.0225 \end{bmatrix}$. These translations, performed with respect to the joints frames (connecting the wheels to the chassis), are necessary to align the generated collision cylinders with the visual geometries. [4]

Steering Hinges

The RACECAR model has two steering hinges for the front wheels, both have identical geometry: they are modeled as spheres.
The steering hinges only have visual properties specified, while no collision spaces are provided.
Each one weights 0.1 kg, and has the following inertial matrix:

$$\begin{bmatrix} 4E-06 & 0 & 0 \\ 0 & 4E-06 & 0 \\ 0 & 0 & 4E-06 \end{bmatrix}$$

Hokuyo Laser

The RACECAR model is also equipped with a Hokuyo LIDAR sensor. To include it in the simulation, a laser Gazebo plugin is used[5].
The laser is set to have an update rate of 40 Hz with 1081 simulated rays scanning the horizontal plane from angle -2.356 rad to 2.356 rad; the distance at which it can detect obstacles ranges from a minimum of 10 cm to a maximum of 10 m, with a resolution of 1 cm.
To have a simulation as close to the real world as possible, a Gaussian noise is introduced in the readings of the LIDAR sensor: with a mean of 0.0 m and a standard deviation of 0.01 m, 99.7% of readings are within 0.03 m of the true distance.
The sensor weights 0.13 kg and its inertial matrix is :

$$\begin{bmatrix} 4E-06 & 0 & 0 \\ 0 & 4E-06 & 0 \\ 0 & 0 & 4E-06 \end{bmatrix}$$

The collision space delimited by a cube having side long 0.1 m.

[4] At a first glance, different translations of the wheels along the z axis may appear odd. However, it is motivated by the fact that the wheels joints have previously been rotated around the x axis.

[5] Visit `https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#Laser` for more information

ZED Camera

The last sensor attached to the RACECAR model is a ZED RGB camera, simulated with a camera Gazebo plugin[6].
This sensor has an update rate of 30.0 Hz, the resolution of the acquired image is 640x480 in $B8G8R8$ color space. Objects closer than the camera near clipping plane (placed at 0.02 m) or further than the camera far clipping plane (positioned at 300 m) won't be visible in the picture.
As in the case of the LIDAR, we have an artificial Gaussian noise with a mean of 0 and a standard deviation of 0.007 affecting the measurements. Such noise is sampled independently per pixel on each frame, and it is added to each of its color channels (having values lying in the range [0,1]).
The sensor weights 1×10^{-5} kg, and its inertial matrix is:

$$\begin{bmatrix} 1E-6 & 0 & 0 \\ 0 & 1E-06 & 0 \\ 0 & 0 & 1E-06 \end{bmatrix}$$

The collision and visual spaces coincide, they are described by a box of dimensions 0.033 m x 0.175 m x 0.030 m.

3.3.2 RACECAR joints

Wheel joints

All four wheel joints are continuous joints which allow the tires to rotate freely.
The effort limit on each joint is $10\,\mathrm{N} \cdot \mathrm{m}$, while the velocity limit is 100 rad/s.
The joints are rotated by $\frac{\pi}{2}$ with respect to the x axis so that the attached wheels end up in the correct alignment. In fact, the tires are described (in the STL file) as cylinders with their bases parallel to the ground, so we need the 90° rotation to make them perpendicular to the terrain.
Apart from the positioning in space, the only difference between front wheels joints and rear wheel joints is their parent link: the former are attached to the steering hinges, while the latter are directly connected to the chassis.

[6]Visit `https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#Camera` for more information

Steering hinge joints

Right and left steering hinges are connected to the chassis through two revolute joints.
Because of the joints nature, the range of motion is limited between a minimum and a maximum angle: in this case, $-1.0\,\mathrm{rad}$ and $1.0\,\mathrm{rad}$. Originally, the effort limit was set to $10\,\mathrm{N}\cdot\mathrm{m}$ in order to simulate a realistic steering dynamics. Since the feedback linearization law we used required to have an instantaneous steering mechanism, we increased such parameter to $1000\,\mathrm{N}\cdot\mathrm{m}$, drastically reducing the steering time.
Finally, the velocity limit is $100\,\mathrm{rad/s}$.

Hokuyo laser Joint

The laser sensor is attached to the chassis by a fixed joint, thus all its degrees of freedom are locked. This type of joint does not require to specify any additional information apart from the position and the connected elements.

ZED camera joint

In a similar way as the laser sensor, the camera is connected to the chassis through a fixed joint. Because of that, only the position, the parent and child connections are specified.

3.4 Plugins

RACECAR model is enriched with functionalities that can control the several components specified before in the model description section.
These functionalities are implemented as plugins: chunks of code compiled as shared libraries and then inserted into the simulation. A plugin has direct access to Gazebo functionalities through C++ classes.
Plugins are used because of several advantages: they let developers control almost any aspect of Gazebo, they are self-contained routines that are easily shared, and they can be inserted and removed quickly from a running system.
There are six different types of plugins and each of them is managed by different Gazebo components, these types are:

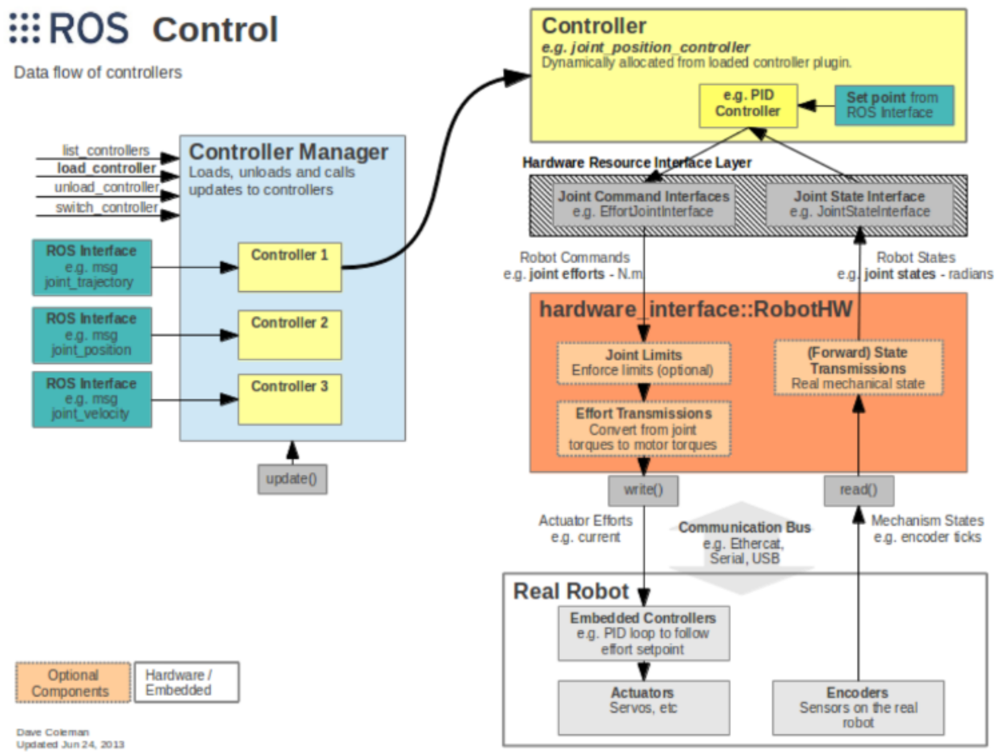
- World, to control the simulated world;
- Model, to control a specific model;
- Sensor, to simulate a sensor behaviour;
- System, to introduce functionalities during Gazebo startup;
- Visual, to modify the visual representation of the simulation;
- GUI, to develop additional elements for the Gazebo interface.

A plugin type should be chosen based on the desired functionality to implement.[7]
In this section we will analyze every plugin present in the /textitRACECAR project by specifying characteristics, functionalities and connections between components.

3.4.1 gazebo_ros_control plugin

The *gazebo_ros_control*[8] plugin is an adapter of the ros_control library for Gazebo.
The *ros_control*[9] package takes as input the joint set data from the robot actuators and an input set point, then it uses a PID control loop feedback mechanism to control the output. Such value, which is typically an effort, is then sent to the robot actuators.
When there is not a one-to-one mapping of joint positions, efforts, etc... it is needed to use *transmissions*: interfaces that map effort/flow variables to output effort/flow variables while preserving power.

[7]Visit `https://classic.gazebosim.org/tutorials?tut=ros_gzplugins` for additional information
[8]See `https://classic.gazebosim.org/tutorials?tut=ros_control` for more information
[9]See `https://wiki.ros.org/ros_control` for more information

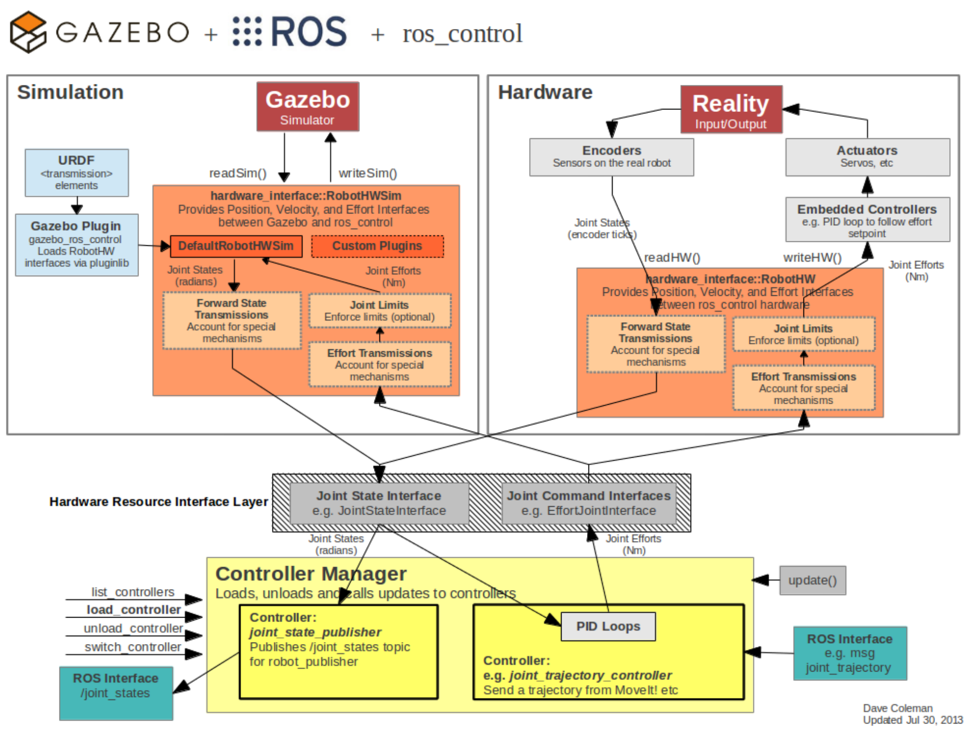


In the *RACECAR project* we have six *effort_controllers*[10] form the ros_control library, they are PID controllers acting on the joint variables:

- four, one for each wheel, of type *joint_velocity_controller*, which receive a velocity setpoint and sends an effort output. They are pure proportional controllers with a gain of 1.0 for the rear wheels, and a gain of 0.5 for the front tires;
- two, one for each steering hinge, of type *joint_position_controller*, which receive a position input and sends an effort output. They are PD controllers with parameters $K_p = 1.0$ and $K_d = 0.5$;

The gazebo_ros_control plugin provides default ros_control interfaces, and it also provides a pluginlib-based interface to implement custom interactions between Gazebo and ros_control for simulating more complex mechanisms.

[10]Relative configuration can be found at `/catkin_ws/src/mit_racecar/racecar_gazebo/racecar_control/config/racecar_control.yaml`



In order to use ros_control within Gazebo, we need to enrich the URDF description adding <transmission> elements in order to link actuators with joints. In our model, we have six transmissions: four to connect each wheel to a motor (with a mechanical reduction of 1), and two connecting each steering hinge to a dedicated motor (also in this case the mechanical reduction is 1).
Finally, we add to the URDF file the gazebo_ros_control plugin by specifying the library filename ("libgazebo_ros_control.so") and its parameters like a desired "robotNamespace".

3.4.2 Laser plugin

The Hokuyo laser plugin is specified in a *Sensor* tag. It refers to the library *libgazebo_ros_laser.so* which takes the role of controller of the laser (named *gazebo_ros_hokuyo_controller*).
This controller gathers range data from a simulated ray sensor, publishes range data using *sensor_msgs::LaserScan* messages on the ROS topic

named */scan*.

3.4.3 Camera plugin

Similarly as the laser scan plugin the camera plugin is specified in a *Sensor* tag. It refers to the library *libgazebo_ros_camera.so* which allows to control this sensor (the controller is named *camera_controller*).
The camera publishes image messages on *camera/zed/rgb/image_rect_color* ROS topic, while the info messages are published on *camera/zed/rgb/camera_info*.
The coordinate frame where the image is published (tf tree) is *camera_link*.

3.5 racecar_gazebo code description

In the previous sections we have illustrated how the model has been designed and its characteristics. In this section we want to explain how the code is organized and the technical details of the implementation.
We will focus only on the *racecar_gazebo* package inside *mit_racecar*.

3.5.1 racecar_description

The *racecar_description* package contains all specifications and files useful to describe the car and the environments of the simulation.
In the *urdf* folder there are four different XML files:

- **macros.xacro** containing macros definitions for inertial parameters, geometries and transmissions for all the elements composing the car model. The usage of a xacro file allows to have a more readable and organized code in the main URDF files.
- **materials.xacro** with the specification of the materials used in the car construction within Gazebo (in this case each material is simply determined by a RGB color that will be used for the visualization).
- **racecar.xacro** which is the main description file. It contains all links, joints and transmissions composing the car model.
  It also includes all the other macro files and the Gazebo one.

- **racecar.gazebo** with friction parameters, plugin settings and all the additional information needed to use URDF within a Gazebo simulation.

In the *models* directory there are *.config* and *.sdf* files used to describe the simulation environments.
The last folder is named *meshes*, it contains *.STL* and *.dae* files which are the 3D models of the car components used by Gazebo for the visual rendering.

3.5.2 racecar_gazebo

In this package there are the files used to launch the Gazebo simulation.
In the *worlds* folder we have several *.worlds* files: they contain the description of the available environments (worlds) in which the model can be simulated.
The second folder, named *script*, contains the python implementation of *gazebo_odometry_node*. This node is in charge of translating Gazebo status messages (coming from "/gazebo/link_states" topic) to odometry data, and then publishing it on "/vesc/odom" ROS topic at a 20.0 Hz rate.
The last folder is named *launch* and contains one launch file for each world already implemented.
A launch file contains the list of ROS nodes to be spawned and their configuration; moreover, it can include other *.launch* files. Hence, we have a quick and tidy way to start the entire simulation.
In the package there are five *.launch* files; the main one is *racecar.launch*, while the others only differ for the world in which the simulation will take place.
File *racecar.launch* contains the following parts:

- world selection: it is specified the *.world* file which describes the simulation environment (in this case an empty flat plane);
- robot description generation: the *.xacro* file containing the description of the car model is converted to URDF format and loaded into the parameter server;
- racecar spawning: *racecar_spawn* node, from *gazebo_ros* package, reads the model URDF from the parameter server and implements it in the simulation;

- call to *racecar_control.launch*: which loads all the joints effort controllers and starts the *servo_commands* node;
- call to *mux.launch*: which creates a multiplexer to the handle different commands, having different priorities, received by the car.
- topic remapping: *better_odom* node, from *topic_tools* package, uses the "relay" functionality to subscribe to */vesc/odom* and republish to */pf/pose/odom*.

3.5.3 racecar_control

This package contains the logic of the car controllers.
In the *scripts* we have the implementation of two ROS nodes:

- *keyboard_teleop*
  Which allows the user to manually control the car by sending commands using the keyboard: *W* to go forward, *S* to move in reverse, *A* and *D* for turning left and right respectively. Pressing a different key will stop the car, and by using the combination *CTRL-C* the node is terminated and the controller stopped.
  A message of type *AckermannDriveStamped* containing speed, acceleration, jerk, steering_angle and steering_angle_velocity is then published on the */vesc/ackermann_cmd_mux/input/teleop* topic to be sent to the car command mutiplexer.
- *servo_commands*
  The main purpose of this node is to read the information published on the */racecar/ackermann_cmd_mux/output* topic by the multiplexer, and forward it to the various link controllers.
  It publishes the desired speed, pre-multiplied by 10, to the four wheel velocity controller topics; while, to the two hinge position controllers, it is forwarded the steering angle setpoints.

The second folder is named *config*, it contains only one file which is *racecar_control.yaml*. A YAML file loads node configuration parameters in the ROS parameter server, it contains the information about type, joint name and PID (proportional-integral-derivative) gain of each controller to be implemented using the ros_control library.

The *launch* directory contains three launch files regarding controller functionalities:

- *gazebo_ sim_ joy.launch*
  starts the simulation in the tunnel environment and calls *teleop.launch* (see below);

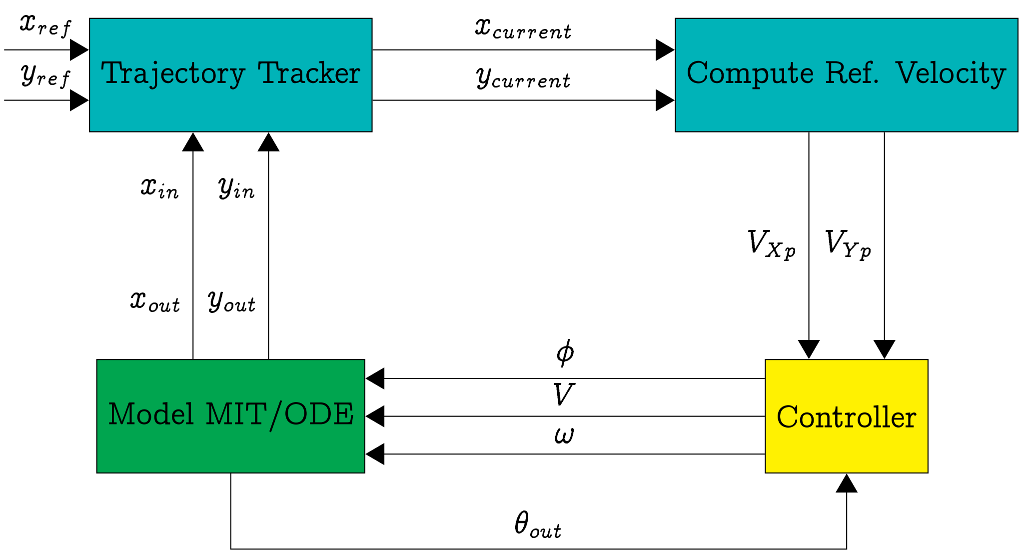
- *racecar_ control.launch*
  loads the joint controller configurations from *racecar_ control.yaml* to parameter server: Then it instantiates a *controller_ manager*: a node (implemented in ros_control) capable of spawning, loading, unloading and updating all the controllers specified in the configuration file (.yaml file). Finally, it starts the aforementioned servo_commands node.
  In addition, this file includes also some remapping of the topics used by the robot state publisher (robot_state_publisher node), the robot movements controller (servo_commands node) and the odometry publisher;

- *teleop.launch*
  starts the *keyboard_ teleop* node to allow the user to manually control the model.

Chapter 4

(Our) System Description

4.1 Scheme of the whole system



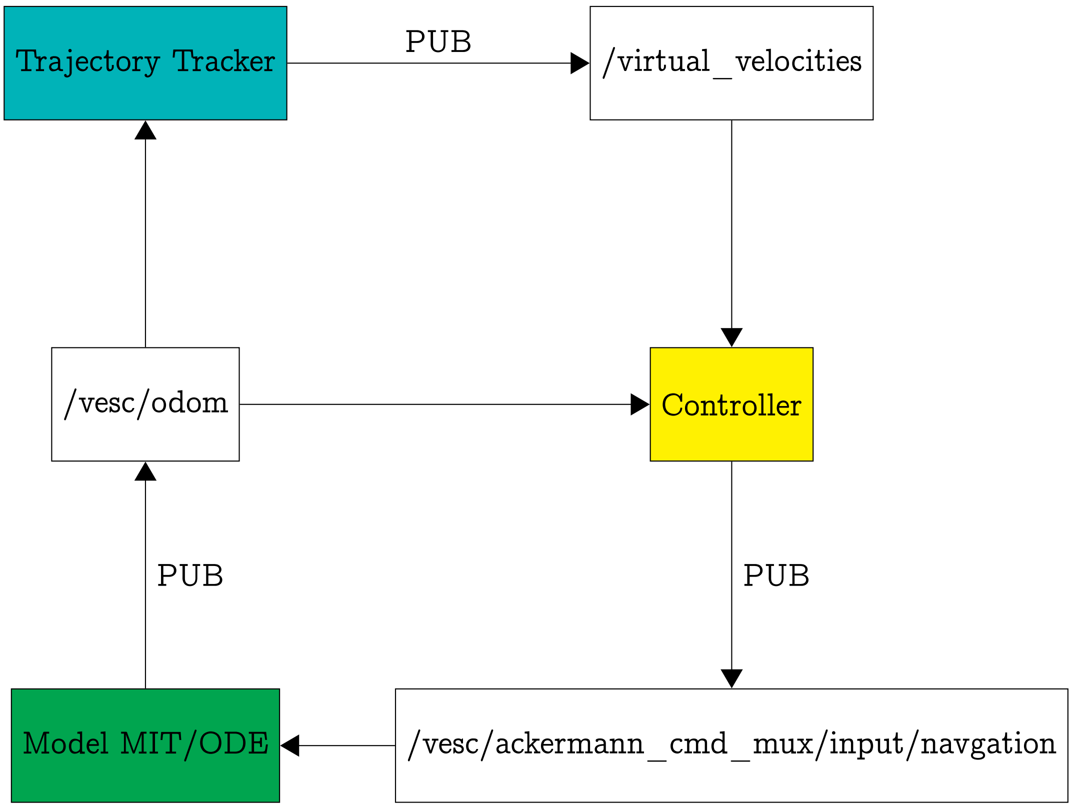
Symbol	**Meaning**
x_{ref}, y_{ref}	Ref. position of the trajectory
V_{Xp}, V_{Yp}	Required velocities of the point. They will be imposed by controller
ϕ	Steer degree of rotation
V	Vector velocity
ω	Steer speed of rotation
θ_{out}	Car pose: rotation around center axis
x_{out}, y_{out}	Car pose: x, y

Note that "trajectory tracker" generated trajectories are hard-coded, even if x_{ref} and y_{ref} are shown as input parameters. The user can select the trajectory using YAML configuration file (which will be explained in the relative section).

4.2 Topics

4.2.1 Scheme of topic publications/subscriptions

ROS Friction Model



4.2.2 Topics meaning

Common Topics

/virtual_velocities	Used by "trajectory tracker" to publish desired velocity components. These are read by controller in order to perform linearization and compute instructions for the model.
/vesc/ackermann_cmd_mux /input/navgation	Contains AckermannDriveStamped messages sent by controller. These messages contains information for the racecar, about velocity and steering.
/vesc/odom	The model uses this topic to publish odometry information of the racecar (position and orientation). These data are used both by tracker and controller. The first one compute differences between actual car position and desired position imposed by trajectory. The last one reads z-axis orientation useful to perform linearization.

There is another topic in which "trajectory tracker" publish, the */reference_trajectory*. This is used to read trajectory information to perform debug and register data for analysis.

Specific Topics

/vesc/low_level/ackermann_cmd_mux/output	xyz

Chapter 5

Detailed Package Description

5.1 Package (original) ackermann_msgs

5.2 Package (original) racecar

5.3 Package (original) racecar_gazebo

5.4 Package car_control

5.4.1 Intro

Even if this package is not used, it's correct to do a description of the objective it should have reached.

The aim was to implement a dynamic controller, which perform exact linearization of the nonlinear bicycle dynamic model. To do this it needs more parameter respect to the kinematic one.

In addition, we put a scheme (5.1) of the principal parameters and variables used for linearization.

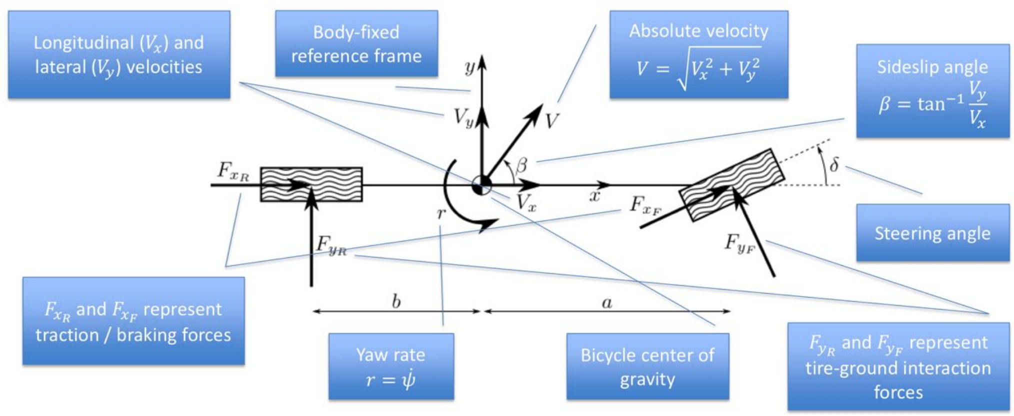


Figure 5.1: dynamic model with main parameters and variables

5.4.2 Configuration

Input values	
Vp_x	Point velocity x
Vp_y	Point velocity y
ψ	Yaw[1]
$\dot{\psi}$	Yaw rate[2]

Model parameters	
C_f, C_r	Viscous friction coefficients
a, b	Distance between wheels center and Center of Gravity
M	Vehicle mass
ϵ	Distance between Center of Gravity and a point P, along the velocity vector. Linearization is done around point P. This parameter should be chosen empirically

[1]In the System Scheme, this is represented by θ_{out}

[2]In the System Scheme, this is **not** represented (as we have used, for tests, only the kinematic model)

Intermediate computed values	
β	Sideslip angle: $\tan^{-1}\left(\frac{Vp_y}{Vp_x}\right)$

Output values	
V	Point absolute velocity
δ	Steering angle
ω	Steering speed

5.4.3 Launch

There is a lunch file which **should be used to execute the node**. This contains also information about debugging level and loads configuration file.

5.4.4 Node car_control

$$\beta = \tan^{-1}\left(\frac{Vp_y}{Vp_x}\right)$$

$$\delta = \frac{MV}{C_f}\omega + \frac{C_f + C_r}{C_f}\beta - \frac{bC_r - aC_f}{C_f}\frac{\dot{\psi}}{V}$$

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} \cos(\beta + \psi) & \sin(\beta + \omega) \\ -\frac{\sin(\beta+\psi)}{\epsilon} & \frac{\cos(\beta+\psi)}{\epsilon} \end{bmatrix} \begin{bmatrix} Vp_x \\ Vp_y \end{bmatrix}$$

5.5 Package car_kinematic_control

5.5.1 Intro

Before starting the explanation, we add a brief high level description of Quaternions, which are used in messages to represent orientations. Even a distinction between pose and position is done.

Quaternion: a different way to describe the orientation of a frame only. It's an alternative to Yaw, Pitch and Roll. A quaternion has four parameters: x, y, z, w. Pay attention, they are NOT a position vector.

Position: position of the robot in a 3D space.

Pose: position (3 DOF) + orientation (3 DOF).

In conclusion the pose has 6 D.O.F. which are: x, y, z, roll, pitch, yaw. Euler angles can be converted to quaternions, which are better. Transformation functions of ROS can do this conversion and the reverse one.

In addition, we put a scheme (5.2) of the principal parameters and variables used for linearization.

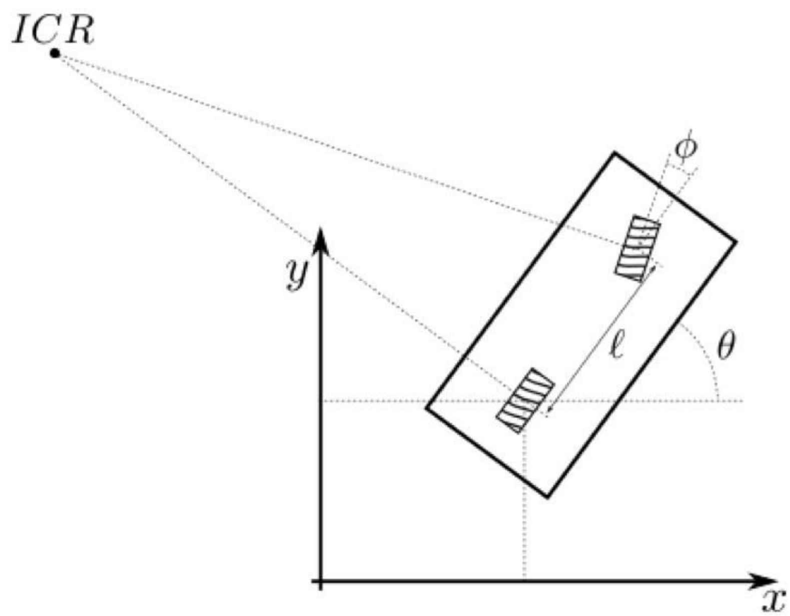


Figure 5.2: bicycle vehicle with main parameters and variables

5.5.2 Configuration

In the package there is a configuration file, containing: the parameter L, which represents distance between rear and front wheels; the parameter ϵ, the distance between Center of Gravity and a point P, along the velocity vector. Linearization is done around point P. This parameter should be chosen empirically.

Both are used in the linearization.

5.5.3 Launch

There is a lunch file which **should be used to execute the node**. This contains also information about debugging level and loads configuration file.

5.5.4 Node car_kin_controller

Node requirements: distance between rear and front wheels as parameter.

The node has two callbacks:

- One used to retrieve desired velocities of the point. These velocities are computed by trajectory tracker and published in `/virtual_velocities` topic, subscribed by the controller node.
- One used to retrieve the orientation of the car around z axis. This is done reading from `/vesc/odom`. The information retrieved are in the form of a quaternion and are converted into roll, pitch and yaw. Yaw is taken. In addition, even the speed around z axis is read (`twist.angular.z`).

The node perform an exact linearization of the nonlinear bicycle cinematic model. The change of coordinates is applied as follows:

$$V = V_{Xp} cos(\theta) + V_{Yp} sin(\theta)$$

$$\phi = \arctan\left(\frac{l}{\epsilon}\frac{V_{Yp} cos(\theta) - V_{Xp} sin(\theta)}{V_{Xp} cos(\theta) + V_{Yp} sin(\theta)}\right)$$

Where

- l is the distance between rear and front wheels
- ϵ is the distance between Center of Gravity and a point P
- V_{Xp} and V_{Yp} are the desired point velocities
- θ is the car orientation around z-axis
- ϕ is the steering angle

- V is the driving velocity of the front wheel

In addition, the program compute the steering speed as $\omega = \frac{V}{l}\tan(\phi)$. This value is not used in the construction of the message because it's ignored by the model.

Once the linearization is performed an `AckermannDriveStamped` message is built, containing V and ϕ. This message is published on `/vesc/ackermann_cmd_mux/input/navigation` topic, which is read by the model to make the car move. Linearization and command sending operations are repeated in a loop, which is the core of the node.

5.6 Package trajectoy_tracker

5.6.1 Package description

The *trajectory_tracker* package is responsible for two main tasks:

- generating the setpoints of the desired trajectory;
- applying the control law to make the robot track such trajectory.

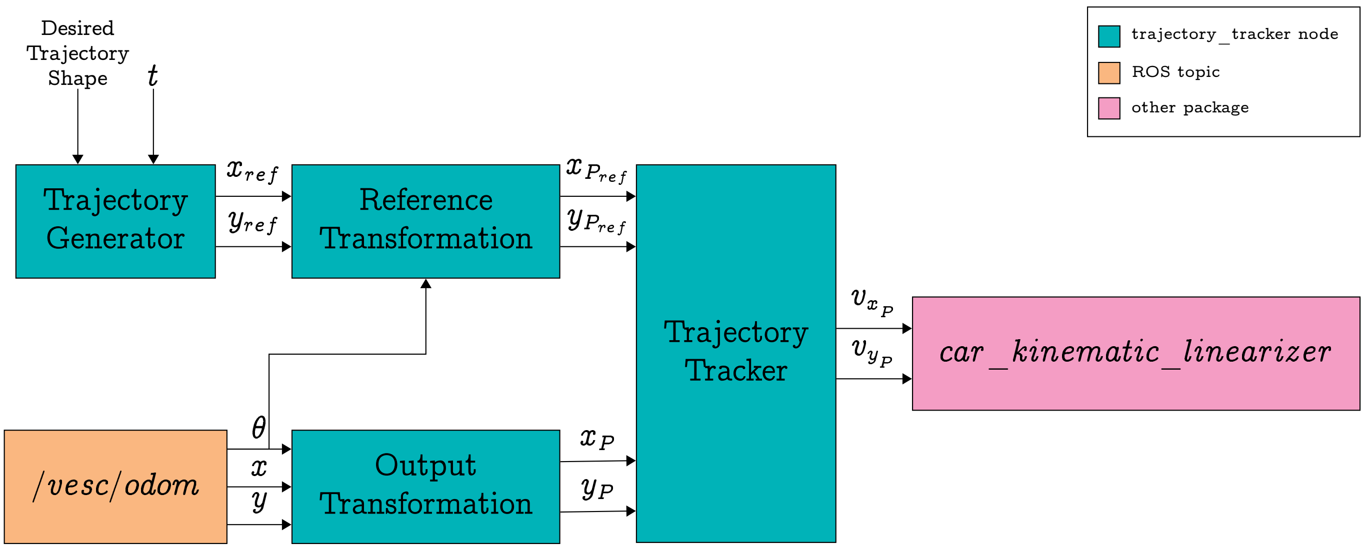


Figure 5.3: trajectory_tracker package schematic

The main components of the package are:

- **Trajectory Generator** which is in charge of computing the appropriate setpoint given the desired trajectory shape and the time t elapsed since the starting of the motion.
  The implemented trajectory shapes are:
  - Line, a simple linear path described by equations:

$$x_{ref} = at \qquad (\dot{x_{ref}} = a)$$

$$y_{ref} = bt \qquad (\dot{y_{ref}} = b)$$

Where a and b determine the direction (the line is parallel to the direction vector $\vec{d} = (a, b)$) and the speed of the trajectory;

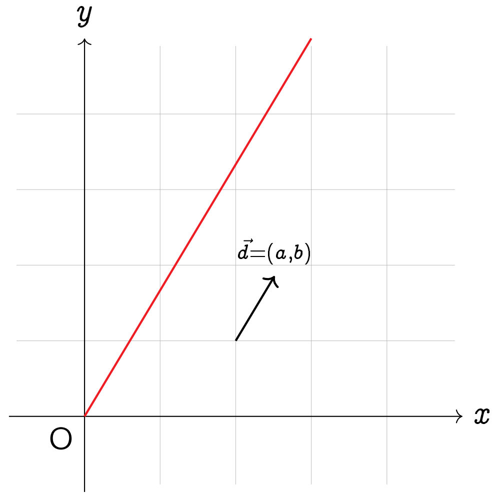


Figure 5.4: Linear trajectory (where $\vec{d} = (3, 5)$)

- Parabola, a parabolic path described by equations:

$$x_{ref} = 2at \qquad (\dot{x_{ref}} = 2a)$$

$$y_{ref} = at^2 \qquad (\dot{y_{ref}} = 2at)$$

Where a is the focal length of the parabola;

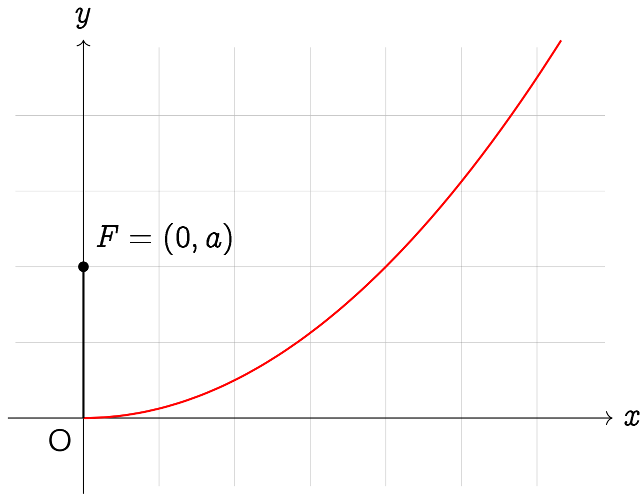


Figure 5.5: Parabolic trajectory (where $a = 2$)

- Circle, a circular path described by equations:

$$x_{ref} = r\cos(\omega t - \frac{\pi}{2}) \qquad (\dot{x_{ref}} = -\omega r \sin(\omega t - \frac{\pi}{2}))$$

$$y_{ref} = r\sin(\omega t - \frac{\pi}{2}) + r \qquad (\dot{y_{ref}} = \omega r \cos(\omega t - \frac{\pi}{2}))$$

  Where r is the radius of the circumference and ω is the angular velocity.
  Since the robot starts at (0,0), a phase shift of $\frac{\pi}{2}$ and an offset of r on the y axis are needed so to have a continuos trajectory (to have it starting at (0,0) when $t = 0$).

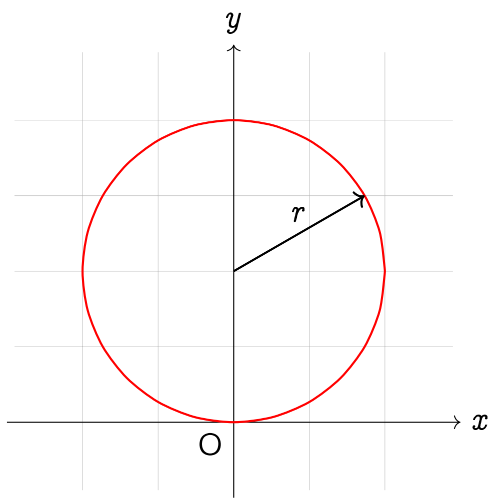


Figure 5.6: Circular trajectory (where $r = 2$)

- Figure Eight, an eight shaped path described by equations:

$$x_{ref} = a\sin(\omega t) \qquad (\dot{x_{ref}} = wa\cos(\omega t))$$

$$y_{ref} = a\sin(\omega t)\cos(\omega t) \qquad (\dot{y_{ref}} = wa[\cos(\omega t)^2 - \sin(\omega t)^2])$$

  Where a is the trajectory amplitude (the eight shape goes from $-a$ to a [m]) and ω is the angular velocity;

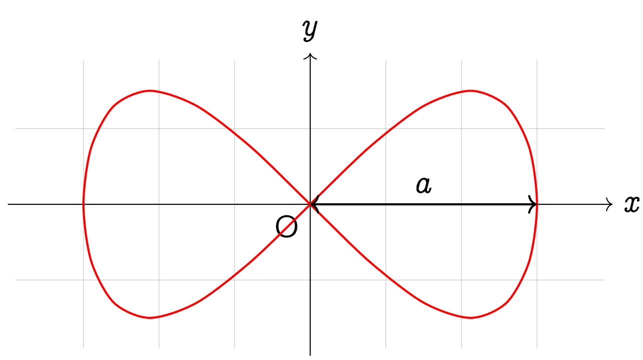


Figure 5.7: Figure eight trajectory (where $a = 3$)

- Curtate Cycloid, the path traced out by a fixed point at a radius $d < r$, where r is the radius of a rolling circle. It is described by equations:

$$x_{ref} = rt - d\sin(t) \qquad (\dot{x_{ref}} = r - d\cos(t))$$

$$y_{ref} = d - d\cos(t) \qquad (\dot{y_{ref}} = d\sin(t))$$

Where r is the radius of the rolling circle, and d is the distance of the point drawing the cycloid from the center such circle ($d < r$ to have a curtate cycloid).

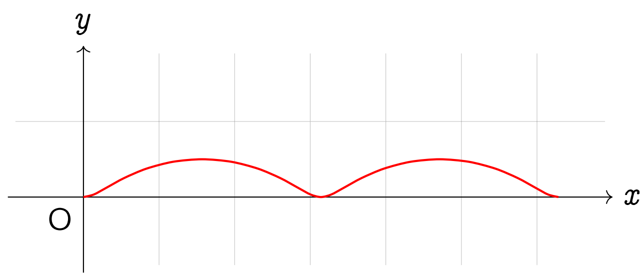


Figure 5.8: Cycloidal trajectory (where $r = 0.5$ and $d = 0.25$)

- **Reference Transformation** which is in charge of performing a coordinate transformation from robot frame (attached to its center of gravity) to the P frame, where P is the point around which we are performing the feedback linearization of the bicycle model.
Since the trajectory will be tracked by point P, and not by the COG of the robot, we have to apply a reference transformation to every trajectory setpoint in order to achieve the desired behaviour.
The required transformation is:

$$x_{P_{ref}} = x_{ref} + \overrightarrow{PL} \cdot \cos(\theta)$$

$$y_{P_{ref}} = y_{ref} + \overrightarrow{PL} \cdot \sin(\theta)$$

Where $x_{P_{ref}}$ and $y_{P_{ref}}$ are the setpoints for point P, x_{ref} and y_{ref} are the setpoints generated by the *Trajectory Generator*, $\overrightarrow{PL}$ is the distance between point P and the COG of the robot, and θ is the robot velocity direction (yaw of the robot).

- **Output Transformation**, for the same reasons we introduced the *Reference Transformation* we need an analogous transformation for the robot odometry data obtained through the "$\backslash vesc \backslash odom$" topic: from the coordinates of the COG we need to retrieve the coordinates of point P which are the control variables of the *Trajectory Tracker*.
The required transformation is:

$$x_P = x + \overrightarrow{PL} \cdot \cos(\theta)$$

$$y_P = y + \overrightarrow{PL} \cdot \sin(\theta)$$

Where x_P and y_P are the coordinates of point P, x and y are the coordinates of the robot COG, $\overrightarrow{PL}$ is the distance between point P and the COG of the robot, and θ is the robot velocity direction (yaw of the robot).

- **Trajectory Tracker** which is a controller in charge of performing trajectory tracking on point P.
Since we have two independent process variables x_P and y_P, we have two independent PID (Proportional-Integral-Derivative) controllers to achieve the trajectory tracking task.
The *Trajectory Generator* generates both position and velocity set-points, thus we can use the latter as feed forward terms to increase the stability of the trajectory tracking controllers.
The position errors are computed as:

$$x_{P_{err}} = x_{P_{ref}} - x_P$$

$$y_{P_{err}} = y_{P_{ref}} - y_P$$

The control actions are:

$$v_{P_x} = \dot{x_{P_{ref}}} + K_p \cdot x_{P_{err}} + K_i \cdot x_{int} + K_d \cdot x_{der}$$

$$v_{P_y} = \dot{y_{P_{ref}}} + K_p \cdot y_{P_{err}} + K_i \cdot y_{int} + K_d \cdot y_{der}$$

Where $\dot{x_{P_{ref}}}$ and $\dot{y_{P_{ref}}}$ are the feed forward terms, x_{int} and y_{int} are the integrals of the variables, approximated using Riemann sums:

$$x_{int_t} = x_{int_{t-1}} + x_{P_{err}} \cdot \Delta t$$

$$y_{int_t} = y_{int_{t-1}} + y_{P_{err}} \cdot \Delta t$$

Lastly, x_{der} and y_{der} are the derivatives of the errors, computed using finite differences:

$$x_{der} = \frac{x_{P_{err_t}} - x_{P_{err_{t-1}}}}{\Delta t}$$

$$y_{der} = \frac{y_{P_{err_t}} - y_{P_{err_{t-1}}}}{\Delta t}$$

The PID controllers are identical for both process variables, here we present the schematic of the PID controlling x_P:

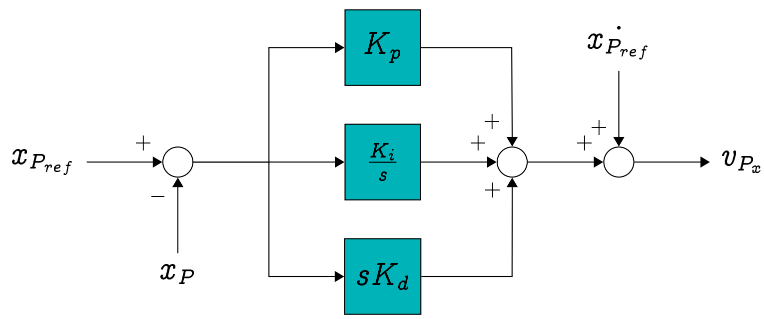


Figure 5.9: PID schematic for controlling x_P

5.6.2 Configuration

The node can be configured through the parameters present in the *trajectory_tracker.yaml* configuration file, in particular we have:

Parameter Name	Values Domain	Description
trajectory_type	$0, 1, 2, 3, 4$	Desired trajectory shape. Values: **0** linear trajectory; **1** parabolic trajectory; **2** circular trajectory; **3** eight-shape trajectory; **4** cycloidal trajectory.

a_coeff	$\mathbb{R}$	The line describing the linear trajectory is parallel to the direction vector: $\vec{d} = (a\_coeff, b\_coeff)$ (only used when *trajectory_type*=0)
b_coeff	$\mathbb{R}$	
focal_length	$\mathbb{R}$	Focal length 'a' [m] of the parabolic trajectory described by equation $y = ax^2$. (only used when *trajectory_type*=1)
R	$\mathbb{R}^+$	Radius of the circular trajectory [m]. (only used when *trajectory_type*=2)
W	$\mathbb{R}^+$	Angular velocity of the circular trajectory [rad/s]. (only used when *trajectory_type*=2)
a	$\mathbb{R}^+$	Amplitude of the eight shape trajectory [m]. (only used when *trajectory_type*=3)
w	$\mathbb{R}^+$	Angular velocity of the eight shape trajectory [rad/s]. (only used when *trajectory_type*=3)
cycloid_radius	$\mathbb{R}^+$	Radius of the wheel [m]. (only used when *trajectory_type*=4)
cycloid_distance	$\mathbb{R}^+$	Distance from the center of the wheel to the point drawing the cycloid ($d < r$) [m]. (only used when *trajectory_type*=4)
Kp	$\mathbb{R}^+$	Proportional gain for both PID controllers

Ki	$\mathbb{R}^+$	Integral gain for both PID controllers
Kd	$\mathbb{R}^+$	Derivative gain for both PID controllers
FFWD	$0, 1$	Feedforward flag for both PID controllers. Values: **0** disable feedforward component; **1** enable feedforward component;
PL_distance	$\mathbb{R}^+$	Distance from the odometric centre of the robot to the selected point P used for the linearization

5.6.3 Dynamic reconfigure

The package uses *dynamic reconfigure* to update some parameters at runtime without having to restart the node. In particular, it can be used to change the PIDs configuration: K_p, K_i, K_d and FFWD values.
Furthermore, the user can also toggle a boolean flag called "active" to start the trajectory generation (and tracking) process at will.

5.6.4 Choiche of PID controller and parameters

Performing an empiric PID tuning, we achieved good performances with a P controller with feed forward enabled; thus setting:

$$K_p = 5 \qquad K_i = 0$$

$$K_d = 0 \qquad FFWD = true$$

5.7 Plugin gazebo_ros_pacejka

5.7.1 Intro

We have added this plugin to allow the model to simulate the interaction between the wheels and the underlying terrain using the Pacejka Magic Formula. The interaction between the material of the wheels and the road surface gives rise to a deformation of the wheel contact surface and consequently to forces and torques acting on the vehicle: there are different models to define these quantities resulting from tire-road interactions. We have implemented the Magic Formula defined by Hans B. Pacejka.

5.7.2 Pacejka Magic Formula

The tire-road interaction gives rise to six resultants:

- F_x - Longitudinal Force
- F_y - Lateral Force
- F_z - Normal Force (Vertical Load)
- M_x - Overturning Moment
- M_y - Rolling Resistance Moment
- M_z - Self-aligning torque

Within our plugin, we have only calculated the resulting F_x and F_y forces. The reason for this choice stems from the adjustable parameters in Gazebo: for each calculated velocity and steering angle, the $slip1$ and $slip2$ parameters are modified, corresponding to the amount of slip in the two reference directions x, y of the vehicle. These values depend on the forces acting on that portion of the contact surface at a given moment in time.

Longitudinal force F_x

The longitudinal force according to the Pacejka Magic Formula depends on the longitudinal slip, which determines how far the wheel deviates from pure slipping behavior during a longitudinal displacement (i.e., a displacement that depends only on the x-axis).

The formula used to compute the longitudinal slip is:

$$\sigma_x = \frac{V_x - r_e\Omega}{r_e\Omega}$$

The result of this computation is then costrained to satisfy the condition: $|\sigma_x| < 1$. To avoid errors due to the calculator's approximation, we have introduced another check on σ_x. When it reaches values very close to zero (less than machine epsilon) the longitudinal slip is set to the smallest positive double value for the system. After calculating the longitudinal slip, we used the Pacejka formula to compute F_x.

$$F_x = D_x \sin(C_x \arctan(B_x\sigma_x - E_x(B_x\sigma_x - \arctan(B_x\sigma_x))))$$

Lateral Force F_y

The lateral force according to the Pacejka Magic Formula depends on the lateral slip, which determines the angle between the direction in which the wheel is moving and the direction it is pointing.
The formula used to compute the lateral slip is:

$$\alpha = -\arctan\frac{V_{c_y}}{V_{c_x}}$$

The value returned by this expression is in radiants. We have constrained this value to the boundaries $[-\frac{\pi}{2}, \frac{\pi}{2}]$. If the α value is lower than the machine epsilon value, we have adjusted it to be no less than the minimum positive double value in the system. After calculating the lateral slip, we used the Pacejka formula to compute F_y.

$$F_y = D_y \sin(C_y \arctan(B_y\alpha - E_y(B_y\alpha - \arctan(B_y\alpha))))$$

Parameters of the Pacejka Magic Formula

Applying the Pacejka Magic Formula also requires determining the parameters used within the formula itself. To calculate these parameters, we need to compute the wheel's normal force F_z: we have done this by considering the total weight of the vehicle and evenly distributing the entire weight among the four wheels.

$$M_{tot} = M_{chassis} + 4M_{wheel} + M_{steering-hinge} + M_{laser} =$$
$$= 4.0 + 4 * 0.34055 + 0.100 + 0.130 = 5.5922\ [kg]$$
$$g = 9.8065\ [m/s^2]$$
$$F_z = \frac{M_{tot}g}{4} = 13.71\ [N]$$

Initially, we roughly calculated the parameters B, C, D, and E using the theoretical specifications from the documentation related to the Magic Formula. Subsequently, we carried out an experimental tuning process of the parameters to enhance the model's performance. For the sake of calculation simplicity, we assumed that the vehicle had no additional load beyond its own weight; therefore, the formulas used to calculate the parameters are simplified.

- C **Shape factor**
  This parameter depends on the cross-section of the wheel's geometry. In the racecar model, the wheels are defined as cylinders with a radius
  $$R = 0.05\ [m]$$
  and length
  $$L = 0.045\ [m]$$
  . We compute the shape factor with the formula:
  $$C = \frac{L + 2R}{2R}$$

- D **Peak factor**
  The peak value of the force generated due to tire-road interaction.
  $$D = F_z(b_1 + F_z b_2)$$
  Where:

- F_z is the wheel's normal force $[kN]$
- b_1 is the load influence on the longitudinal friction coefficient($*1000$)
In our case $b_1 = 0$ (no load assumption).
- b_2 is the longitudinal friction coefficient($*1000$)
In our case b_2 varies depending on the considered wheel. The longitudinal friction coefficient and the reference direction are defined for each wheel within the $racecar.gazebo$ file (mu_1, mu_2, and $fdir_1$). Therefore, depending on whether the wheels are front or rear, we will have two values D_x and D_y with different values.

- BCD **Cornering stiffness**
It is the slope at the origin of the curve between force and slip. It is particularly important to allow for a linear approximation of the curve.

$$BCD = (b_3 {F_z}^2) + (b_4 F_z) e^{-b_5 F_z}$$

Where:

- F_z is the wheel's normal force $[kN]$
- b_3 is the curvature factor of stiffness/load
In our case $b_3 = 0$ (no load assumption).
- b_4 is the change of stiffness with slip $[N/\%]$
This parameter concerns the change of stiffness with slip. It is a physical property, how much an object resists deformation. The stiffer the object is, the more resistant to deformation it is.
- b_5 is the change of progressivity of stiffness/load
In our case $b_5 = 0$ (no load assumption).

We computed the parameter BCD using the reference values provided by the Pacejka Magic Formula documentation and performing parameter tuning operations.

- B **Stiffness factor**
The stiffness factor is a measure of the resistance of a material to deformation. The higher the stiffness factor, the more resistant the

material is to deformation.

$$B = \frac{BCD}{CD}$$

We computed the parameter B using the reference values provided by the Pacejka Magic Formula documentation and performing parameter tuning operations.

- **E Curvature factor**
  The parameter E controls the shape of the nonlinear force curve with respect to the slip. It influences the curvature of the force curve and helps to model the asymmetry of tire behavior for positive and negative slips.

  $$E = (b_6 {F_z}^2 + b_7 F_z + b_8)(1 - b_{13} sign(slip + H))$$

  Where:

  - F_z is the wheel's normal force $[kN]$
  - b_6 is the $curvaturechangewithload^2$
    In our case $b_6 = 0$ (no load assumption).
  - b_7 is the curvature change with load
    In our case $b_7 = 0$ (no load assumption).
  - b_8 is the curvature factor
  - b_{13} is the curvature shift
  - H is the horizontal shift
  - $slip$ is the longitudinal or lateral slip factor

  We computed the parameter E using the reference values provided by the Pacejka Magic Formula documentation and performing parameter tuning operations.

Below, we have included the table with the values of all the parameters used within the formulas for the calculation of Fx and Fy, respectively.

	Longitudinal Force $[F_x]$	Lateral Force $[F_y]$
B	3	10
C	1.45	1.45
D	19.0569	1.371
E	0.52	0.97

The value of these parameters was calculated and tested specifically for the "eight trajectory".
Using these values for other trajectories could lead to unexpected behavior. This is because the parameters of the formula also depend on the angle of curvature that the wheels must sustain during the path.
Optimum values should, therefore, be assigned experimentally for each trajectory.

5.7.3 Plugin description

Our goal was to modify the physical parameters that govern the dynamics of the interaction between the vehicle and the terrain in real time during the simulation. To achieve this, we had to directly modify the parameters in the simulator using the updated velocities and angles in the ROS model. It was necessary to develop a plugin that could receive information from the model's controller and communicate it in real-time to the simulator. After some research, we have decided to use it as a base and modify the two plugins, *GazeboRosWheelSlip* and *WheelSlipPlugin* developed to modify slip compliance in the two fundamental directions of a vehicle with one or more wheels. While we kept the *GazeboRosWheelSlip* plugin practically unchanged, we substantially modified *WheelSlipPlugin*. We have included all the computations necessary for the application of Pacejka formulas. For this reason, we wanted to differentiate the two plugins by changing the name, from *WheelSlipPlugin* to *wheel_slip_pacejka*.

Plugin configuration

The plugin must be called within the configuration file, in our case *racecar.gazebo*.
For each wheel, we have assigned a *wheel* tag in which, through sub-tags, we have passed the values that the plugin needs to perform all computations.
In particular, each *wheel* tag includes:

- **slip_compliance_lateral**
  Unitless slip compliance ($slip/friction$) in the lateral direction.
- **slip_compliance_longitudinal**
  Unitless slip compliance ($slip/friction$) in the longitudinal direction.
- **wheel_normal_force**
  Normal force value impressed on the wheel.
- **wheel_radius**
  Radius $[m]$ of the cross-section of the wheel's geometry.

All these values are applied to all wheels declared in the Element_Ptr passed firstly to $GazeboRosWheelSlip$ and secondly to $wheel\_slip\_pacejka$ plugin.

gazebo_ros_pacejka

This plugin was necessary to interface our ROS model with the Gazebo plugin wheel_slip_pacejka, and permit the exchange of values to update and modify the slip compliance values. This plugin calls directly the *Load* function of wheel_slip_pacejka, so it does not add any computation to the slip compliance, it only handles the communication between ROS and Gazebo.

wheel_slip_pacejka

This is a plugin that updates ODE wheel slip parameters based on linear wheel spin velocity $radius * spin\_rate$. The ODE slip parameter is documented as force-dependent slip ($slip_1$, $slip_2$) in the ODE user guide, it has units of $[velocity/force] = [m/s/N]$, similar to the inverse of a viscous damping coefficient. The $slip\_compliance$ parameters specified in this plugin are unitless, representing the lateral or longitudinal slip ratio to the force value. At each time step, these compliances are multiplied by the linear wheel spin velocity and divided by the wheel_normal_force parameter specified below to match the units of the ODE slip parameters. We used these formulas as already present in the $WheelSlipPlugin$, but with the forces computed by using Pacejka Magic Formula and the slip computed with the formulas presented above in this section.

$$slip_1 = \frac{\frac{\sigma_x}{F_x} * spin\_speed}{F_z}$$

$$slip_2 = \frac{\frac{\alpha}{F_y} * spin\_speed}{F_z}$$

To correctly apply friction dynamically, we also had to modify the coefficients of friction along the first and second friction direction, μ and μ_2.
Whenever we evaluate the friction forces employing Pacejka's formulae, we are also adjusting these values. In this fashion, Gazebo's internal process will apply the fitting friction to the wheels, calculated utilizing consistent parameters.
The expressions we have used to calculate the friction coefficients use the longitudinal or lateral force, depending on the direction chosen, normalized with the force normal to the wheel.

$$\mu = \frac{F_x}{F_z}$$

$$\mu_2 = \frac{F_y}{F_z}$$

Chapter 6

Python Scripts

We developed two python scripts that plot useful graphs to quickly visualize the system performance, and that compute error metrics to objectively compare different test instances.

6.0.1 Metrics

Let $q(\vec{t}) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$ be the position of the robot center of mass in global coordinates at time t, and $q_{r\vec{e}f}(t) = \begin{bmatrix} x_{ref}(t) \\ y_{ref}(t) \end{bmatrix}$ be its target position at time t, we define the error vector $\vec{e}$ at time t as:

$$e(\vec{t}) = \begin{bmatrix} x(t) - x_{ref}(t) \\ y(t) - y_{ref}(t) \end{bmatrix}$$

We neglect the heading θ in the error computation since, having used the feedback linearization law to control the system, we have a loss of observability: we no longer can control the robot heading; however, we know that, starting from any initial condition, the car will automatically align itself with the desired trajectory.
To quantify the controlled system performance we chose the following two indicators:

- Root Mean Square Error (RMSE) which is computed as the square root of the mean square error:

$$RMSE = \sqrt{\frac{1}{T}\sum_{t=0}^{T} e(\vec{t})^{\intercal} e(\vec{t})}$$

It is a measure of the mean error that we have at each time instant;

- Integral Square Error (ISE) which is computed as the integration over the entire trajectory time T of the square of the error:

$$ISE = \int_T e(\vec{t})^{\intercal} e(\vec{t})\, dt$$

It is a measure of the total error accumulated over the entire trajectory.

6.0.2 Graphs

analyze_ bag.py script plots three graphs:

- a 2D representation of the robots center of mass trajectory (continuous blue line) together with the desired trajectory (dotted orange line);
- actual and desired center of mass trajectory along x-axis over time;
- actual and desired center of mass trajectory along y-axis over time.

When using Pacejka tire-ground interaction model, we can use *analyze_ speed_ bag.py* script that draws the three aforementioned plots plus two more:

- the longitudinal force F_x acting on the wheel contact point with respect to the longitudinal slip;
- the lateral force F_y acting on the wheel contact point with respect to the lateral slip.

6.0.3 Scripts usage

The typical use case of the python scripts is to, first, record a ROS bag during the simulation; it can be done using command:

```
$ rosbag record -O out.bag --duration=1m /vesc/odom
    /reference_trajectory
```

Or, if we are using Pacejka tire-ground interaction model, we need to record additional data with the following:

```
$ rosbag record -O out.bag --duration=1m /vesc/odom
    /reference_trajectory /long_pub/right_front
    /lat_pub/right_front /fx_pub/right_front
    /fy_pub/right_front
```

Then, we can simply launch the desired python script passing the bag file as parameter:

```
$ python3 analyze_bag.py -O out.bag
```

or

```
$ python3 analyze_speed_bag.py -O out.bag
```

Please note that in order to use *analyze_speed_bag.py*, the bag file must have been recorded using the command above.

Chapter 7

Conclusions

On completion of the implementation of all the modules presented in the previous sections, we compared the machine's behavior in different applications.
There were four test configurations:

- Kinematic linearizer + Gazebo Physics
- Kinematic linearizer + ODE Physics
- Kinematic linearizer + Gazebo Physics + Pacejka friction model
- Kinematic linearizer + ODE Physics + Pacejka friction model

The experimental results obtained were consistent with theoretical expectations.
In this chapter, we illustrate two general considerations from the execution of each test case.

7.0.1 Original Gazebo model vs. ODE model

As it turned out, with the same modules applied, these two models gave very similar results.
This is due to the fact that the Gazebo simulator uses the ODE (Open Dynamics Engine) library as its physics engine to simulate rigid-body dynamics.

Since these two models are based on the same physics engine, there were no differences during the application.

7.0.2 Pacejka friction model

The testing phase of the Pacejka module provided results consistent with what was expected from the theory. By defining the graphs between force and slip, it can be seen that the data collected takes the form of a sigmoid function.
Theoretically, Pacejka's application and the related machine behaviour are consistent.
Since no empirical data is available, we cannot ascertain whether our model works optimally. To be able to compare the different models, we would need a basic model to refer to and set as ground truth.
The results presented below were collected by applying the eight trajectory.

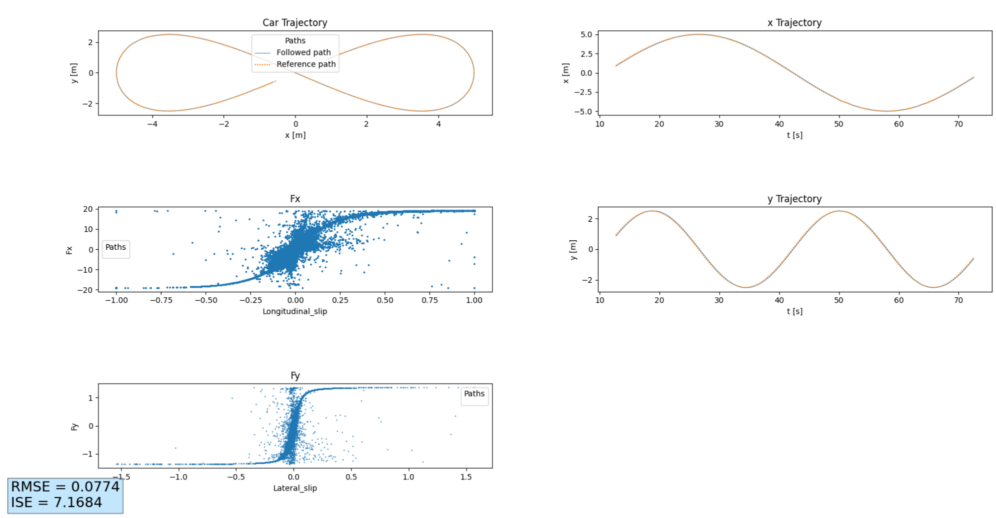


Figure 7.1: The eight trajectory simulated using Pacejka friction and Gazebo physics models

We would also like to present the results obtained by applying the Pacejka modulus correlated with the machine's rectilinear trajectory.

As can be seen from the force graph, the values are scattered and not as precise as in the case of the eight trajectory. The reason is related to the parameters used in the calculation of the Pacejka formula. The values chosen were experimentally tested on the eight trajectory, as specified in the section on the plugin.

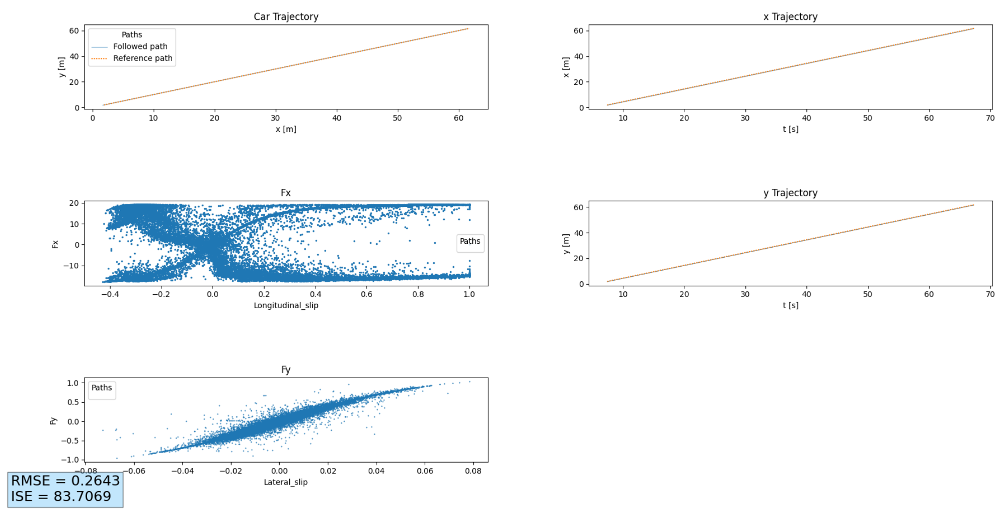
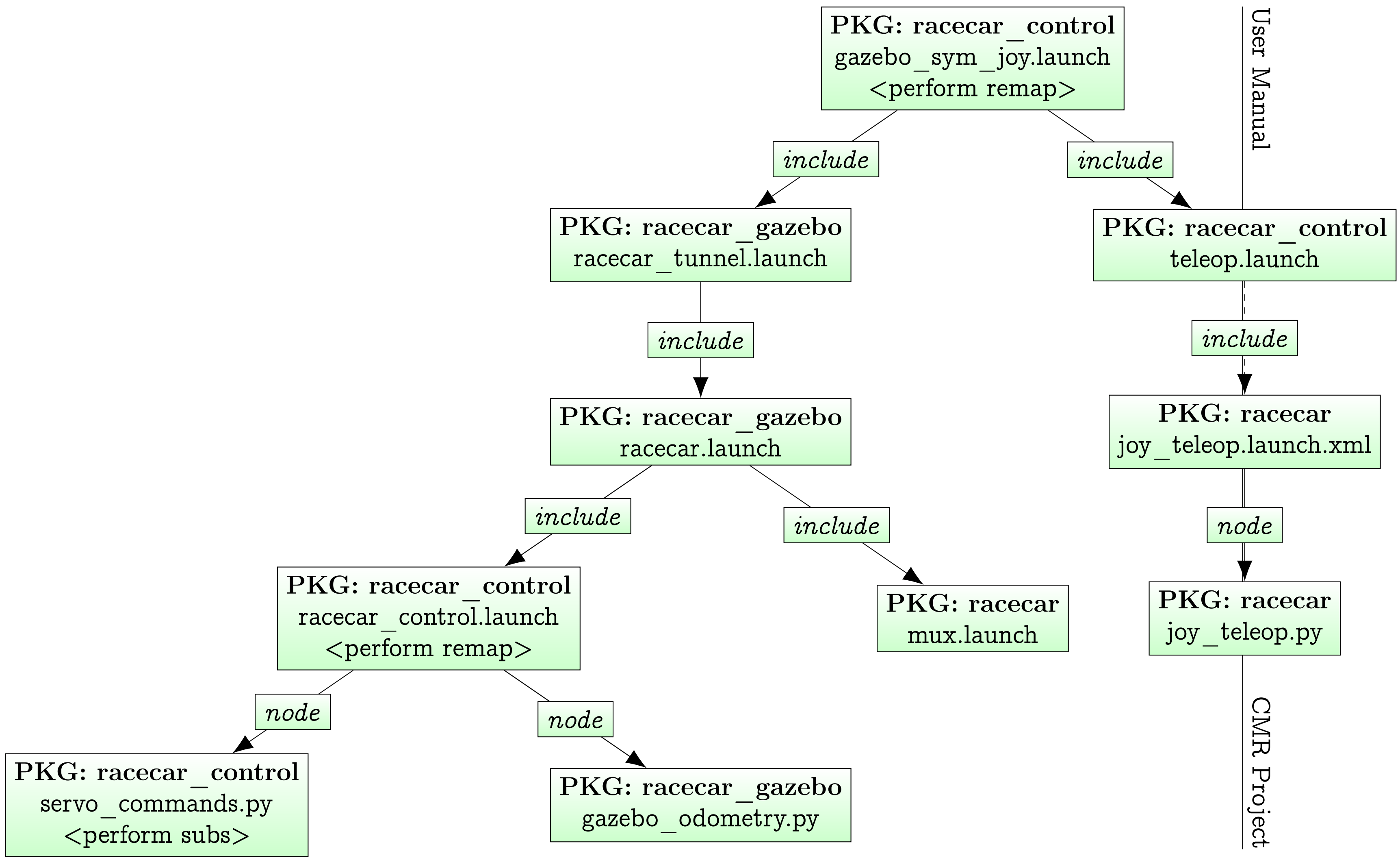


Figure 7.2: The rectilinear trajectory simulated using Pacejka friction and Gazebo physics models

Appendix A

launch package inclusion


PKG: racecar_control
gazebo_sym_joy.launch
<perform remap>
include
include
PKG: racecar_gazebo
racecar_tunnel.launch
PKG: racecar_control
teleop.launch
include
include
PKG: racecar_gazebo
racecar.launch
PKG: racecar
joy_teleop.launch.xml
include
include
node
PKG: racecar_control
racecar_control.launch
<perform remap>
PKG: racecar
mux.launch
PKG: racecar
joy_teleop.py
node
node
PKG: racecar_control
servo_commands.py
<perform subs>
PKG: racecar_gazebo
gazebo_odometry.py

PKG: racecar_control
keyboard_teleop.py
<perform publish>

Package	**File**	**Remap**
racecar	joy_teleop.launch.xml	(none)
racecar	joy_teleop.py	(none)
racecar	mux.launch	(none)
racecar_control	gazebo_sim_joy.launch	REMAP /ackermann_cmd_mux/input/teleop TO /racecar/ackermann_cmd_mux/input/teleop
racecar_control	teleop.launch	(none)
racecar_control	racecar_control.launch	REMAP /racecar/ack/output TO /vesc/low_level/ack/output
racecar_control	servo_commands.py	SUBSCRIBE /racecar/ackermann_cmd_mux/output
racecar_control	keybpard_teleop.py	PUBLISH /vesc/achermann_cmd_mux/input/teleop
racecar_gazebo	racecar_tunnel.launch	(none)
racecar_gazebo	racecar.launch	(none)
racecar_gazebo	gazebo_odometry.py	(none)