Polytechnic University of Turin

master's degree in mechatronic engineering

Development of a motion planner framework for autonomous driving applications

Academic advisor
Prof. Massimo Violante

Candidate
Alberto Santagostino

A.A. 2018/2019

Abstract

This project aims to describe the integration of a motion planner in an existing environment developed for autonomous driving applications. The framework used is based on ROS and exploits different technologies and algorithms for guidance, navigation and path planning. While the final step of the validation is the testing on a real vehicle, the purpose of this thesis is to implement the motion planner in Simulink, run a series of simulations using a virtual vehicle in CarMaker and perform an evaluation using the obtained results.

Contents

1 Introduction and technologies used

The target of this work is the design and the development of a **motion planner** module for autonomous driving applications.
This thesis is the second part of a project started by other students that aims to realize an autonomous driving framework in its entirety. The first part, the module responsible for **data fusion** and **path planning**, was already completed[1] and will be object of future improvements to extend its functionalities.

The input of the motion planner is the output of the path planning module, following the classic architecture for the control of a wheeled mobile vehicle.[2]
After the localization is performed, the navigation algorithms use information on the vehicle state, on the environment and on the target point to find a trajectory to follow. The defined path is then communicated to the motion planner module that performs the actuation (figure 1).
The final step needed to validate the controller is the simulation of its execution in a virtual vehicle and environment.

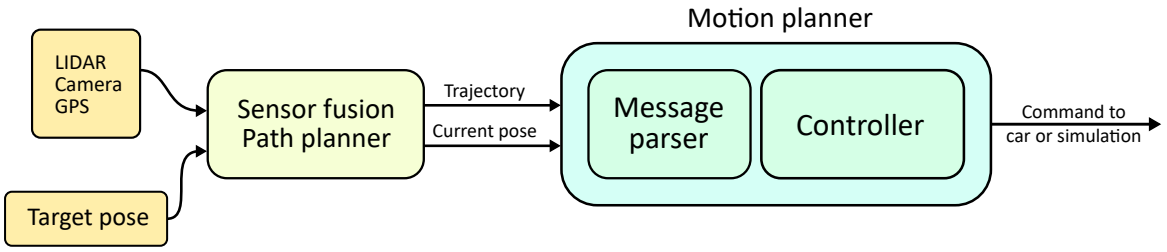


Figure 1: Architecture of a generic navigation and control system

To proceed by incremental steps with the development process this work has been divided into two parts:

1) **Validation of the infrastructure** (*Chapter 3*)
   In the first part, the possibility to integrate a motion planner in the existing ROS framework is analyzed and tested, working only with local interconnected ROS nodes, implementing a controller in MATLAB Simulink able to receive and send data to the framework.
   To perform this first implementation a simplified plant and a basic controller for wheeled vehicles are developed in a Simulink model. The trajectory provided to the motion planner is computed real-time by the ROS nodes running the path planner locally.

2) **Design of the controller and validation in CarMaker** (*Chapter 5*)
   In the second part, CarMaker is used to generate a simulated vehicle to drive. The controller is designed and deployed inside the vehicle model itself. The existing module responsible of the path planning is communicating to the controller through the ROS infrastructure developed in the first part, as the whole model is still executed in a Simulink environment.

Multiple technologies are used in this project:

- **ROS** is the framework used for communication that acts as infrastructure between all the developed modules.
- **MATLAB Simulink** is the modeling software used for the design, the implementation and the testing of the motion planner.
- The **robotics library** provided with Simulink is used as interface between MATLAB code and ROS components, to ease exchange of information between the two environments.
- **CarMaker** is used as simulation and validation tool, to perform tests and check the behavior of the algorithms using a virtual vehicle, performing multiple sets of test drives.

All the mentioned tools and technologies are treated in detail in the next sections.

1.1 ROS

ROS[3,4] (*Robotic Operating System*) is a framework created in 2007, mainly used in research and academic contexts for robotics applications. It provides an infrastructure on which a powerful and highly customizable communication layer can be built, implementing exchange of messages, subscriber/publisher logic, server/client operations. It makes use of `C++` and Python as core programming languages and its stable version is developed to run on Linux systems.

ROS is used in this project for several reasons:

- It simplifies exchange of messages through user-defined data types that can suit any needing.
- It is supported by MATLAB and its usage is simplified thanks to tools and blocks available in the Simulink robotics library.
- It makes the deployment of a ROS node to different hardware platforms easier, accelerating the development and testing process.

An additional reason ROS was chosen is continuity with the previous work. This thesis aims to develop a module that extends an already existing framework built on a ROS architecture, thus it makes sense to continue on this path. The exchange of messages between all the modules and the final deployment on a platform is much easier to perform if a common infrastructure is used in the whole system.

1.1.1 Nodes

ROS nodes are the basic executable instances that can be viewed as **processes** able to read inputs, execute operations and communicate with other nodes.
In a typical ROS architecture there are multiple nodes, each one of them performing a specific set of operations. The motion controller developed in this project is going to be deployed as one of these nodes, receiving inputs from other ones (the path planner) and sending control commands to the outside world (or to other nodes that take care of the actuation).

The usage of nodes in ROS is also increasing the degree of robustness and fault tolerance of the whole system: if an error occurs during the execution, it can be handled through routines implemented to detect problems and take care of the consequences, thanks to the decoupling between different processes.

The node **roscore** is a collection of standard processes required to run the ROS system and to enable basic diagnostics. It's usually the first process started to prepare and initialize the communication infrastructure.

1.1.2 Messages

Exchange of information is handled in ROS through messages (*rosmsg*), data containers that are sent from one node to the others. It is important to point out that nodes don't communicate directly with each other (one-to-one): a node publishes the message to broadcast on a **topic** (section 1.1.3), accessible to all the other ROS processes that

may be interested in that data or information, which then retrieve it and use it.
To make an example, let's imagine that a node assigned to the computation of trajectory information has just generated a new result. This node will compose and publish a message containing the relevant data to a topic used for communications regarding the path (for example the topic */trajectory_information*). Another node that needs this data will periodically check for new messages on */trajectory_information* and retrieve them.

ROS messages are defined in `.msg` files. A generic message is defined as a list of its contained variables, using the common syntax:

```
data_type variable_name
```

As an example, the definition of the message representing a point in a three-dimensional space is:

geometry_msgs**Point.msg**

```
float64 x
float64 y
float64 z
```

It is worth mentioning that messages can be nested to be reused in other message definitions, as new data types. This is the case of the one representing the pose of a physical object, that uses the point message previously shown:

geometry_msgs**Pose.msg**

```
Point position
Quaternion orientation
```

The specific message used to broadcast trajectory information is presented in section **1.3**.

1.1.3 Topics

While messages define the transmitted frame content, ROS topics are the software buses that connect different nodes. The logic used in this communication infrastructure relies on the concepts of **publisher** and **subscriber**. The definition of a new topic (or channel) is simultaneous to the creation of the publisher object that will post messages on it. While the publisher is behaving as a message sender, the subscriber is an object listening to a specific topic, able to receive all the messages broadcasted.
A topic is defined using the syntax:

```
/sample_topic
```

There might be multiple nodes that listen to the same topic, as shown in figure 2.

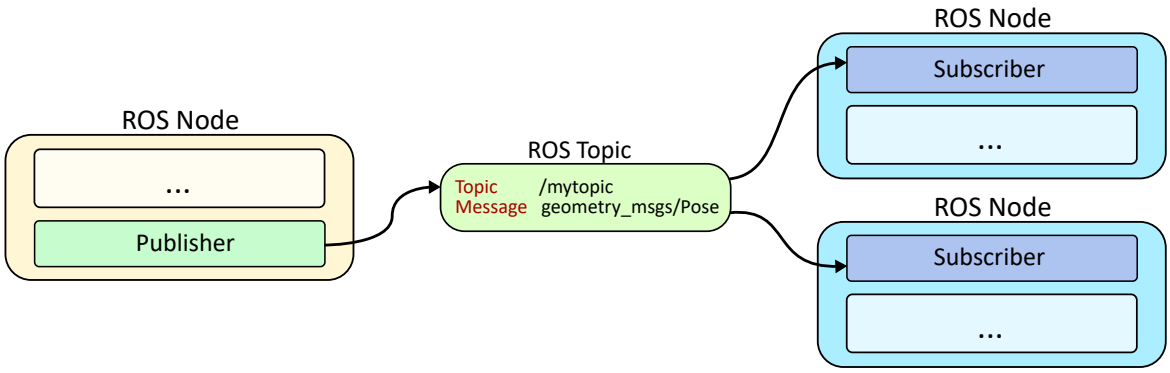


Figure 2: Example of communication in ROS via topics

1.1.4 Publishers and subscribers APIs

ROS is distributed with useful APIs to ease the job of developers. To handle publication and subscription to topics, classes like **ros::Publisher** and **ros::Subscriber** are available and well documented. While the first one accepts as parameters a defined message and a topic to which broadcast, the second waits for new messages on a topic and through a callback function executes actions at the reception of data.

1.2 Simulink robotics library

A library available in Simulink written to simplify the interface to ROS systems and projects is the *Robotics System Toolbox*. Inside this collection are available multiple blocks that implement basic functionalities related to the handling of messages and topics between nodes. The block used in this project is the **subscriber block** (figure 3).

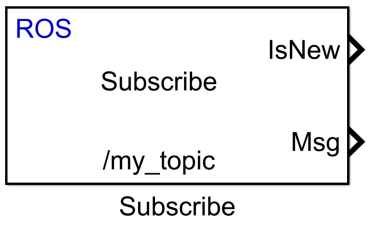


Figure 3: ROS subscriber block

An important thing to notice that has direct consequences on the Simulink model design is the *isNew* output port. This port emits the signal *true* when a new message is published on the selected topic. This port is the graphical equivalent of the callback function used in the ROS publisher API previously mentioned.

1.3 Definition of the trajectory input

The input of the motion planner is the target trajectory that the vehicle has to follow. The trajectory is fixed on the reference frame of the vehicle and when is computed by the path planner it is assumed that the current pose of the car is [0,0,0]. This means that every time a new trajectory is received, a new navigation problem needs to be solved. The system will react to the received trajectory considering its global position

and will apply a control command using as target the list of future waypoints. The error of the final trajectory with respect to the desired one will be expressed as **cross track error** (cte) and **heading error** (θ_e).

From the previous overview given on the ROS architecture it is clear that the target trajectory is a ROS message published on the topic */astar_path*. The message is of type *Path*.

nav_msgs**Path.msg**

```
Header header
geometry_msgs/PoseStamped[] poses
```

geometry_msgs**PoseStamped.msg**

```
Header header
Pose pose
```

Note that the trajectory is expressed as an array of poses, each one of them containing a target position and a target orientation (see also section 1.1.2).

At the current development stage the trajectory sent by the path planner node is containing only geometrical information. Other students are working on an advanced algorithm for the path planner that will introduce the possibility to assign to every point a target velocity to reach, making possible the avoidance of moving objects (other vehicles, pedestrians) and allowing new control techniques to be studied and implemented.

1.4 CarMaker

CarMaker is a software widely used for virtual test driving and vehicle simulation.

With CarMaker it is possible to generate a virtual vehicle and its surrounding environment with an extreme degree of detail and to run test drives to check the performances of the implemented control systems.

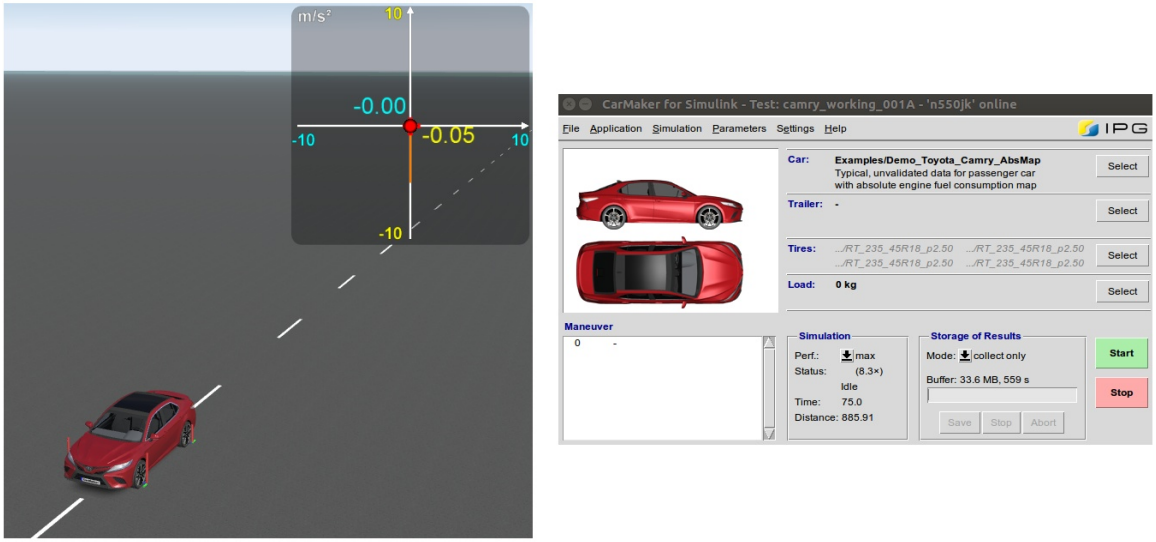


Figure 4: CarMaker simulated environment and GUI

With CarMaker it's also possible to recreate advanced events like driver interventions, malfunctions, sensor failures and other situations that could occur during a real test drive.

1.4.1 CarMaker for Simulink

CarMaker for Simulink is the integration of the vehicle simulation software into the MATLAB Simulink modeling environment. Thanks to the two platforms combined it is possible to generate a customizable simulated vehicle ready to be used in Simulink. The main result of the integration is that the virtual vehicle model is represented as a series of editable Simulink blocks. Thanks to this block structure the analysis of signals, the study of the simulated car and the implementation of a controller are really straightforward to perform.

In chapter 4 the internal structure of a CarMaker vehicle generated for Simulink is presented in detail.

2 Pathplanner overview and localization technologies

The path planner developed in the existing framework makes use of the A* algorithm to compute the trajectory to follow. To navigate the physical world a 2D **costmap** (section 2.3) is generated performing data fusion of different sensors.
This chapter contains an overview of the different technologies used in the navigation module.

2.1 Sensors for localization and environmental modeling

To perform pose estimation and environmental modeling multiple sensors were used. Some of them were included in the final implementation, others were exclusively used and evaluated in the testing phase.
To perform a good localization it was clear after different tests that the fusion of GPS, IMU and odometry data were sufficient to reach a sufficient degree of accuracy and coherency.
All the devices tested and used are presented in the next sections.

2.1.1 IMU

An IMU (*Inertial Measurement Unit*) is a device able to measure the dynamics of a body, computing its accelerations along the three axes.
It operates using a set of high accuracy accelerometers and gyroscopes and it can provide the output in m/s^2.
The advantage of these devices in localization systems is that they're self-contained, as they don't rely on any external source of data to estimate the pose of the body. The main downside is that they suffer from drifting errors with the increase of time. For this reason, if not coupled with other typologies of sensors to eliminate the bias, they may be extremely unreliable.

2.1.2 GPS

The most known and used technology to determine the position of an object in the world is GPS (*Global Positioning System*). Even if it's a known fact that the accuracy of a standard GPS sensor is not really high (order of magnitude of meters), it doesn't suffer from any incremental error with the passage of time, like an inertial unit does. Moreover, the latest GPS devices that use the L5 band have higher accuracy, being able to pinpoint an object with a positioning error of just 30 centimeters.
The module developed is able to extract GPS information both from external receivers or directly from the vehicle built-in device via CAN serial bus.

2.1.3 Odometry

Another classic way to estimate the traveled distance in automotive applications is through wheels odometry.
Thanks to rotary encoders directly mounted on the wheels it is possible to obtain the amount of revolutions completed and, knowing the radius of the tires, compute the covered distance. In the developed platform the odometry is another information that can be retrieved directly from the car built-in sensors via CAN.

2.1.4 LIDAR

LIDAR[5] (*Laser Imaging Detection And Ranging*) is a technique that makes use of a pulsed laser beam to determine the distance to an object. Devices that exploit this technology to provide environmental data are called **laser scanners**.
To perform measurements that include the distance of every surrounding object these devices are kept in constant rotation with high revolutional speed while they use multiple laser beams at the same time.
While the output obtained from the bounce of a single laser beam is the distance from a targeted object, normally it is also possible to obtain preprocessed data represented as a **point cloud**. A point cloud represents the surrounding objects in a single time instant through a collection of points in 3D space that share a common coordinate system.
To convert a point cloud into an useful representation of objects and obstacles, an operation called **clustering** needs to be performed. Through clustering, it is possible to identify among all the points which ones are part of the same object. Usually from this grouping it is possible to construct **bounding boxes** enclosing every single detected object (figure 5, 6).

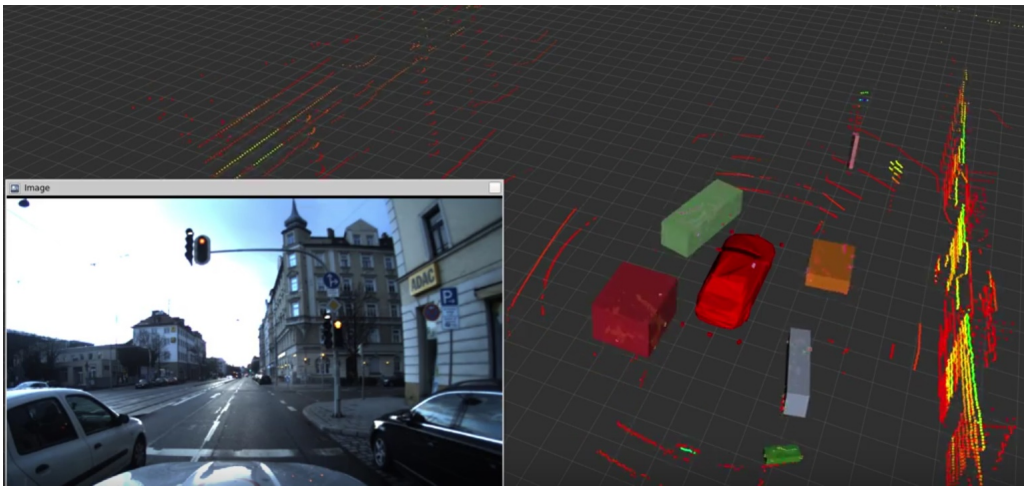


Figure 5: Test vehicle in the simulated environment (red car).
The other visible boxes represent identified objects

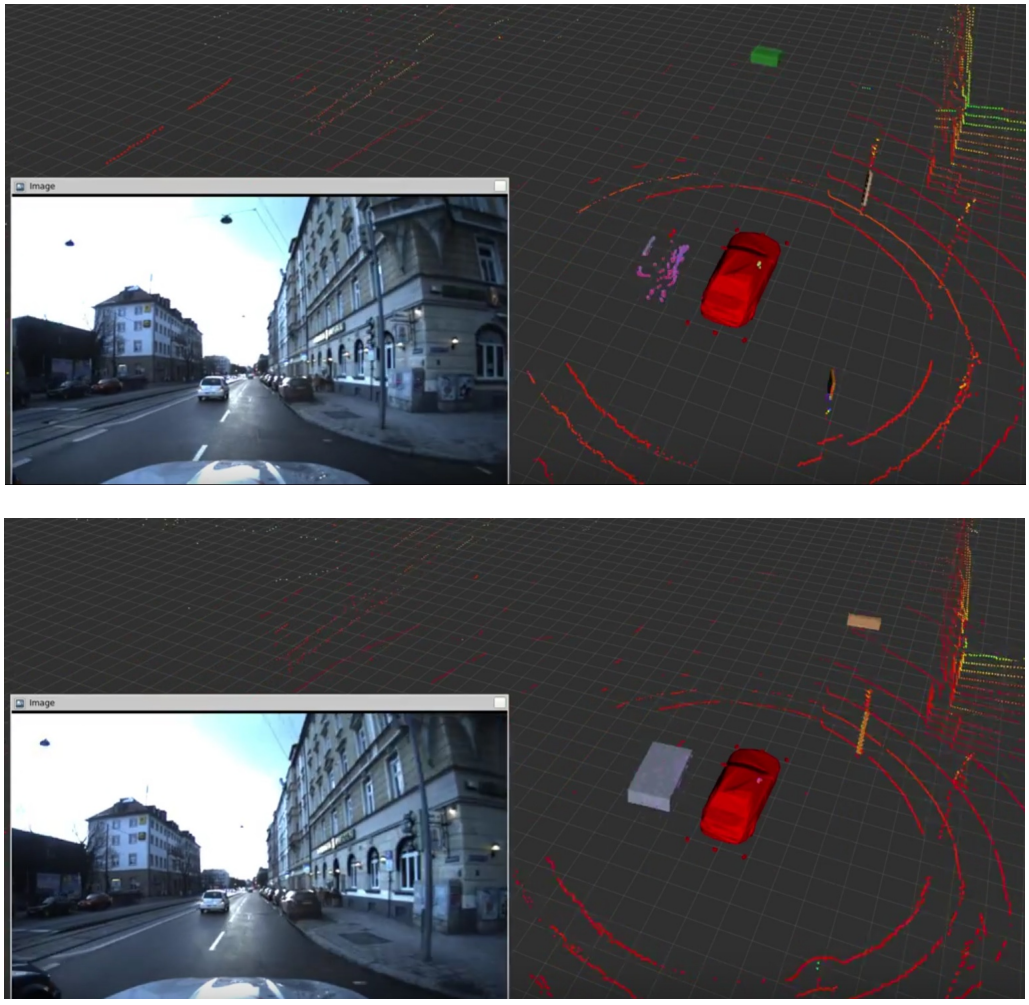


Figure 6: The set of points representing the car overtaking on the left is grouped in a solid gray box after clustering is performed

2.1.5 Camera

Another way to collect environmental information is through the usage of cameras, in different ways:

- Techniques like **visual SLAM** offer a really powerful method to perform localization and navigation at the same time.
- **Neural networks** like **YOLO** are modern methods that make possible to collect data on other vehicles and obstacles while simultaneously classify them.

Visual SLAM
SLAM[6,7] (*Simultaneous Localization And Mapping*) is a technique widely used in robotics that allows a robot to navigate an environment and at the same time to build a map of it. While this seems a really hard problem (*how to navigate an environment not fully discovered yet?*) solutions can be found using approximation methods like GraphSLAM, particle filter or EKF (*Extended Kalman Filter*).

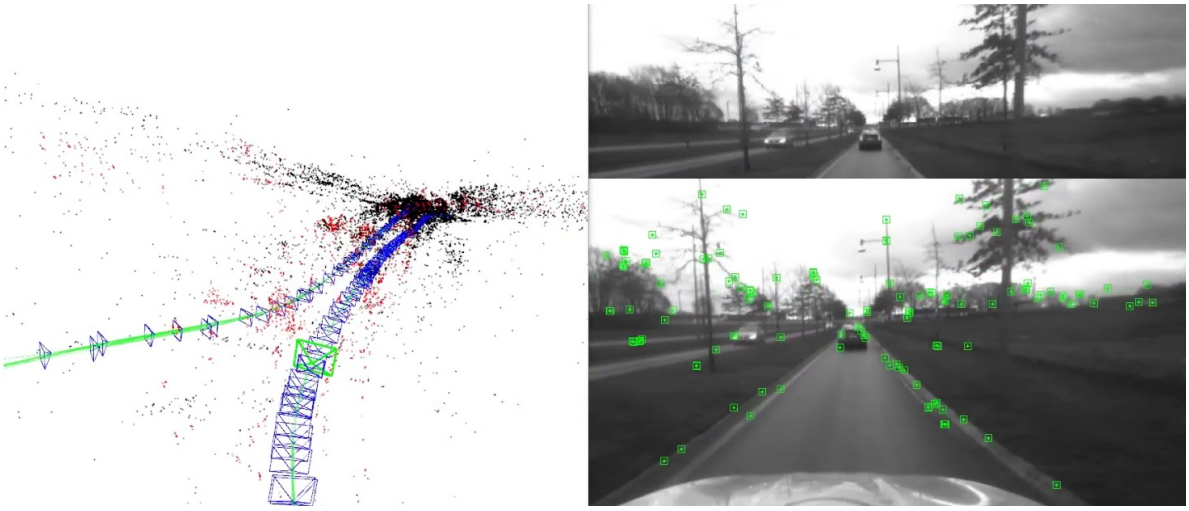


Figure 7: Visual SLAM used to map the road during a test drive. On the left side of the image the result of the mapping, updated in real-time during the navigation. On the right side the input from the front camera. Notice the green markers acting like anchors to keep track of the environment key points during the movement of the vehicle

Neural networks

Object recognition systems like YOLO[8,9] (*You Only Look Once*) are convolutional neural networks able to quickly recognize objects with a good precision.

An implementation of YOLO2 (the second version of the algorithm) was used in the developed platform, to classify the obstacles on the road and in close proximity to the vehicle.

Figure 8: Results from YOLO2 during test drives. Notice the boxes around the cars that indicate a successful object recognition

2.2 A* algorithm

A widely known and used algorithm for **pathfinding** and graph search is **A*** (*A star*). A* was first published[13] in 1968 and it is an extension of Dijkstra's algorithm, improved through the usage of heuristics methods to perform the search. It was chosen mainly due to its good performances compared with other graph navigation techniques.

2.2.1 Description of the A* algorithm

Starting from the first node of a graph, A* is able to find a path to the destination minimizing a cost function. The algorithm keeps an internal tree representation of all the paths starting from the nodes already visited and it propagates them until the search is ended and a valid result is found.
A big advantage of this path finding algorithm compared to others is that A* performs expansion only if it's reasonable to explore towards a certain direction. The only focus of the algorithm is to reach the target node as fast as possible starting from the current position.

The map of the environment itself is treated like a graph, where the nodes are the possible tiles that can be visited, as it will be explained in the next sections.

2.3 Costmap

A **costmap** is an environment representation that can be obtained after the clustering of obstacles as bounding boxes is performed. To obtain a costmap, bounding boxes are projected on a 2D surface, creating a grid in which empty spaces can be differentiated from tiles occupied by objects (figure 9), a model that can be easily parsed by a navigation algorithm.

The term *costmap* means that every tile in the map is associated to a single or multiple **cost values**. An example of cost could be the *occupancy probability*: instead of assigning to each tile a binary value representing the occupancy (0 = `not occupied`, 1 = `occupied`), a numeric value between 0 and 1 is assigned, based on the probability that an obstacle is not exactly where the bounding box is, but nearby, increasing the risk of collision. This additional value compensates for errors in the estimation of the obstacles position, decreasing the risk of collisions. During the construction of the costmap, an **occupancy grid** is generated using multiple cost values, to navigate the environment keeping track of obstacles and possible collisions. During the generation process some tiles are marked as non-navigable due to obstacles, after which a subprocess called **inflation** takes place, expanding the occupied area by a chosen factor, to consider additional occlusions in the real world that cannot be predicted with precision. Typically the chosen factor is proportional to the radius of the ideal circle in which the vehicle is inscribed.

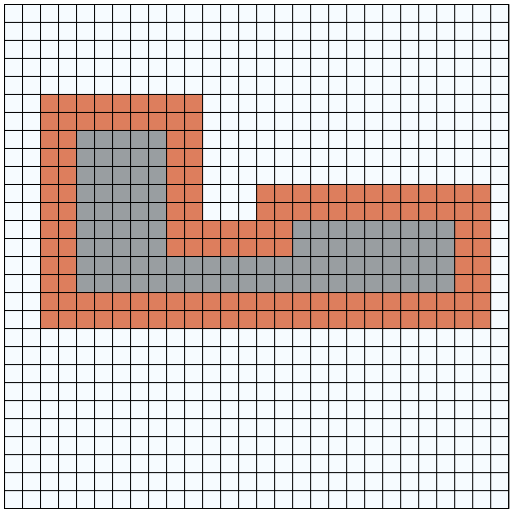


Figure 9: Example of a costmap, where the gray tiles represent a recognized obstacle, while the orange tiles are automatically added at the inflation step.
In this example the robot footprint can't navigate in the gray area, its center can't go in the orange tiles and its whole body can move freely in the white tiles

In the developed module, the package *costmap_2d* provided by ROS is used to generate and handle the costmap representing the environment. To convert information coming from the sensors data (point cloud) to virtual objects on the costmap, a ROS node called *euclidean_clustering* is executed. Using **rviz**, a visualization tool included in ROS, it is possible to observe the costmap with its obstacles and weights being generated and updated during the execution of the code.

For testing purposes it is possible to use a bitmap image to simulate a static costmap. Furthermore, it is possible to interact with the map, selecting interactively the initial and final pose of the vehicle to observe the trajectory computed by the path planner in real-time.

Practical applications of this technique are shown together with the performed tests in the next section.

2.3.1 Testing of path planner

A graphical tool included in ROS (rviz) is used as graphical interface to provide input poses to the system and to observe the generated paths (figure 10).

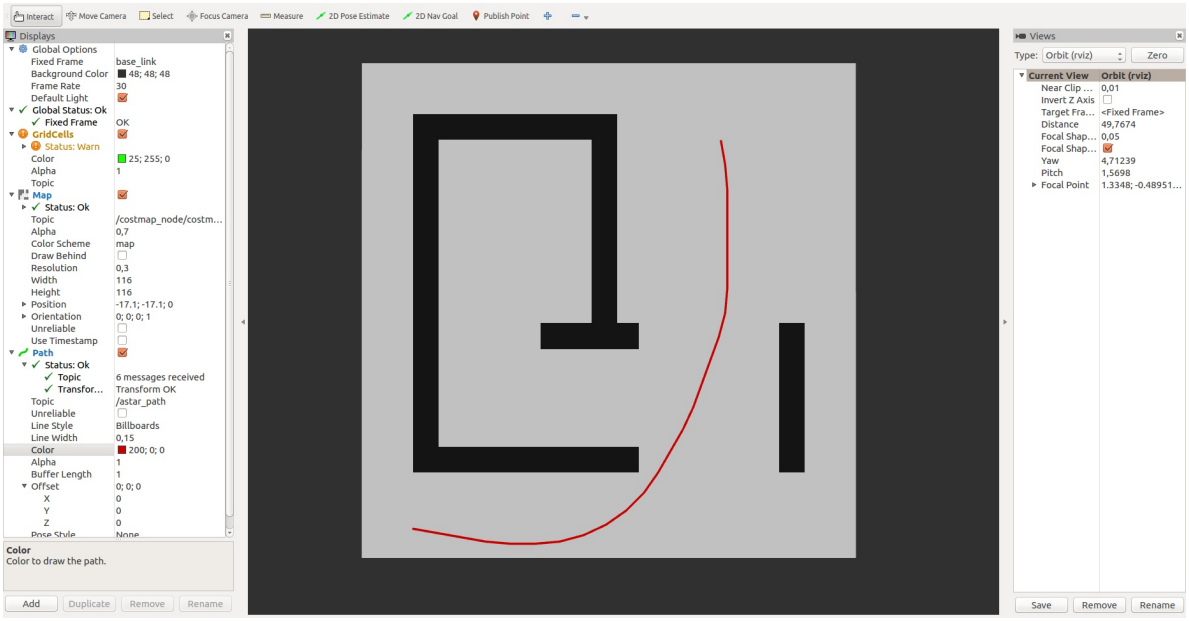
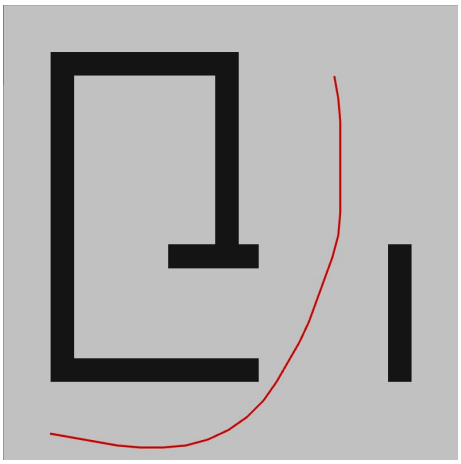


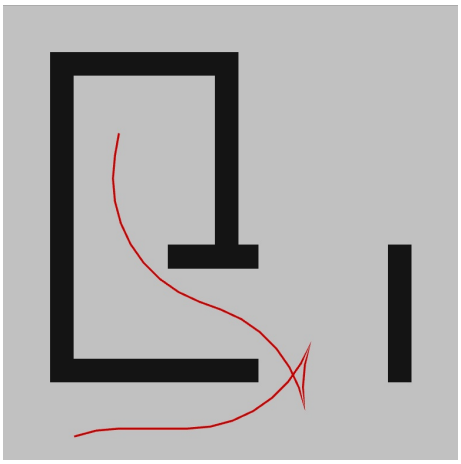
Figure 10: rviz interface

Using a manually defined costmap, various tests are performed. In every presented figure a different combination of initial and target poses is set and the resulting path can be visualized immediately after the calculation in rviz.

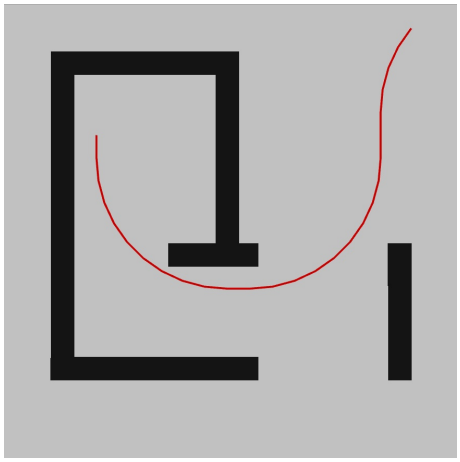
Test series 1



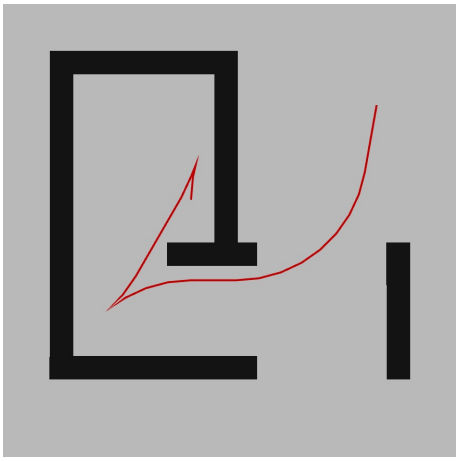
Test 1A



Test 1B



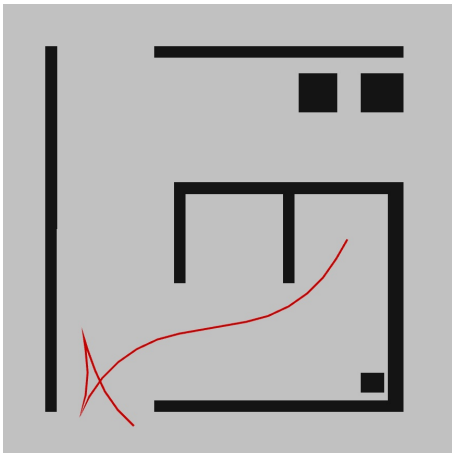
Test 1C



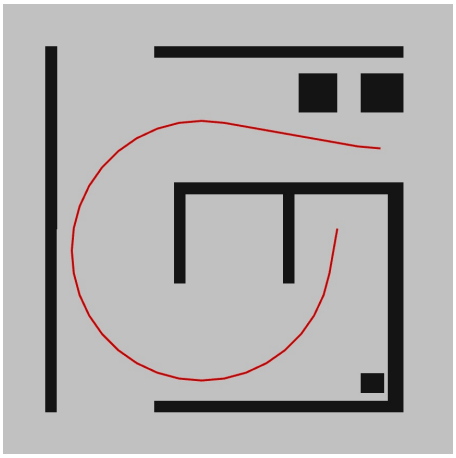
Test 1D

In tests **1B** and **1D** it can be observed that the computed path includes some sharp maneuvers. These maneuvers are the outcome of segments traveled in reverse gear. The algorithm is in fact able to understand if the destination can be reached with these kind of movements and to produce trajectories of this type.

Test series 2



Test 2A



Test 2B

Another slightly more complicated costmap is used to perform this other pair of tests. Also in this case it can be observed the previously described behavior: in test **2A** a section of the trajectory is traveled in reverse gear.

2.4 Execution of ROS nodes

In order to run the path planning algorithm and to perform testing in the interactive way described, different ROS nodes and services need to be executed in the first place.

- Execution of a `roscore` node
- Execution of `drawable_costmap` node
  This node activates the visualization of the costmap in rviz and enables the possibility for the user to interact with it, setting current and desired pose using the mouse input.
- Publication of string message on `load_image` topic
  In order to load a sample costmap from an existing image file, the local path of the image needs to be published as a string on this ROS topic through a publisher.
  It's also possible to change the message content while the whole algorithm is running, to simulate the navigation inside the environment.
- Execution of `freespace_planner` node (`astar_navi`)
  This node is the most important for the computation of the trajectory, being the one in which the path finding process is performed.
  The core of the module is the class `AstarSearch`, where the A* algorithm is implemented.

3 Infrastructure design and ROS parser development

Before proceeding with the design of the controller for the simulated vehicle in CarMaker, the new module for the framework needs to be developed and tested in a simplified scenario to validate the architecture in its entirety.

The objective to achieve in this phase is the design and the implementation of a Simulink model able to parse correctly the trajectory sent by the path planner node and to use it to control a simple simulated robot.

To perform the validation the already existing ROS nodes for the generation of the trajectory are used, together with rviz as visualization and input tool, a standalone Simulink model hosting a simple robot model and a static costmap with fixed obstacles configured. The evaluation aims to actuate a wheeled robot through the usage of a classic controller known as **pure pursuit controller**.[10–12]

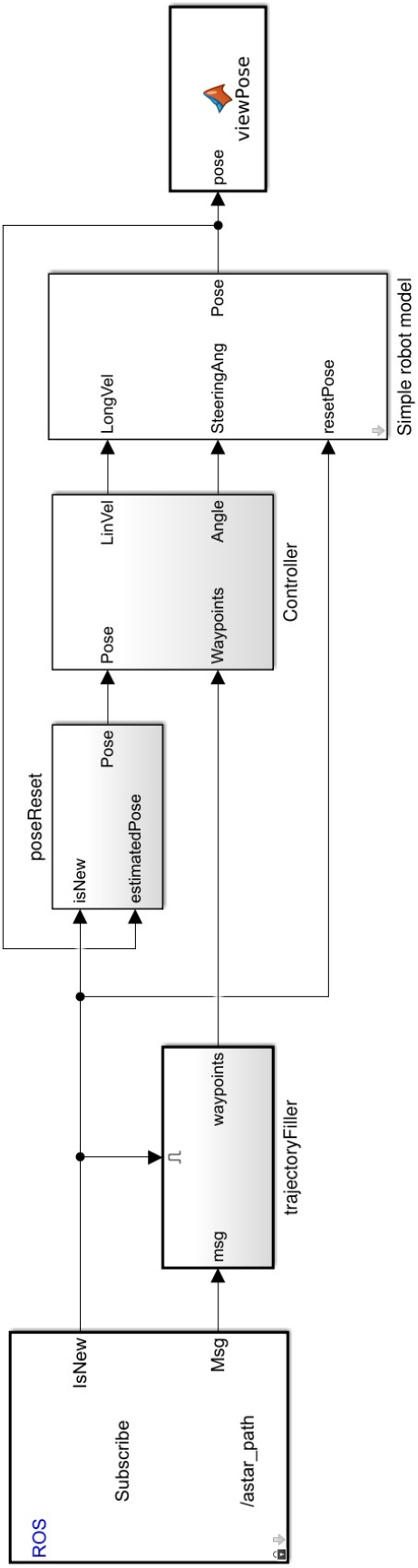


Figure 14: Simulink model developed for the control of a simple robot

3.1 Controller model in Simulink

The Simulink model in figure 14 has been designed and implemented. The main blocks that compose the model are the following:

- **ROS subscriber**
  The subscriber is needed to receive trajectory messages coming from the path planner node. The topic on which new trajectories are published is *astar_path*.
- **ROS message parser** (*Section 3.1.1*)
  This subsystem is able to parse the desired trajectory and to convert it from a ROS message to a series of waypoints (in the subsystem *trajectoryFiller*).
- The **plant** (*Section 3.1.2*)
  This block contains the model of the robot to control.
- The **controller** (*Section 3.1.3*)
  This block reads input signals, executes the pure pursuit algorithm and sends control commands to the robot.

In the following sections plant, controller and trajectory parser will be analyzed in detail. The ROS subscriber concept was already covered in section 1.1.

3.1.1 ROS subscriber node and trajectory parser

To understand how the trajectory is received and converted into a set of waypoints, the relevant Simulink subsystem is presented, describing the content of each block composing it.

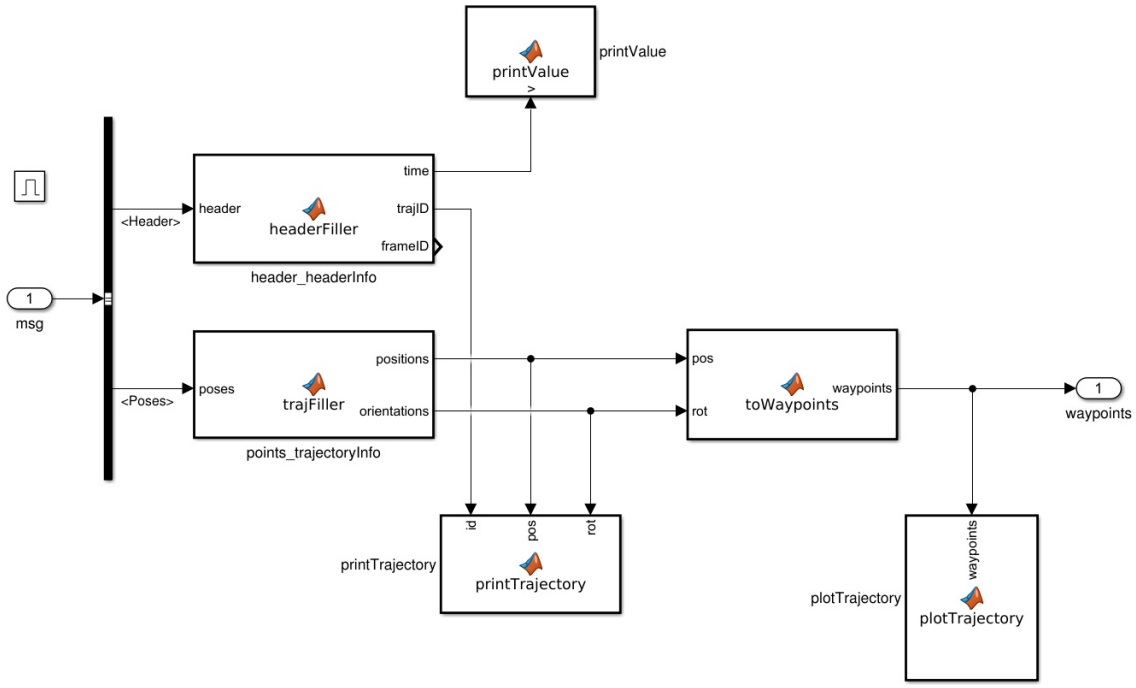


Figure 15: The content of the *trajectoryFiller* subsystem

Looking at the model in figure 15, starting from the left side of the image, the first input port (`msg`) is where the ROS message is received. The signal flows into a **bus selector** that filters the parts of the message that are relevant and forwards them to the block *points_ trajectoryInfo*.

points_trajectoryInfo.m

```
function [positions, orientations] = trajFiller(poses)
   arraySize= 128;

   positions = zeros(arraySize,4);
   orientations = zeros(arraySize,5);
   lastPosX = 1e10;
   j = 1;
   while (poses(j).Pose.Position.X ~= lastPosX)
      % Append 1 as first value in each row of the matrix, indicating
      % that the current row contains valid values
      positions(j,:) = [ 1
                         poses(j).Pose.Position.X
                         poses(j).Pose.Position.Y
                         poses(j).Pose.Position.Z ];

      orientations(j,:) = [ 1
                         poses(j).Pose.Orientation.X
                         poses(j).Pose.Orientation.Y
                         poses(j).Pose.Orientation.Z
                         poses(j).Pose.Orientation.W ];

      % Store the latest valid value read
      lastPosX = poses(j).Pose.Position.X;
      j=j+1;
   end

   % Consider the trajectory null if the first and
   % second value are both equal to zero
   if((j==2)&&(poses(1).Pose.Position.X==0))
      disp('Null trajectory');
      positions = zeros(arraySize,4);
      orientations = zeros(arraySize,5);
   end
end
```

While some information contained in the header (like the timestamp) is printed for debugging purposes, the list of poses contained in the message is forwarded to the block *toWaypoints*, that converts the trajectory from the ROS format to a simpler format stored in a matrix. The waypoints list obtained is then used to plot the pose of the robot during the simulation and is finally sent to the next main block in the model, the controller.

3.1.2 Robot model

In this first validation phase, a differential drive two-wheeled robot is used as plant to control.
This type of robot is characterized by two wheels on the same axis able to rotate with different velocities, in order to change the heading. The robot structure is shown in figure 16.

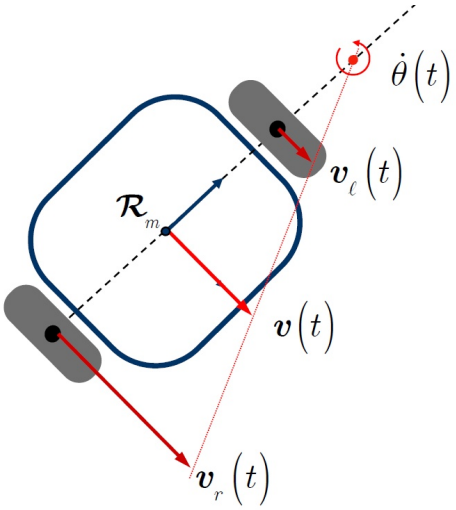


Figure 16: Scheme of a simple differential drive robot

The equation describing the robot kinematics is the following:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega \tag{1}$$

where ω is the angular speed around the center of rotation (ICC, the red dot in figure 16).

After the exact integration of (1):

$$\begin{aligned} x_{k+1} &= x_k + \frac{v_k}{\omega_k}(\sin\theta_{k+1} - \sin\theta_k) \\ y_{k+1} &= y_k + \frac{v_k}{\omega_k}(\cos\theta_{k+1} - \cos\theta_k) \\ \theta_{k+1} &= \theta_k + \omega_k T_s \end{aligned} \tag{2}$$

3.1.3 Pure pursuit controller

The algorithm used to control the robot is the implementation of the pure pursuit method.

The name *pure pursuit* derives from the main feature of this tracking algorithm, inspired by the behavior of human drivers: a person driving a car has the tendency to focus on a virtual point located a certain amount of distance in front of the vehicle and to drive towards it. The distance that separates the vehicle from this target point is called **lookahead distance**. While in a human driver this value is variable and

changes depending on different factors (road shape, presence of obstacles, visibility, current speed) in the basic version of this algorithm it is a fixed parameter.

The working principle behind the pure pursuit algorithm is that every time step the robot identifies the nearest trajectory point, find the one along the path with distance equal to the lookahead distance and points towards it.

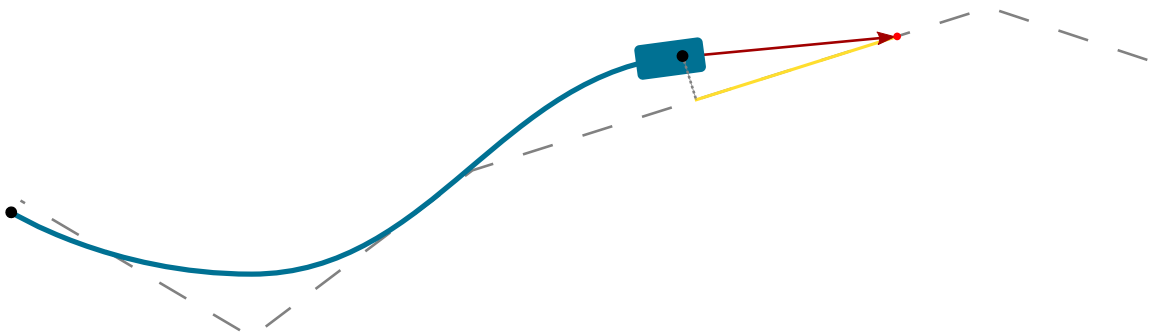


Figure 17: The robot is represented as a light blue rectangle, the lookahead distance measured along the trajectory is represented in yellow

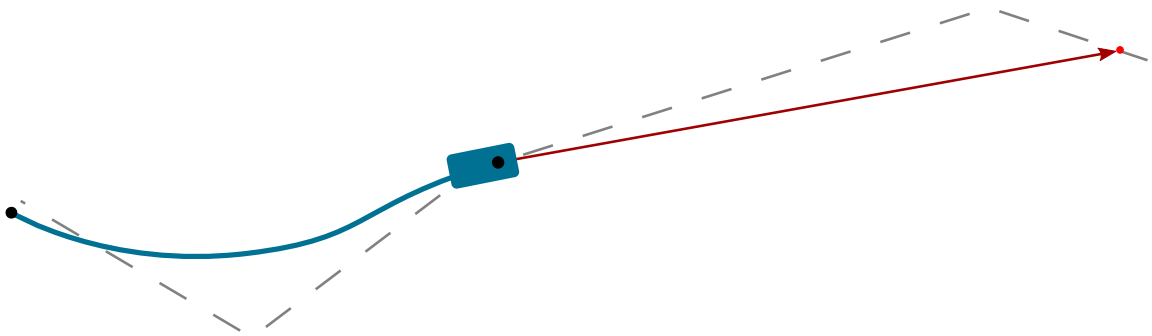


Figure 18: Behavior with a **large** lookahead distance value

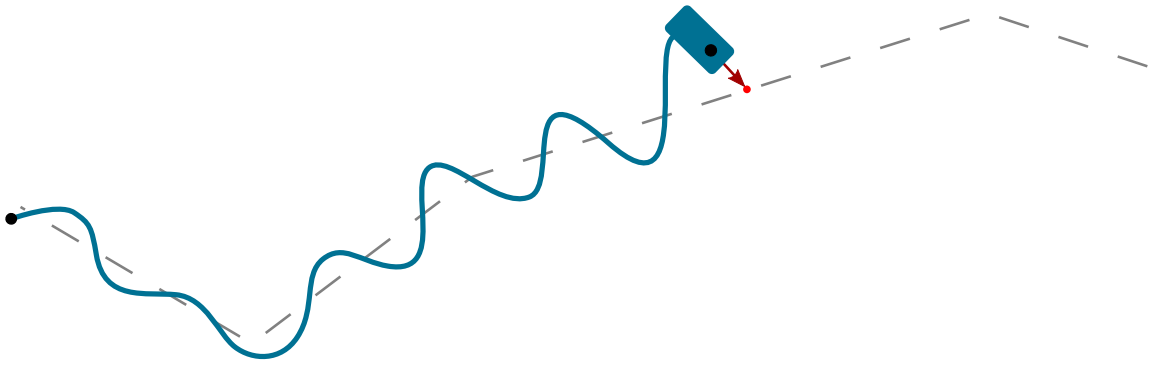


Figure 19: Behavior with a **small** lookahead distance value

The variation of the lookahead distance parameter clearly affects the resulting path, as figures 18 and 19 clearly show. The two main observable consequences are differences in **path regaining** and **path keeping**.

A large lookahead distance will result in less oscillations but in a final path not really close to the desired one (good path regaining, bad path keeping); on the other hand a small value will make the vehicle follow the trajectory more accurately but it will result in an unstable and oscillatory behavior (good path keeping, bad path regaining).
It's clear how the tuning of the lookahead distance is important. A priori knowledge on the possible trajectories generated by the path planner is useful to choose the best value for this parameter.

The lookahead distance is used in the algorithm to draw an arc that connects the current position with the target position. The chord of the arc will coincide with the lookahead value. Due to this reason, the trajectory generated by a pure pursuit controller can be always represented as a series of arc segments.

3.2 Infrastructure validation

Multiple series of tests are performed under different conditions to validate the architecture and to check if ROS messages are parsed and interpreted correctly.

The two main results to verify in this phase are:

1) The controller is working as expected and is able to control the simple robot using a trajectory provided manually.

2) The whole control system is able to receive and parse a trajectory coming from the other ROS modules and to control the robot using that information.

3.2.1 Step 1: Controller validation

In the first test a single hardcoded trajectory is used as input in the Simulink model. The ROS parser is temporarily overridden and replaced with a trajectory provider not depending on any ROS message.

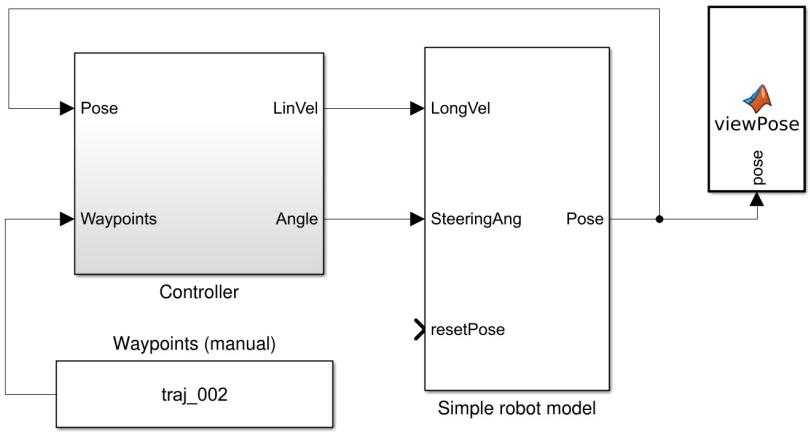


Figure 20: Simulink model that allows a manual injection of the trajectory to follow

After the target path to follow (figure 21) is provided to the model, the controller is executed. The path followed by the simulated robot is then plotted through the block *viewPose* and can be seen in figure 22. The obtained results visible in the figures validate this first implementation step, confirming the correct functioning of the system.

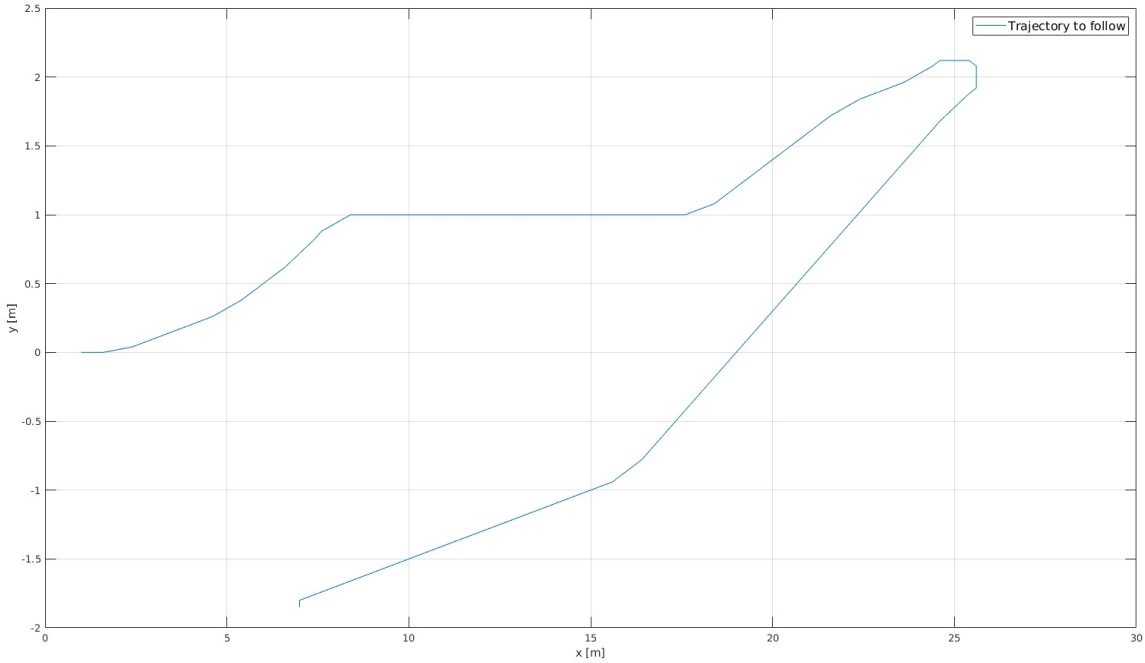


Figure 21: Input trajectory created from arbitrary numerical data and smoothed using the `smooth` command in MATLAB

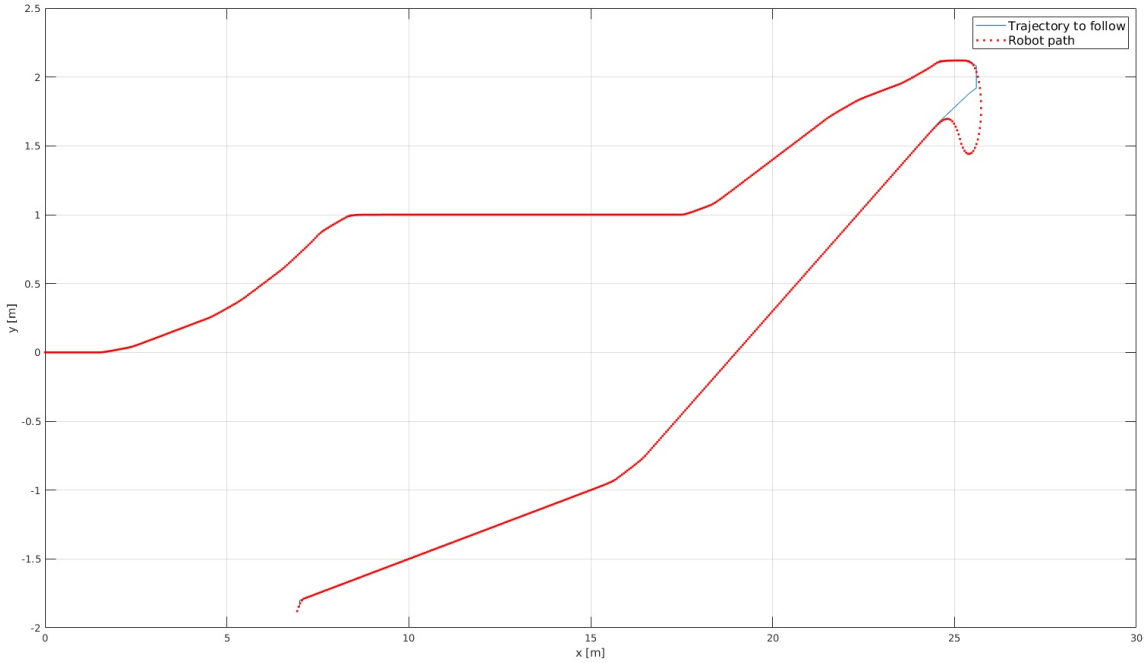


Figure 22: Output trajectory. The blue line is showing the provided path to follow, the red points compose the trajectory followed by the robot during the execution of the algorithm. In this example the only visible issue is that in the upper right corner the vehicle is not able to follow the path, due to the sudden variation of the target trajectory. Other tests of this scenario at the variation of the lookahead distance values are available in **Appendix A** (figures LA1 and LA2)

3.2.2 Step 2: ROS communication layer integration and validation

The second step is the validation of the architecture in a realistic use case, with the ROS path planner module enabled, to check if the transmission and the parsing of the target path is working properly. To perform this validation, rviz is executed together with all the ROS nodes that perform input handling, costmap generation, path estimation and trajectory publication as previously described.

In rviz starting and target poses are chosen via user input while the Simulink model is running. The output of the path planning algorithm is also visualized on the costmap.

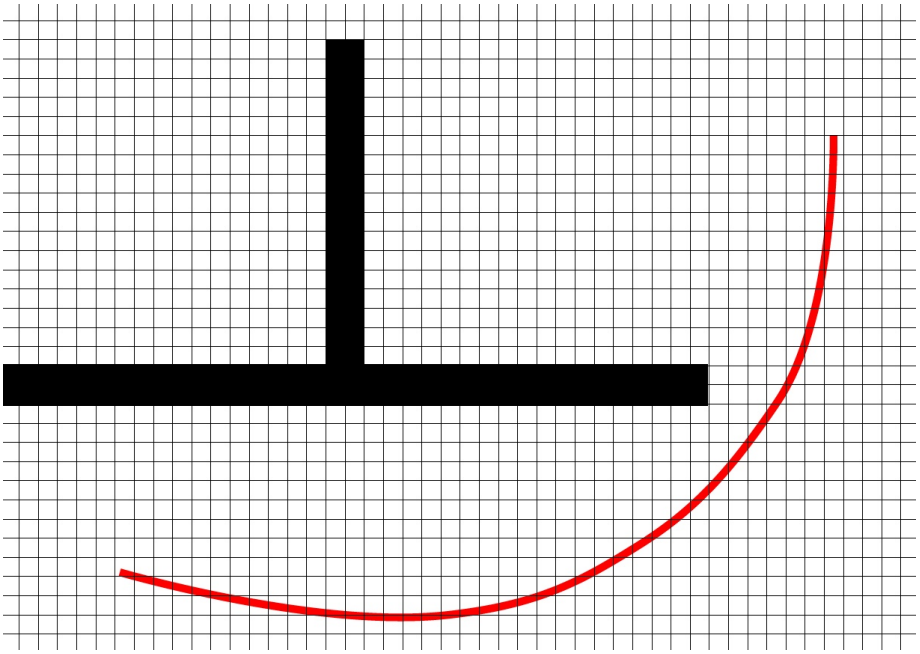


Figure 23: The trajectory computed from the A* algorithm is visualized in rviz as shown in the figure. The same trajectory is published to the relevant ROS topic

After the trajectory is computed and published on the */astar_path* topic, a block in the Simulink model (*plotTrajectory*) is triggered and plots the current pose on a graph, as shown in figure 24.

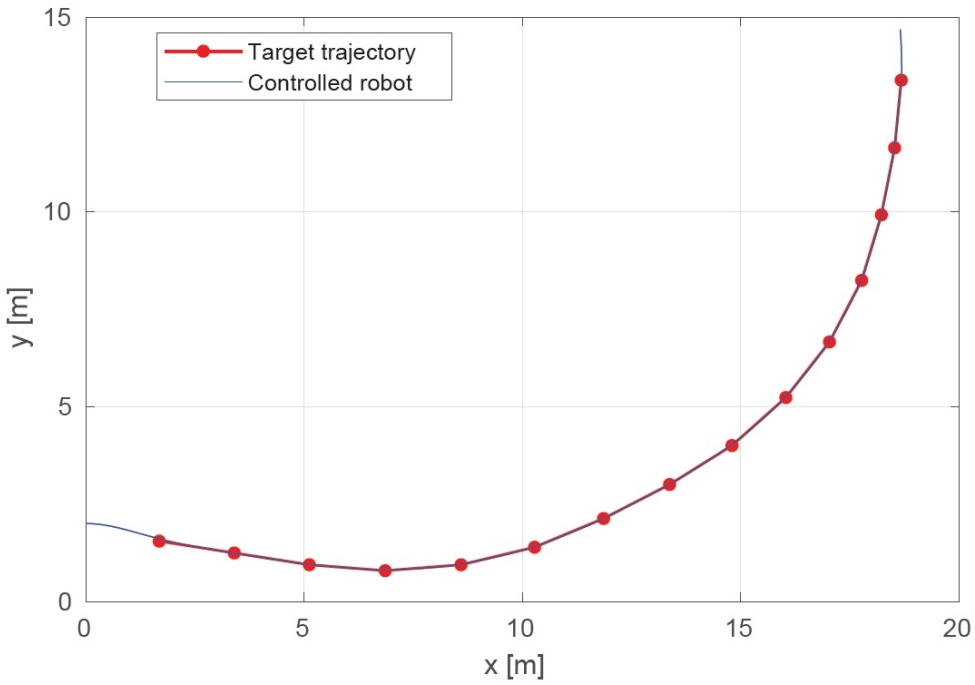


Figure 24: The result of the test validates the infrastructure: the trajectory calculated from the ROS path planner (red line) is correctly parsed and used to control the robot model in Simulink (blue line). The tracking is really accurate, in part due to the smoothness of the target trajectory, that doesn't require hard maneuvers to be followed

With both the validation steps completed, the obtained results demonstrate that the architecture developed for Simulink is working as expected and that all the blocks that compose the designed motion planner are communicating correctly.

The goal of this phase is reached: given a costmap and a target pose the designed infrastructure is able to compute a trajectory to follow, correctly communicate it to the Simulink model and drive the robot along the desired path, thanks to the parser and the controller subsystems.

4 Simulation in CarMaker

In this project CarMaker is used to simulate the vehicle on which the whole infrastructure is validated through the design, implementation and testing of the motion planner.

The main advantage in using a simulation software is the possibility to visualize in a 3D environment the behavior of the virtual vehicle, making easy to perform a quick evaluation of the simulation outcome, without the need to preprocess and analyze the recorded data if a clearly wrong behavior is observed. For example, the variation of the lookahead distance has a direct effect in the simulation that can be immediately observed and noted.

4.1 CarMaker GUI

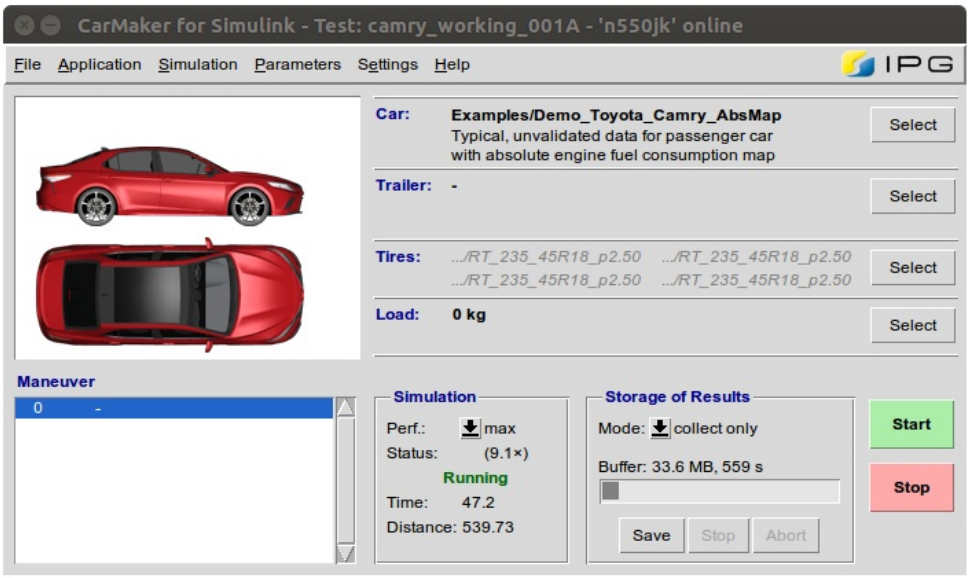


Figure 25: CarMaker graphical interface

The CarMaker GUI (figure 25) is the main interface that allows the user to create scenarios, load test cases, perform simulations, log data and change the vehicle parameters. The tuning possibilities are really broad considering that using the configuration tool (figure 26) is possible to change a large amount of physical properties and characteristics of the simulated car:

- Geometrical information of body and wheels
- Car loads and trailer
- Tires
- Suspension type, elastic properties, dampers, wheel bearings
- Engine mounting
- Steering model
- Brake model

– Powertrain model
– Aerodynamics
– Sensors
– Vehicle control

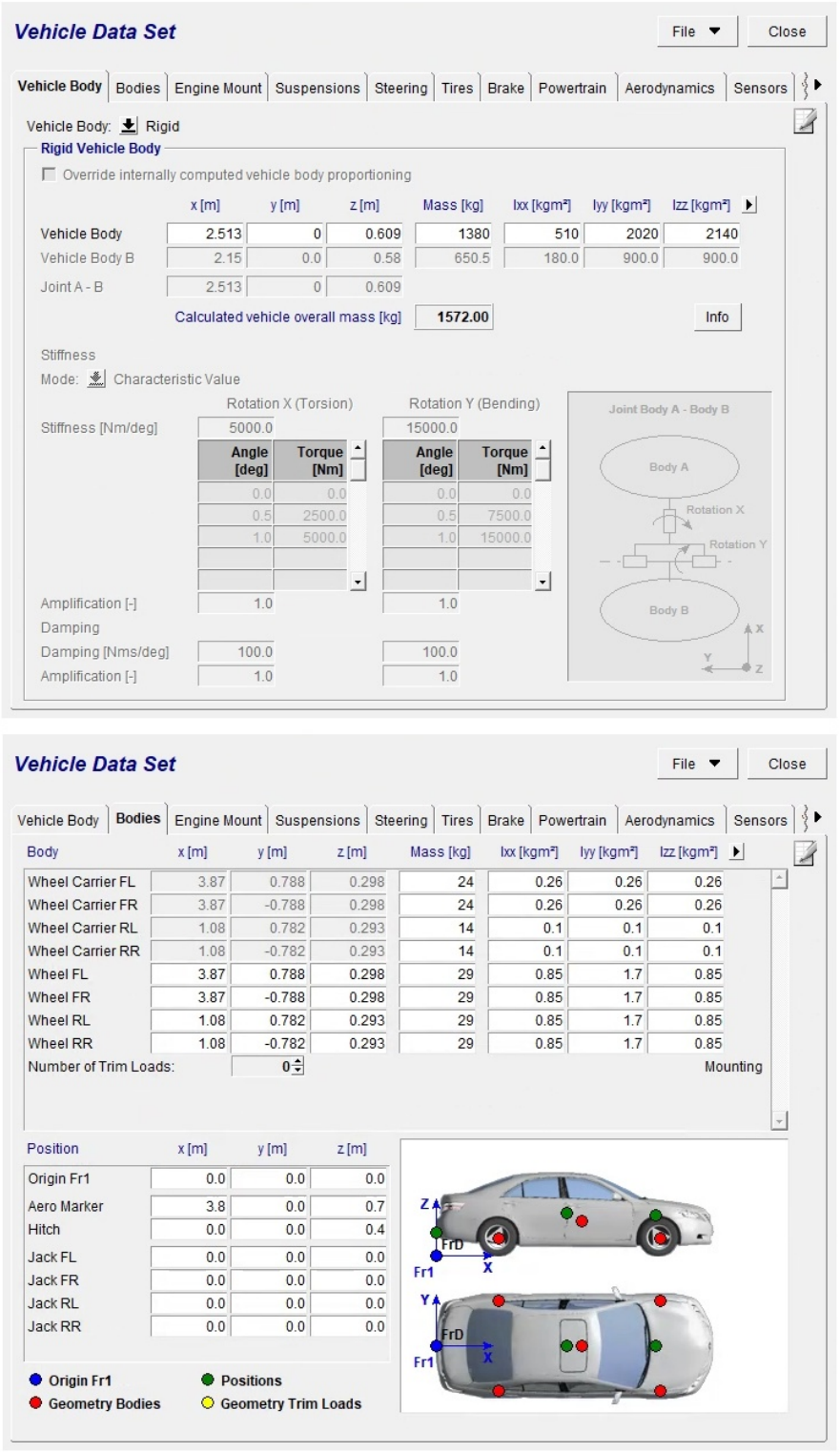


Figure 26: Vehicle data configuration windows

Moreover, the environment can be customized in multiple way, ranging from changes in the road parameters to the weather conditions, making easy to recreate a variety of different scenarios.

Among the mentioned tunable parameters, the ones relevant for this project are geometrical information, steering model, powertrain model and vehicle control.

4.2 Vehicle settings basic setup

To generate an accurate model of the vehicle some parameters (size of the car, weight, wheel geometry) are specified in the configuration window. In the development of this framework the vehicle considered as an ideal target for the final deployment is a **BMW i3**. Measurements are obtained and inserted in CarMaker.
Table 1 lists the values retrieved from technical datasheets.

Measurement	**Value**
Weight	1297kg
Wheelbase	2,57m
Track/tread (front)	1,571m
Track/tread (rear)	1,576m
Length	4,008m
Width	1,775m
Height	1,578m
Ground clearance	0,14m
Wheel size (front)	5Jx19
Wheel size (rear)	5Jx19

Table 1: BMW i3 specifications

4.3 Structure of a vehicle in CarMaker for Simulink

Thanks to the integration of CarMaker in Simulink, the virtual vehicle is generated and represented as a series of **connected subsystems**. At the highest level of the model the division in blocks is the one shown in figure 27.

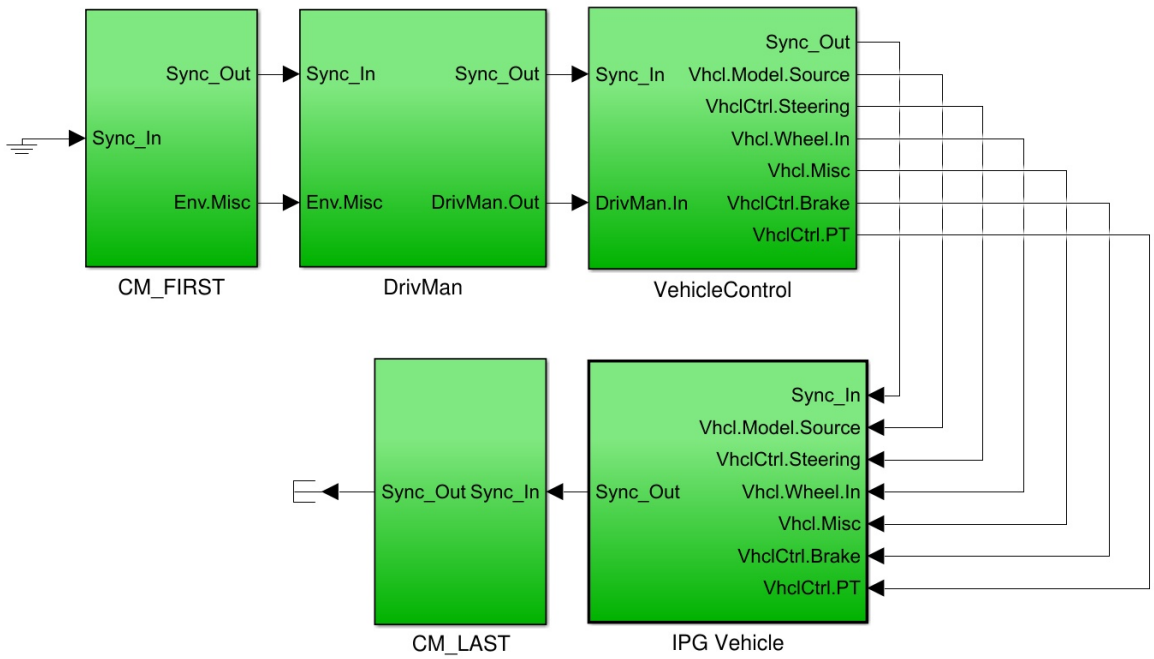


Figure 27: Model of a vehicle generated with CarMaker

The most relevant subsystem for the implementation of a motion controller is the one handling and transmitting input commands: VehicleControl (or *VhclCtrl*), inside which multiple actuation signals are received and propagated (figure 28).

Inside *VhclCtrl* input signals are received from **virtual drivers** (*DrivMan*) and are propagated in the model to finally reach the actuators of the simulated vehicle (*IPG Vehicle*).

From the point of view of a CarMaker user that wants to perform a simulation without implementing new control systems, everything is ready to be used and doesn't require any modifications of the Simulink model. Through the CarMaker graphical interface, the user can set up a virtual driver that acts on the steering wheel following a defined rule (for example, applying a sinusoidal wave signal) and that maintains a target speed.

In the context of this work the mentioned signals need to be overridden to implement a new control system. To achieve this, some of the connections shown in figure 28 are cut and rerouted, as will be explained in detail in section 5.1.

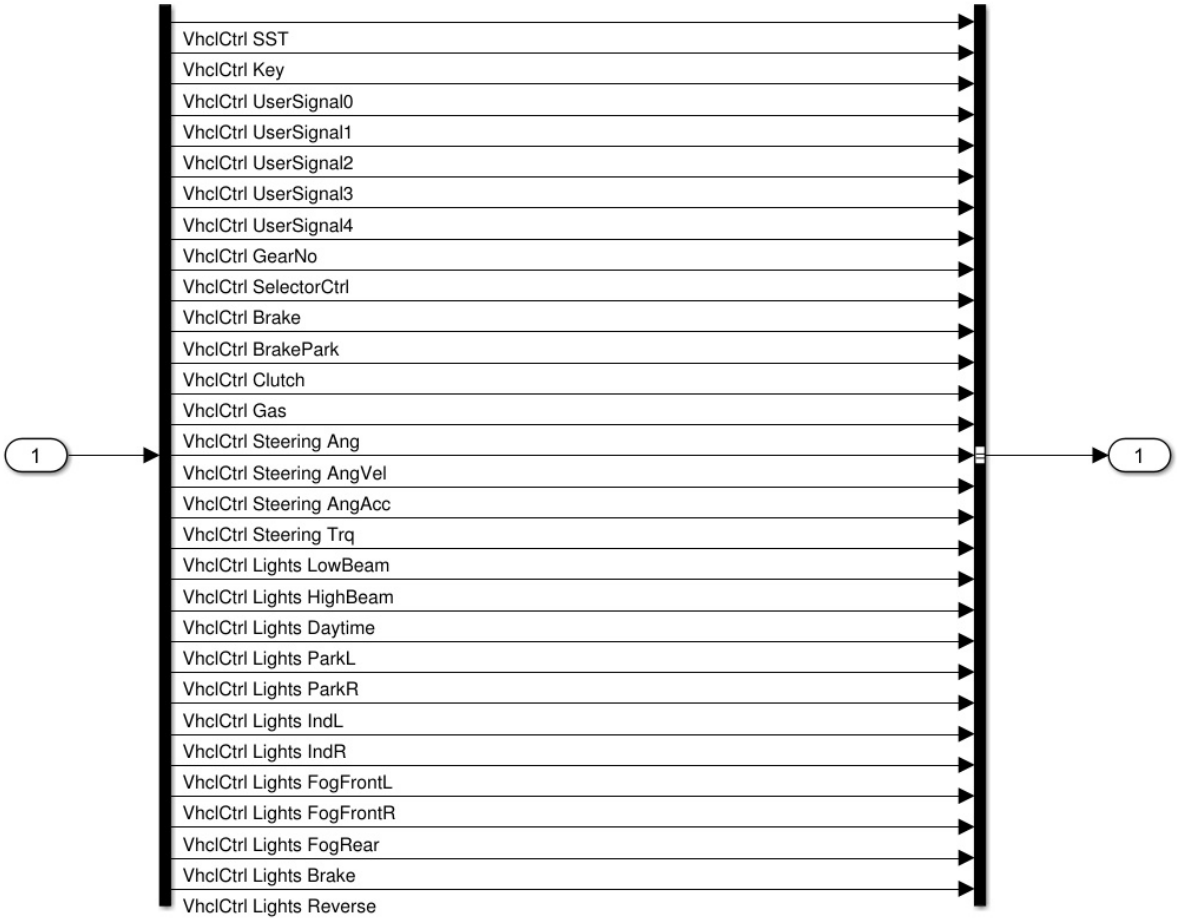


Figure 28: *VhclCtrl* subsystem

4.4 Simulation data storage

A possible way to record and store the data generated during simulations is the **output quantities** menu in CarMaker. Through a graphical interface (figure 29) output values to record and store can be selected.

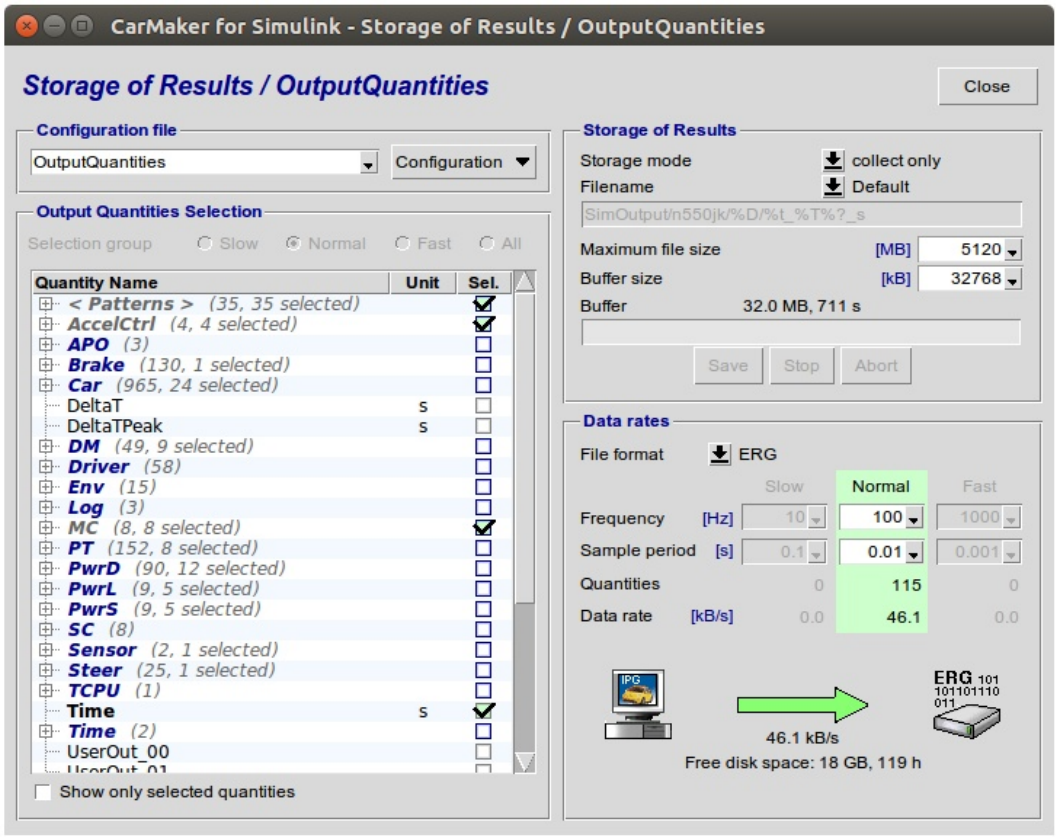


Figure 29: CarMaker output quantities configuration window

To study the behavior of the vehicle, the quantities listed in table 2 are selected to be saved during the testing.

State variable	Direction	CarMaker variable	Units
Timestamp	-	*Time*	[s]
Position	x	*Vhcl.Fr1.x*	[m]
	y	*Vhcl.Fr1.y*	[m]
	z	*Vhcl.Fr1.z*	[m]
Speed	$\dot{x}$	*Vhcl.vx*	[m/s]
	$\dot{y}$	*Vhcl.vy*	[m/s]
	$\dot{z}$	*Vhcl.vz*	[m/s]
Angular position	ϕ	*Vhcl.Roll*	[rad]
	θ	*Vhcl.Pitch*	[rad]
	ψ	*Vhcl.Yaw*	[rad]
Angular speed	$\dot{\phi}$	*Vhcl.RollVel*	[rad/s]
	$\dot{\theta}$	*Vhcl.PitchVel*	[rad/s]
	$\dot{\psi}$	*Vhcl.YawRate*	[rad/s]

Table 2: Variables saved during a simulation

After a simulation is performed, a simple MATLAB script is used to load the recorded data, extracting the variables that can be used for analysis and visualization afterwards. As an alternative, it's also possible to collect data directly from the Simulink model using *To Workspace* blocks. In this project both solutions are used.

4.5 CarMaker dictionary variables

In control theory there are two basic types of control systems: **open loop** controllers and **closed loop** (or feedback) controllers. While controllers of the first type don't need any information on the state of the system to work properly, closed loop controllers need to measure the process variable and compute its error with respect to the desired value in order to run the control algorithm.
In this project the motion planner designed implements a closed loop control system, therefore it needs to know the pose of the vehicle every time step. To get this information Simulink blocks called *Read CarMaker Dictionary Variable* are used. These blocks available in the CarMaker library ensure that the information retrieved is synchronized with the whole simulation through the usage of special blocks called *Sync_In* and *Sync_Out*, placed respectively at the beginning and at the end of the generated model (see figure 27).

5 Controller design

5.1 Model of the vehicle

The vehicle to control is simplified using the **bicycle robot model**.
This approximation is often applied to four wheeled vehicles as it fits well the behavior of a standard car with differential gear if the lateral acceleration is limited, as it happens in standard conditions.[14,15]

The simplified bicycle model is composed of two wheels linked through a fixed rigid body of arbitrary length L. Only the front wheel is able to perform steering, rotating around its center.

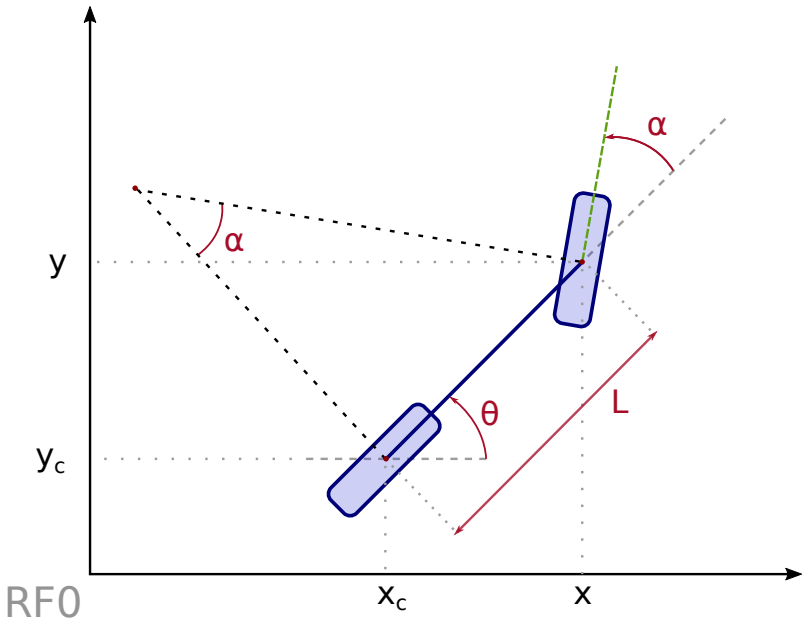


Figure 30: Bicycle model

L: **baseline**, distance between the wheels

α: **steering angle** of the steering wheel

θ: **orientation** with respect to the world reference frame (**RF0**)

(x_c, y_c): **position** of the non-steering (rear) wheel

(x, y): **position** of the steering (front) wheel

Given these information, the state of the system can be described as:

$$\mathbf{q} = [x \;\; y \;\; \theta \;\; \alpha]^T. \tag{3}$$

For each wheel a constraint is defined to prevent slipping (wheels cannot move along the lateral direction):

$$\dot{x_c}\sin(\theta+\alpha) - \dot{y_c}\cos(\theta+\alpha) = 0 \tag{4}$$

$$\dot{x}\sin\theta - \dot{y}\cos\theta = 0 \tag{5}$$

An additional constraint represents the rigid link between the wheels:

$$\begin{aligned} x_c &= x + L\cos\theta \\ y_c &= y + L\sin\theta \end{aligned} \tag{6}$$

Using (6), equation (4) can be rewritten as:

$$\dot{x}\sin(\theta+\alpha) - \dot{y}\cos(\theta+\alpha) + L\dot{\theta}\cos\alpha = 0 \tag{7}$$

Considering that the bicycle model is approximating a car, the power is assumed to arrive from the non-steering wheel (rear wheel drive).

The equation describing the state of the system is then:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\alpha} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{1}{L}\tan\alpha \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{8}$$

where v is the linear velocity and ω is the rotational speed of the steering wheel.

Equation (8) can also be written as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\alpha} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ \frac{1}{L}\tan\alpha & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{9}$$

The first problem to solve is how to control the bicycle model without acting directly on the linear or angular velocity. A CarMaker simulated vehicle accepts as input the same commands that a real driver would apply: pedal pressure and steering wheel actuation.

Due to the mentioned reasons, the next step is the design and implementation of controllers able to act on steering wheel, gas pedal and brake pedal, to track the desired linear and angular velocity.

5.2 Actuation controllers design

Before starting to approach the actuation problem, it's important to observe how the simulated vehicle gearbox is configured. Considering that the ideal target vehicle is a BMW i3, which uses an automatic transmission system, a standard automatic gearbox is selected and used in CarMaker to perform the validation.
The choice of a generic gearbox is the consequence of a simplification: the level of precision in the CarMaker configuration tool would allow to recreate exactly the same transmission system of the BMW i3, but specifications of the mechanical structure would be necessary. However, this is something that can be done as a further refinement of the infrastructure after the foundations are working properly, hence is not performed in this thesis project.

The whole controller infrastructure is deployed in the CarMaker *VhclCtrl* subsystem (section 4.3). In the next sections the controllers implemented for pedals and steering wheel are presented.

5.2.1 Speed control

The two signals *VhclCtrl.Gas* and *VhclCtrl.Brake* are disconnected and rerouted to be used as command inputs for the controllers.

A pair of P controllers with limited output are used for the actuation of gas and brake pedals. Parameter estimation is performed through classic approaches,[16,17] evaluation of the obtained velocity and tuning tools available in MATLAB.
As feedback, the current speed value is used, read directly from the signal *Vhcl.v*, using a CarMaker dictionary variable block (section 4.5).

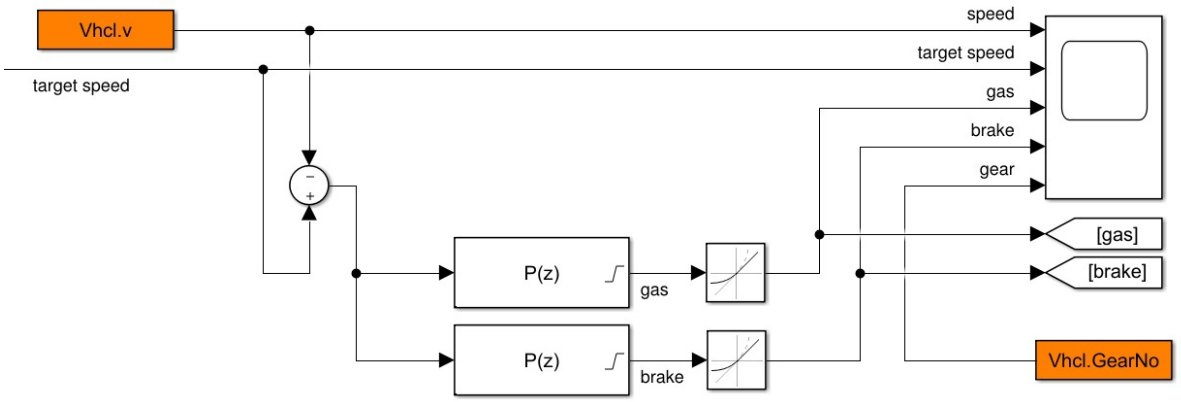


Figure 31: Controller designed for brake and gas pedals actuation

The developed model is shown in figure 31. Let's analyze in detail how the signals are propagated. The target velocity representing the signal to track is on the left side of the picture, labeled *target speed*. Just above it, the CarMaker dictionary variable used as feedback can be seen. The first round block is a simple subtractor, used to compute the difference between the desired and the current value, following the classic architecture of negative feedback controllers.

The speed error computed by the subtractor is then used as input for two limited proportional controllers, one for the gas pedal and one for the brake pedal. The blocks visible right after the controllers are slew rate limitators, used to avoid abrupt changes in the commands.
On the right side of the model a scope is placed to analyze the signals during the execution of the system. The input ports labeled *gas* and *brake* are conveying the command signals directly to the *VhclCtrl* subsystem.

Note that the dictionary variable *Vhcl.GearNo* (current gear) is being fed into the scope. When a upshifting or downshifting occurs there is a small spike in the speed of the car, due to the acceleration/deceleration effects in the transmission. This spike is so sudden that the controller is not able to react properly, but it can still be observed and analyzed in the obtained graphs. In the future it would be a good improvement the implementation of an advanced controller able to use information on the rotational speed of the motor to predict the gear shifting, adapting the commands accordingly and avoiding spikes in the resulting speed.

5.2.2 Steering control

Instead of using a P controller for the actuation of the steering wheel a PI controller is selected. The reason behind this choice is to compensate for an offset observed between the desired yaw rate value and the obtained one. In this model the variable to track is the yaw rate, controlled using the steering wheel angle as input.
The structure of the designed model is really similar to the one handling the speed control previously described.

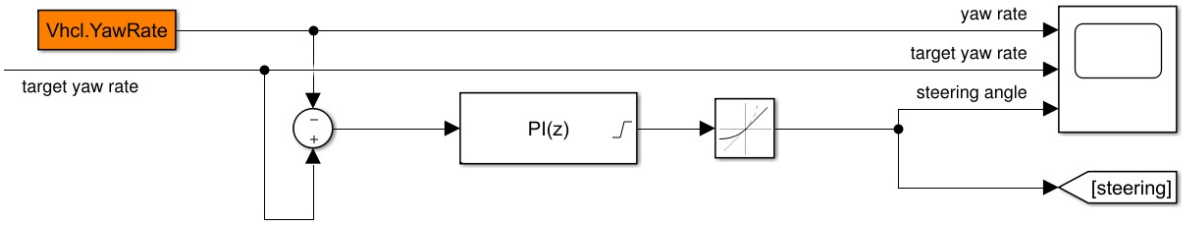


Figure 32: Controller designed for steering wheel actuation

5.3 Actuation controllers validation

To perform the validation of the developed controllers, step signals and sinusoidal waveforms are used as target values for yaw rate and linear speed of the vehicle.
To check the response to both types of signals in a single test a Simulink block able to generate mixed customizable signals is created and used. The test architecture is shown in figure 33 while the block used for validation and testing is visible in figure 34.

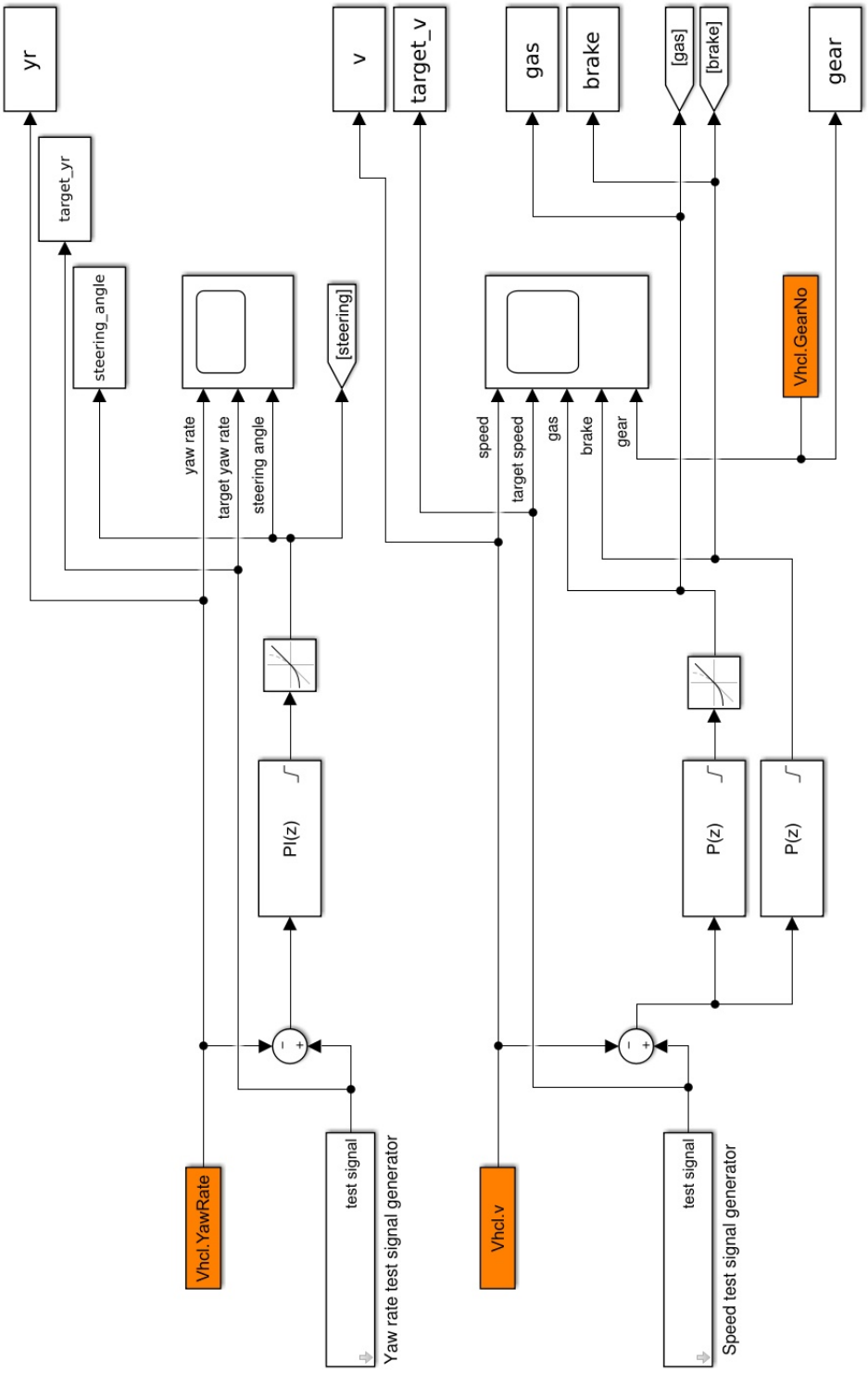


Figure 33: Model used for the validation of the actuation controllers

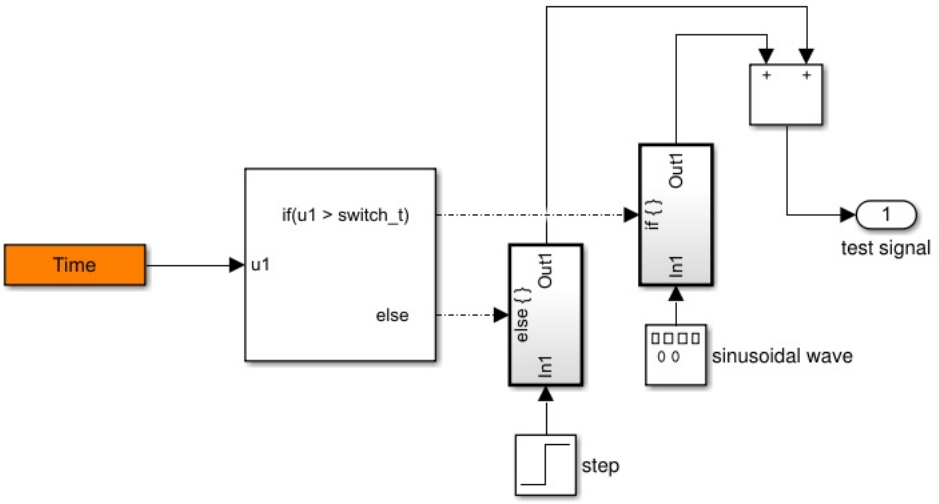


Figure 34: System used to generate step signals and sine waves to check the response of the tested controller. This subsystem switches the generated signal from a step signal to a sinusoidal waveform at the time specified by the user through a graphical interface

Speed controller validation

To validate the speed controller isolating it from the rest of the system, the target yaw rate is fixed at $0 rad/s$. The test subsystem is used to perform multiple test drives at the variation of the signal to track.

Tests are performed using the combinations of parameters listed in table 3.

Amplitude $[m/s]$	**Frequency** $[Hz]$	**Figure**
2	0.1	35
2	0.4	SP1*
6	0.2	36
14	0.05	SP2*

(* results in **Appendix A**)

Table 3: Parameters used in yaw rate controller validation

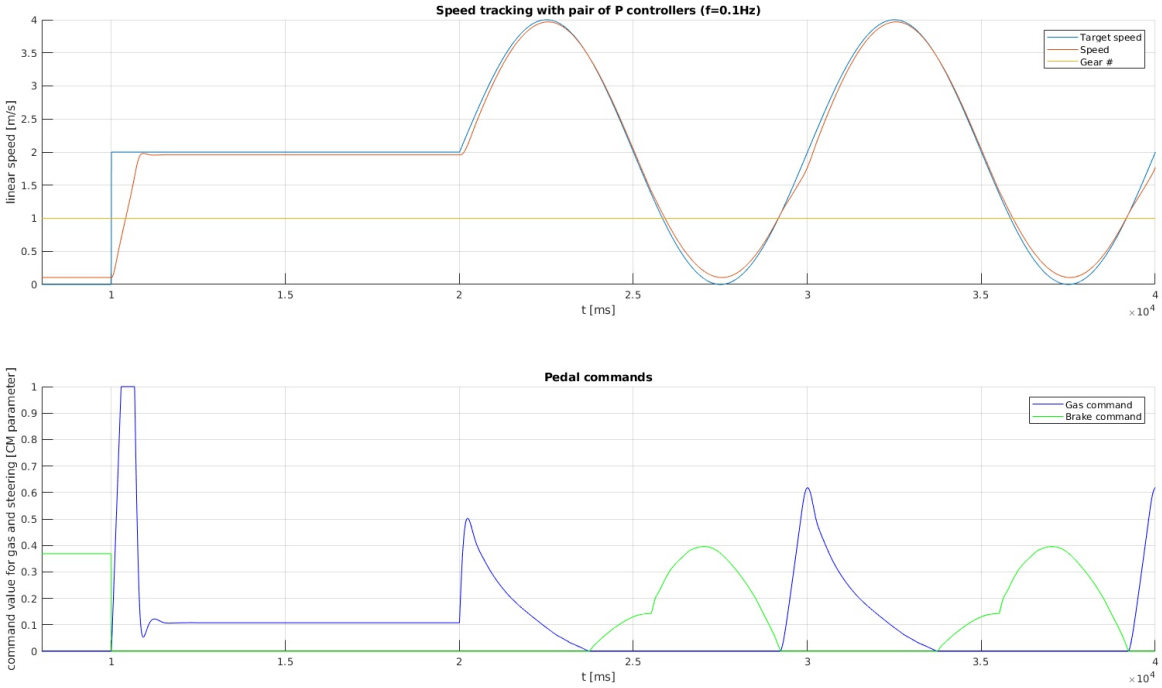


Figure 35: Speed tracking result ($f = 0.1Hz$, $v_{target} = 2m/s$)

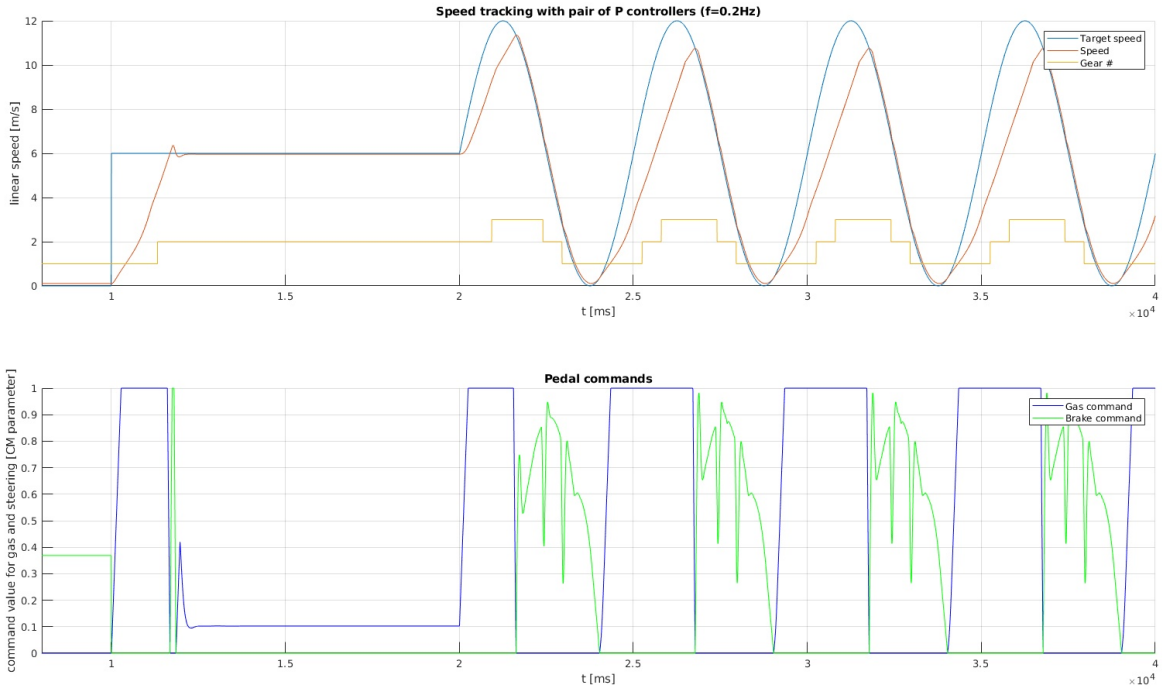


Figure 36: Speed tracking result ($f = 0.2Hz$, $v_{target} = 6m/s$)

Yaw rate controller validation

To validate the yaw rate controller isolating it from the rest of the system, the target speed is set and fixed at $3m/s$.

Tests are performed using the combinations of parameters listed in table 4.

Amplitude $[rad/s]$	**Frequency** $[Hz]$	**Figure**
0.1	0.1	37
0.2	0.1	YR1*
0.1	0.4	38
0.2	0.4	YR2*

(* results in **Appendix A**)

Table 4: Parameters used in yaw rate controller validation

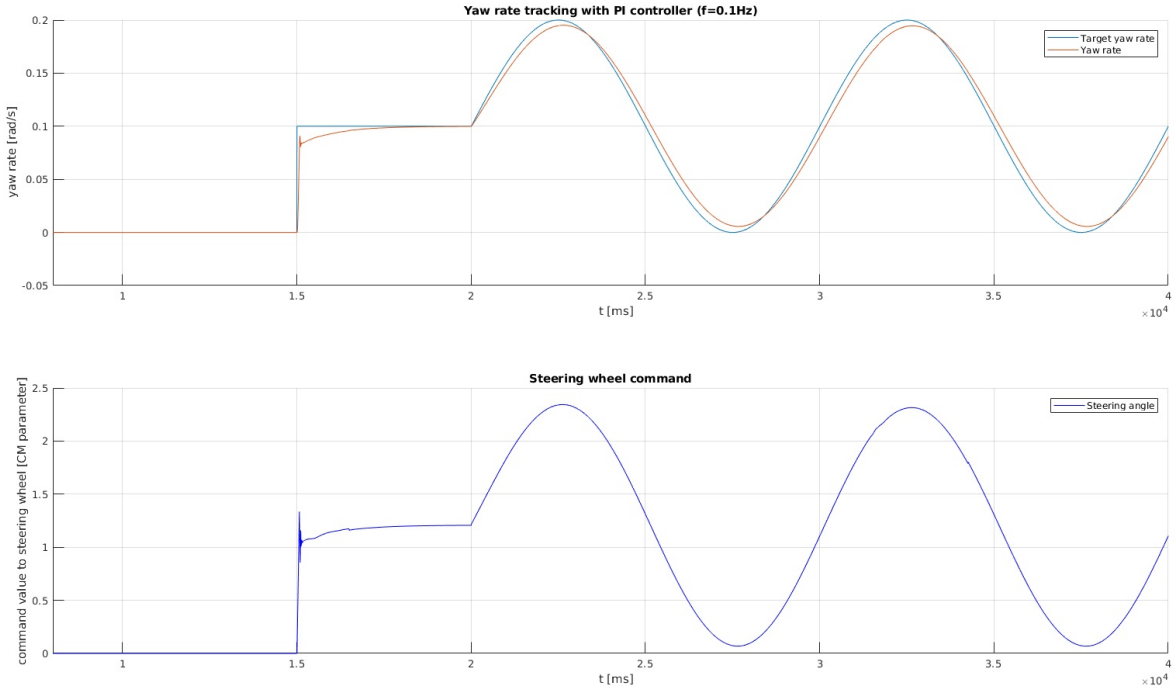


Figure 37: Yaw rate tracking result ($f = 0.1Hz$, $yr_{target} = 0.1rad/s$)

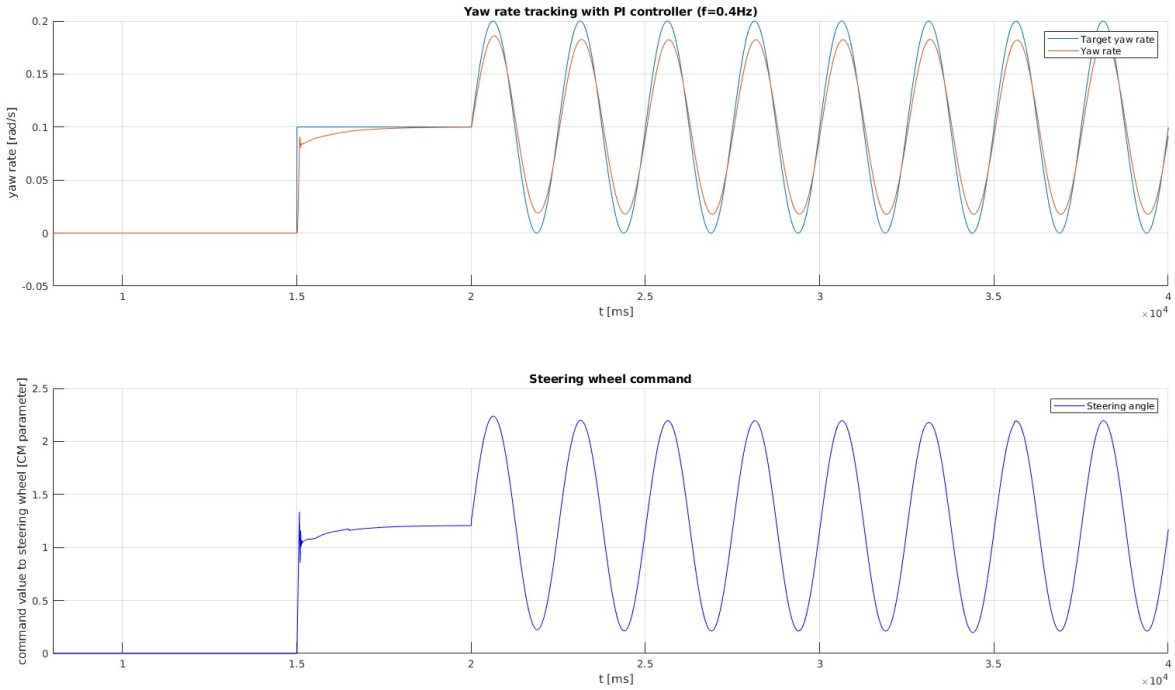


Figure 38: Yaw rate tracking result ($f = 0.4Hz$, $yr_{target} = 0.1rad/s$)

5.3.1 Final CarMaker model

Coupling the previously tested controllers the final infrastructure able to track linear speed and yaw rate is obtained.
The whole model is shown in figure 39.

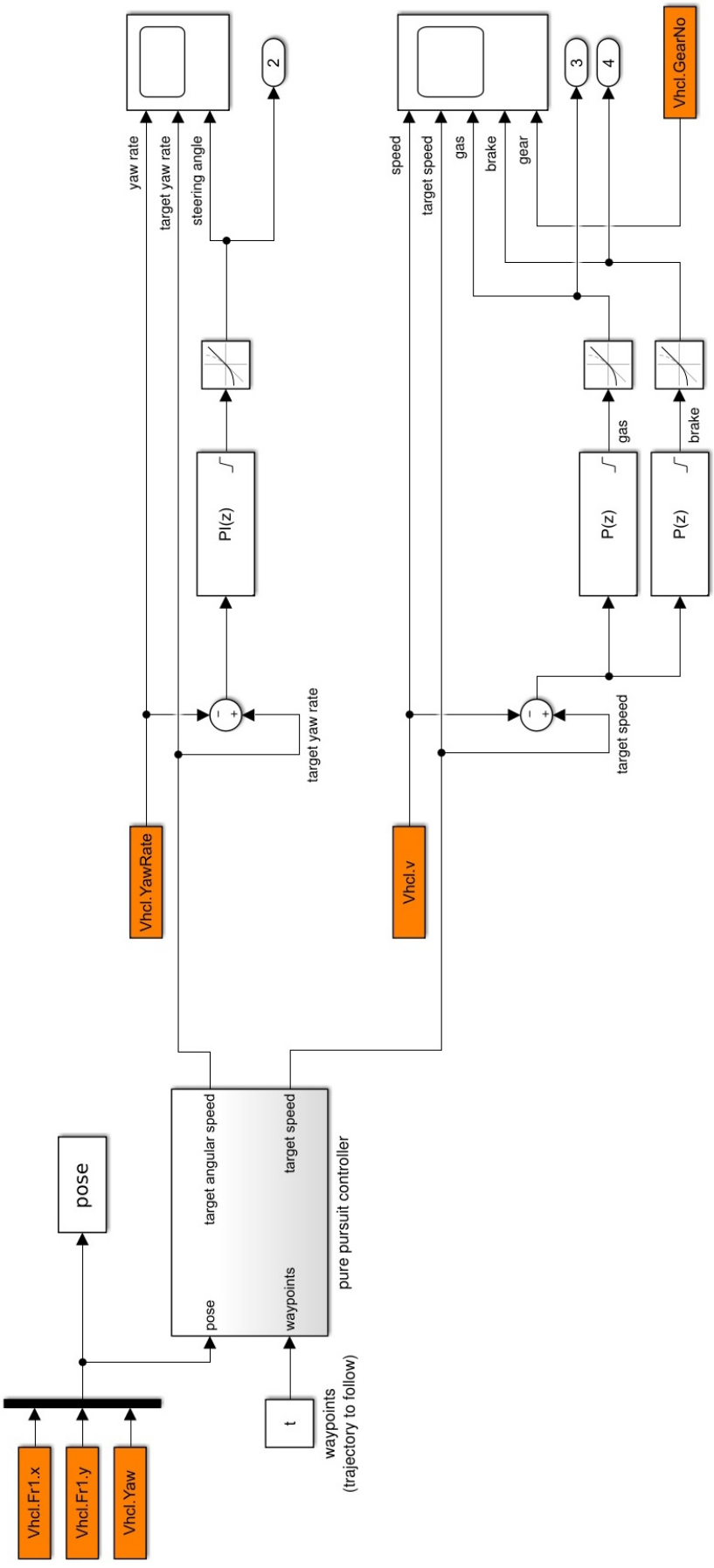


Figure 39: Entire model of the developed motion planner

5.4 Application of pure pursuit controller

The pure pursuit controller is used in the final set of tests to validate the whole motion planner and to check if the obtained behavior is promising. The target of this final validation is to check if the simulated CarMaker vehicle is able to follow a trajectory provided through the developed infrastructure.

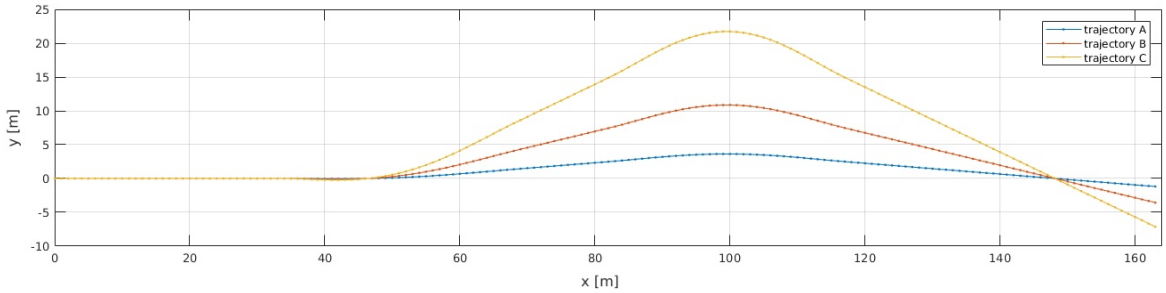


Figure 40: Trajectories used for testing and validation. All paths have the same geometry but different scaling. Y coordinates of path B and C are scaled respectively of a factor 3 and 6 with respect to the ones of path A

The results of performed tests are shown in figure 41 and figure 42. In both figures the target trajectory chosen is **A** (see figure 40).

Figure 41 shows a series of simulations with target velocity set at $3m/s$ ($10.8km/h$) at the variation of the lookahead distance, using the values [$2m$, $3m$, $6m$].

Figure 42 shows a series of simulations with target velocity set at $10m/s$ ($36km/h$) at the variation of the lookahead distance, using the values [$2m$, $6m$, $10m$].

In **Appendix A** (page 50) additional results obtained using different trajectories and parameters are included.

5.4.1 Evaluation

The results obtained show a really good behavior of the whole control system at low speeds but also a significant influence of small variations of the lookahead distance. A good KPI indicating the outcome of the tests is the *cte*, the cross track error. In every plot a box in the upper left corner highlights oscillations in the vehicle path, providing a good visual indicator of the simulation outcome.

Notice how the simulation of trajectory **A** with target speed of $3m/s$ and lookahead distance of $3m$ (figure 41, center plot) is really promising and shows how the implemented motion planner can give really good results.

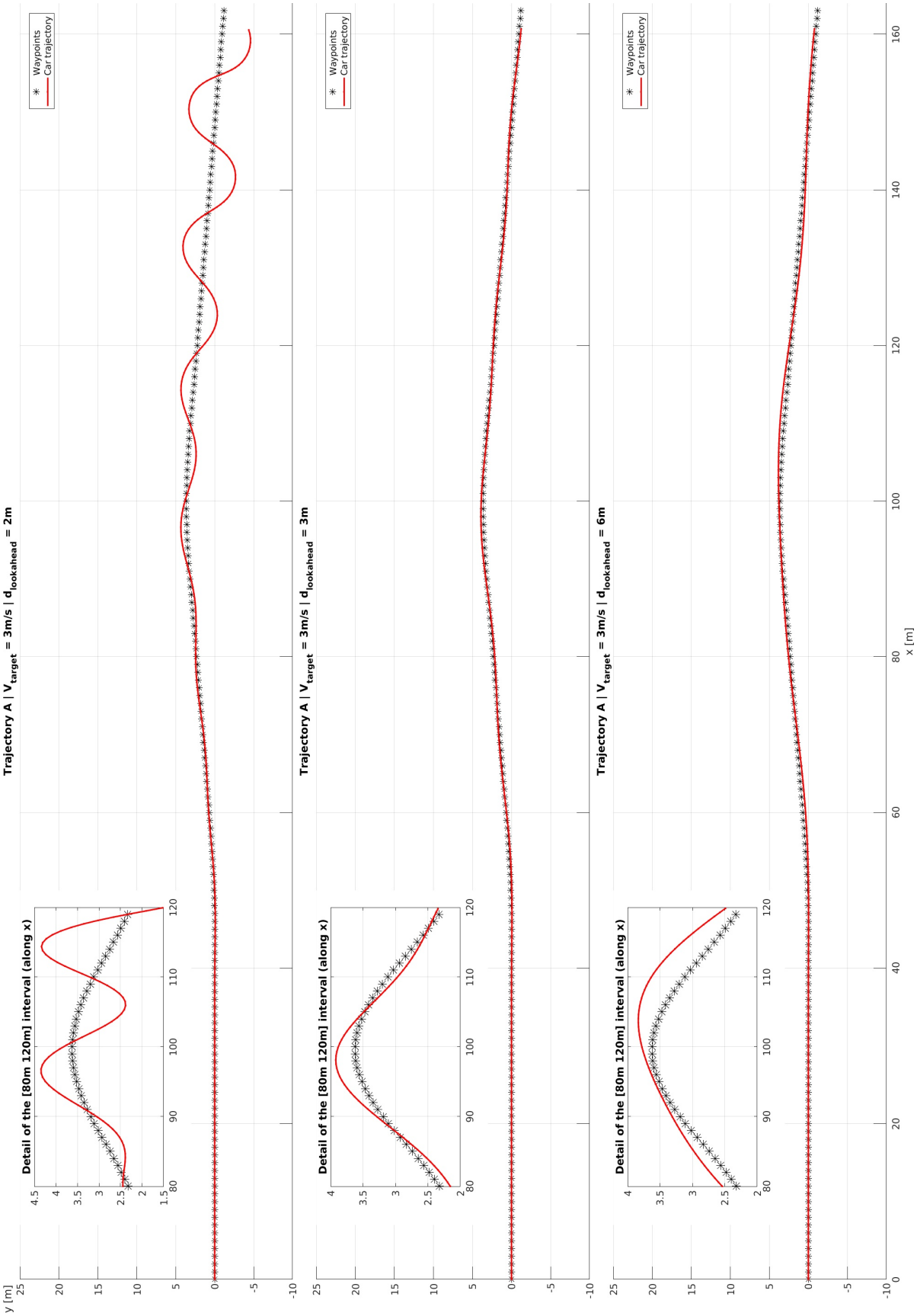


Figure 41: Results of simulation using trajectory A with fixed target speed of $3m/s$ at the variation of the lookahead distance

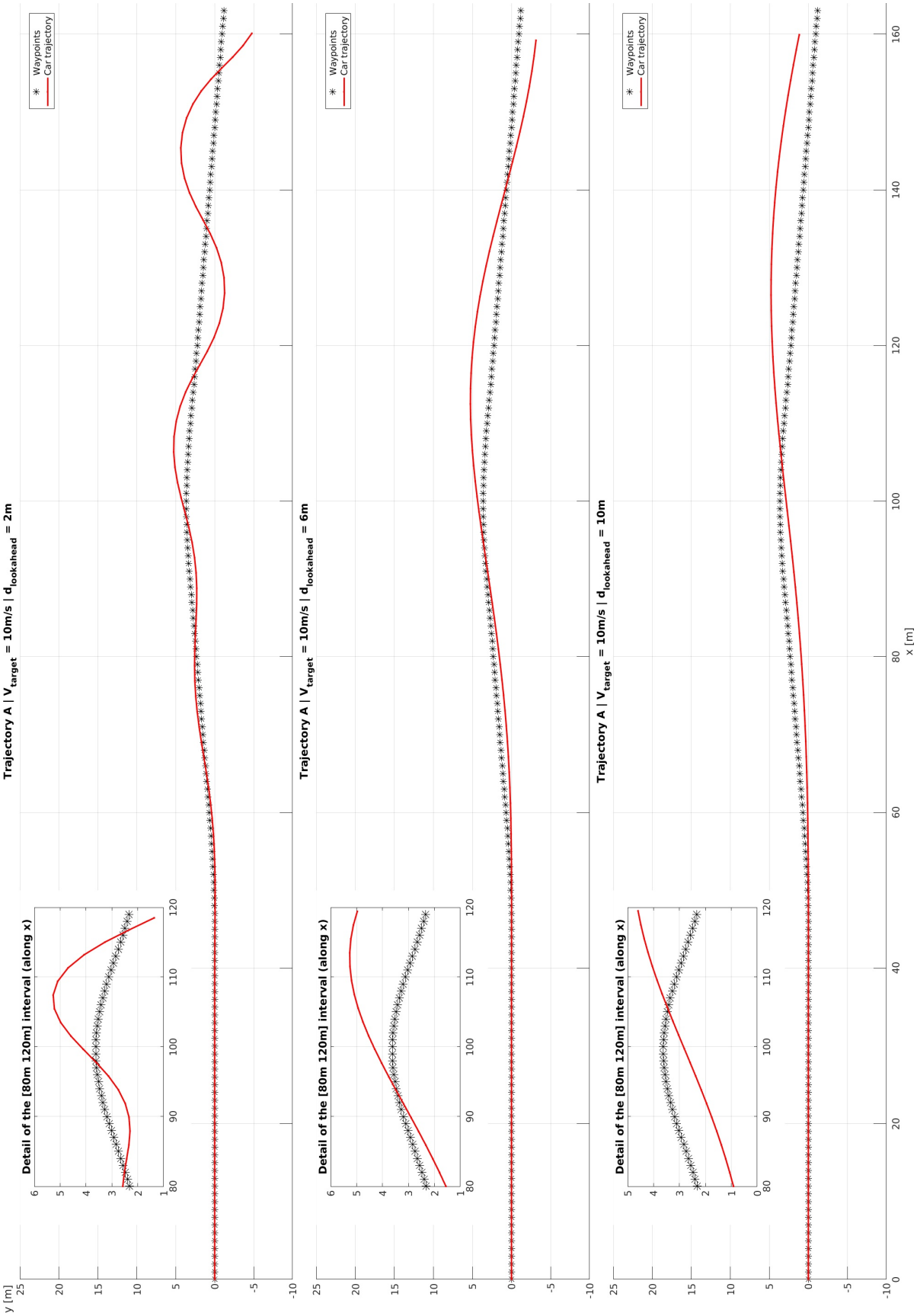


Figure 42: Results of simulation using trajectory A with fixed target speed of $10m/s$ at the variation of the lookahead distance

6 Further development and conclusion

In this last part is presented a collection of ideas to improve the framework, together with the evaluation of the obtained results.

6.1 Future improvements

The presented framework can be improved and expanded in several different ways, being it a platform born to grow through the addition of modules, to finally reach a state that makes possible deployment and testing on a real vehicle.
The next step would be to close the control loop in CarMaker, implementing the possibility to generate a costmap starting from the simulated environment.
The way to obtain this can be expressed as a series of incremental steps, of increasing complexity:

Step 1) Implement a node able to perform advanced environmental modeling using information retrieved from the CarMaker simulation. After this is done, it would be possible to set up a scenario containing obstacles like trees, walls and other cars, that will be taken into account to create a costmap representing the environment. This improvement will be even more important after the possibility to update the costmap considering moving obstacles is implemented, feature that is being developed at the time of writing by other students working on a different thesis project.

Step 2) Add measurement errors and noise to the state variables that are used in the control feedback loop. In this way it would be possible to simulate a realistic use case where no quantities can be obtained with absolute precision, unlike it is done in this project thanks to the CarMaker dictionary blocks.

Step 3) Instead of adding generic measurement errors to the data obtained from the simulated vehicle in CarMaker, add a set of **simulated sensors** like LIDARs, GPS receivers, IMUs, to make the simulation become a proper test bench that can be used before deploying the system on a real vehicle. This is something already possible in CarMaker but the difficulty is to configure the sensors so they reflect the ones used in a real vehicle.

6.1.1 MPC/NMPC controller techniques

Another possible improvement is to design and use in the motion planner a different type of controller known for its good performances in many different scenarios: *Model Predictive Controller* (MPC)[18,19] or its nonlinear variant (NMPC). These techniques have the ability to control specific processes under sets of constraints that can be changed during the execution. The main disadvantage is that complex optimization problems need to be solved every time step. This is quite time consuming and due to this reason these controllers are usually designed to provide a control command every defined amount of cycles.

6.2 Results and final considerations

From the start of this project it was clear that the main result to achieve would've been the addition of an important module to an already existing complex framework. Other students are contributing to the completion of the whole system, adding new functionalities like the ability to avoid moving obstacles, but there are still some steps missing before this autonomous driving package can be deployed on a real car. Most likely one of the future steps will involve a project dedicated exclusively to the validation on a real test vehicle.

The controller infrastructure designed and integrated in the framework, together with the validation process defined, will be available to anyone that will continue with the development.

Some of the results achieved in this thesis project:

- It's clear how a simulation software like CarMaker is essential in the development of an autonomous driving framework, thanks to the possibility to perform SIL (*Software In Loop*) validations in different conditions and scenarios, really useful to tune controllers and algorithms. Without a simulated vehicle to be used as a plant to control, the whole validation step would've been significantly more difficult.

- The infrastructure developed is a good test bench where other motion planning techniques can be developed, validated and tuned. Using the Simulink model created in this project it's really easy to replace the controller block with another one and immediately test the new behavior. While pure pursuit isn't the best technique available, it still gives good results considering the low amount of time required for the algorithm to run and its simplicity.

- For what concerns the usage of a navigation method like pure pursuit on a real vehicle, an important observation related to the lookahead distance can be made. With the selection of a fixed value (or a small range, with the value to use chosen depending on other conditions) of the lookahead distance, a constraint on the number of possible trajectories that the path planner needs to calculate is introduced. While for the short range navigation the whole ROS framework based on cameras, LIDARs and IMUs is needed to have an accurate representation of the environment, for the long range planning only the GPS receiver is sufficient, considering that with the definition of the maximum lookahead distance a big constraint is introduced and there is no need to know precisely the road and the surroundings so far away from the vehicle.

Hopefully the results obtained will be good starting points for the next developers taking over the development of the framework. They will find a test bench already set up that can be used for validation and to perform experiments with more complex control systems, to finally conclude this ambitious project, deploying the system to a real vehicle to provide it autonomous driving capabilities.

Appendix A: Additional figures

In this section are included additional figures omitted from the body of chapter 5 (*Controller design*) but still relevant to understand the behavior of the developed controllers. Note that some of the presented figures don't show optimal behaviors, they are included to demonstrate the influence of some parameters on the results.

Figure	**Page**	**Description**
LA1	51	Large value of lookahead distance, tracking not accurate
LA2	51	Smaller value of lookahead distance with respect to LA1, improvements in tracking
SP1	52	Speed tracking result ($f = 0.4Hz, v_{target} = 2m/s$)
SP2	52	Speed tracking result ($f = 0.05Hz, v_{target} = 14m/s$)
YR1	53	Yaw rate tracking result ($f = 0.1Hz, yr_{target} = 0.2rad/s$)
YR1	53	Yaw rate tracking result ($f = 0.4Hz, yr_{target} = 0.2rad/s$)
A1	54	Results of simulation using trajectory B with fixed target speed of $3m/s$ at the variation of the lookahead distance [$2m$, $3m$, $6m$]
A2	55	Results of simulation using trajectory B with fixed target speed of $10m/s$ at the variation of the lookahead distance [$2m$, $6m$, $10m$]
A3	56	Results of simulation using trajectory C with fixed target speed of $3m/s$ at the variation of the lookahead distance [$2m$, $3m$, $4m$]
A4	57	Results of simulation using trajectory C with fixed target speed of $10m/s$ at the variation of the lookahead distance [$2m$, $3m$]

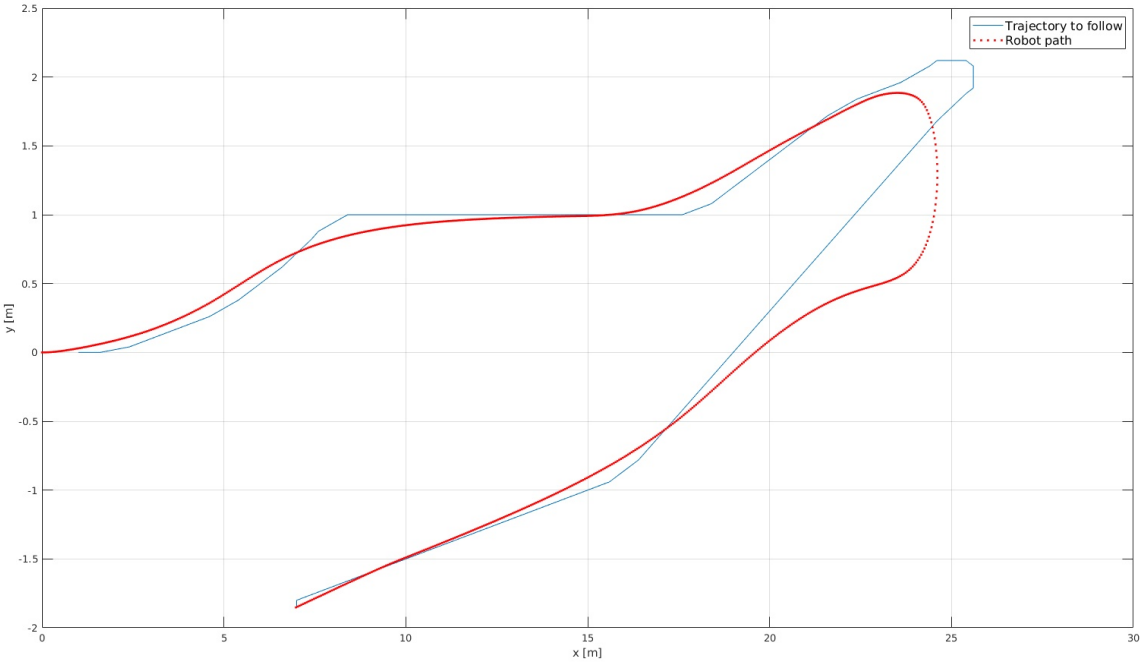


Figure LA1

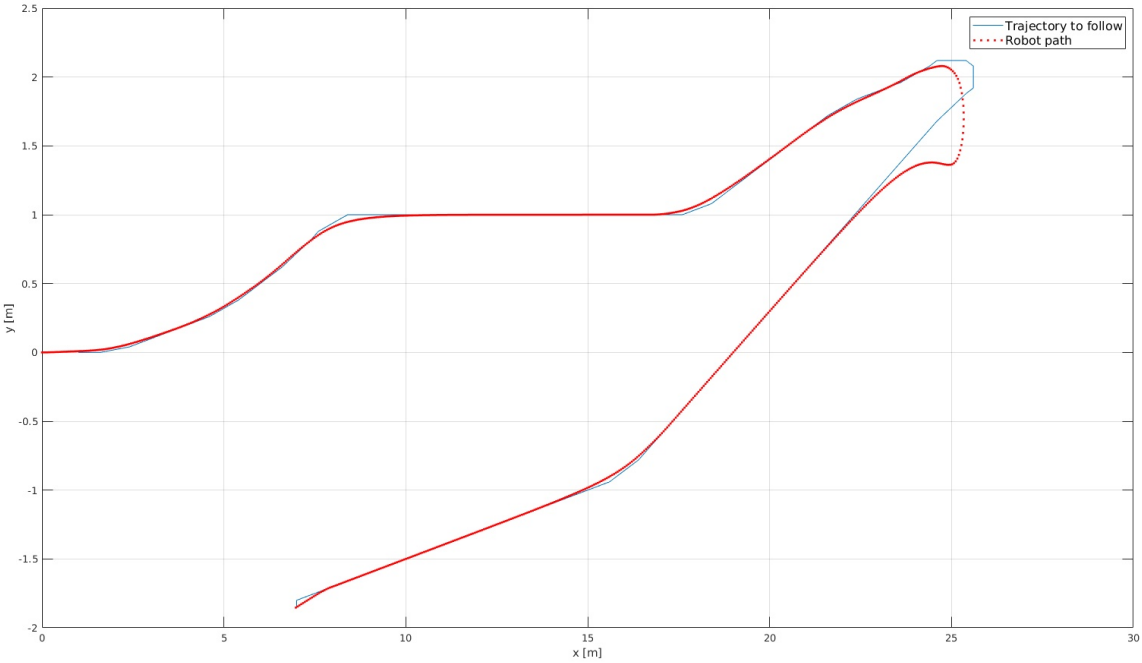


Figure LA2

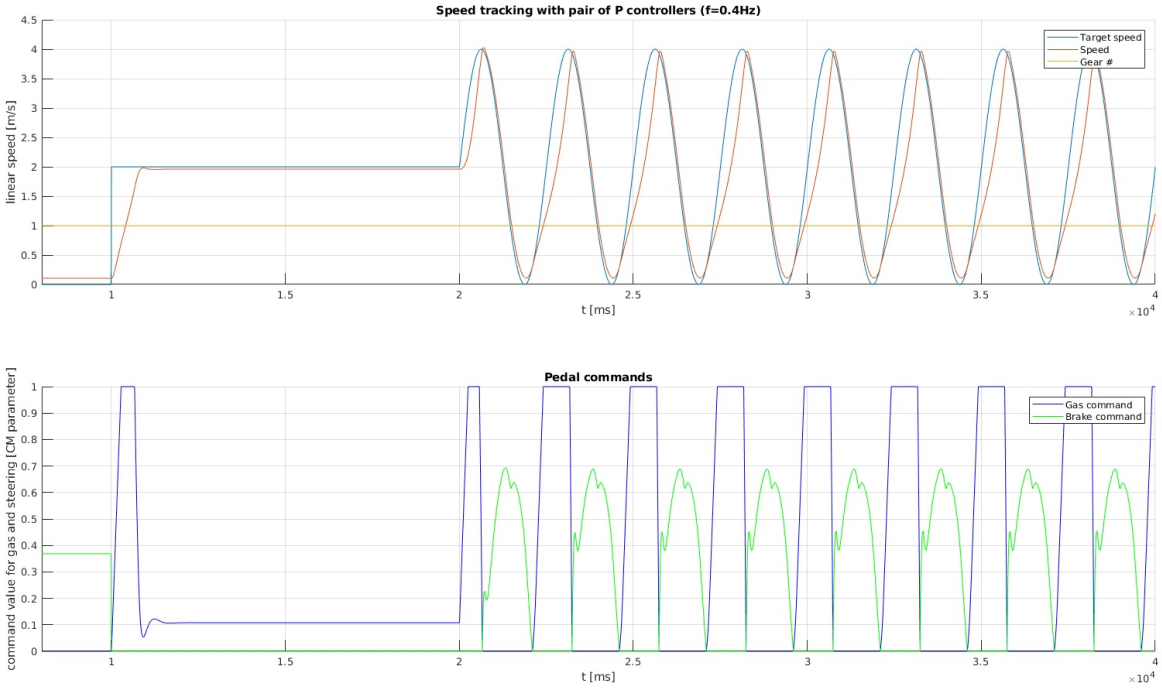


Figure SP1

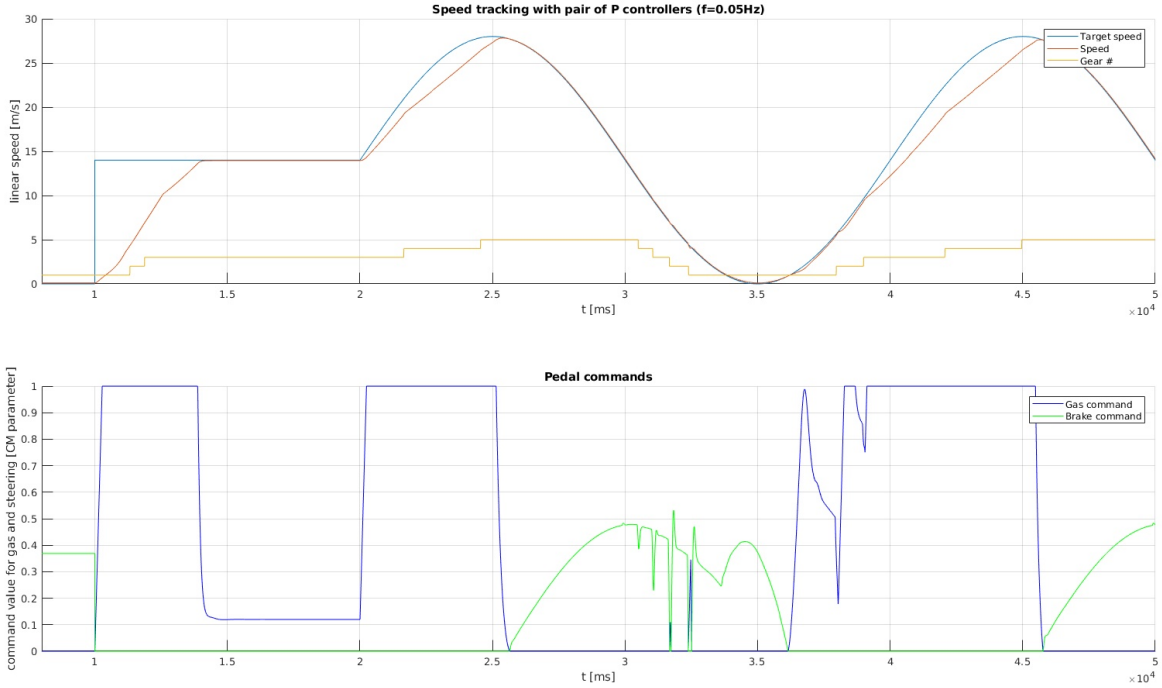


Figure SP2

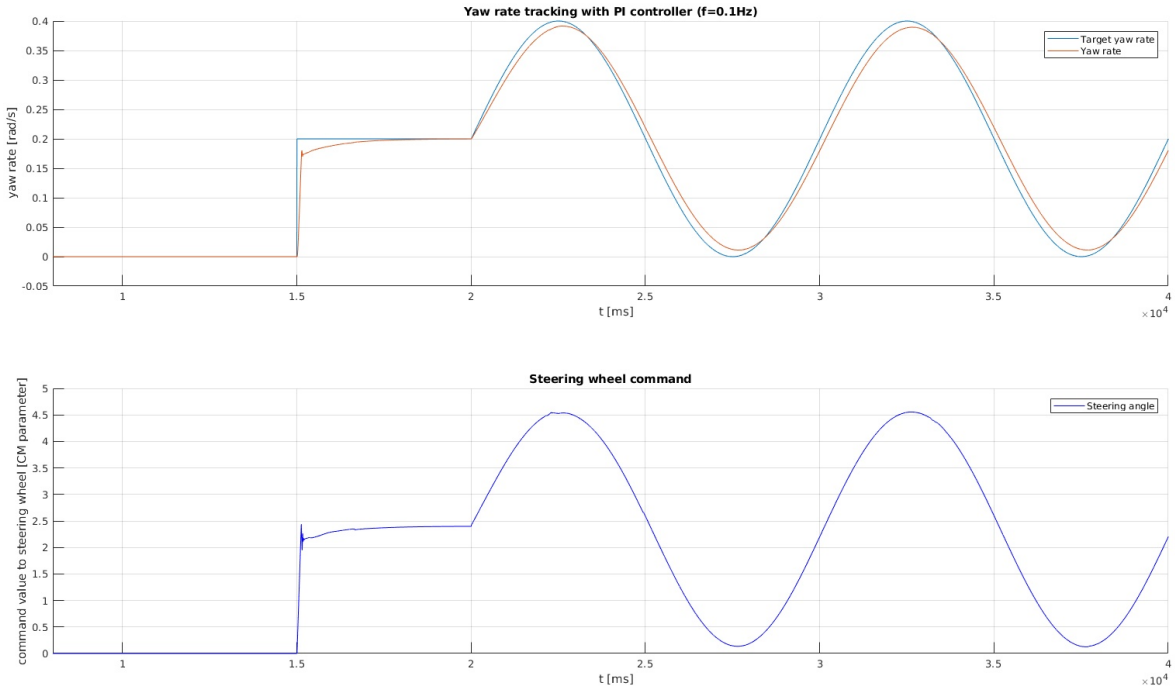


Figure YR1

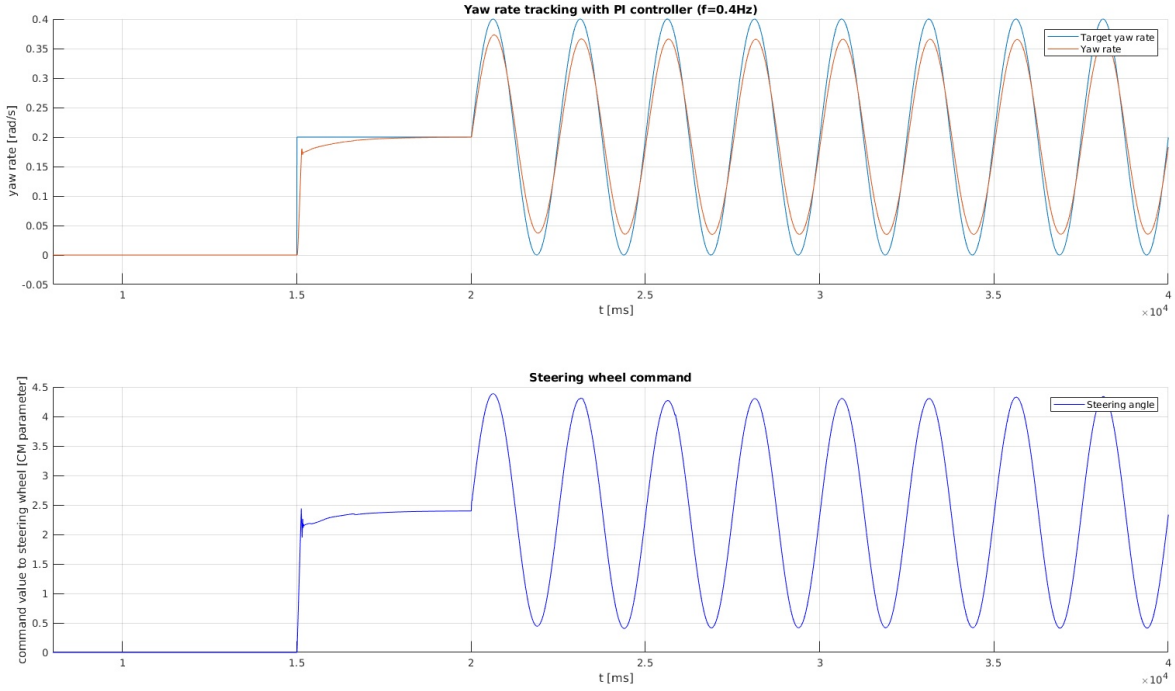


Figure YR2

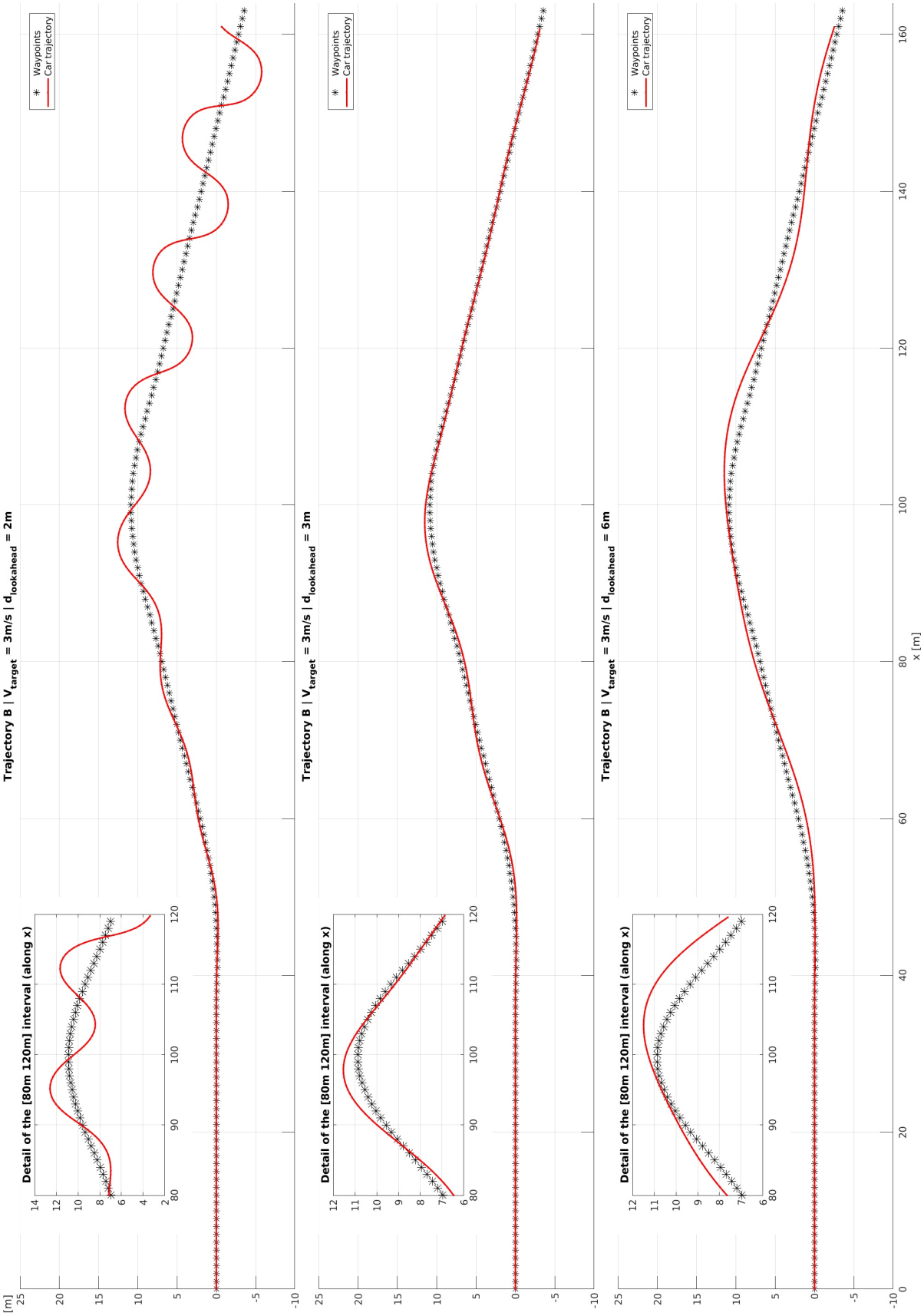

Trajectory B | V_target = 3m/s | d_lookahead = 2m
Trajectory B | V_target = 3m/s | d_lookahead = 3m
Trajectory B | V_target = 3m/s | d_lookahead = 6m
Waypoints
Car trajectory
Detail of the [80m 120m] interval (along x)
y [m]
x [m]

Figure A1

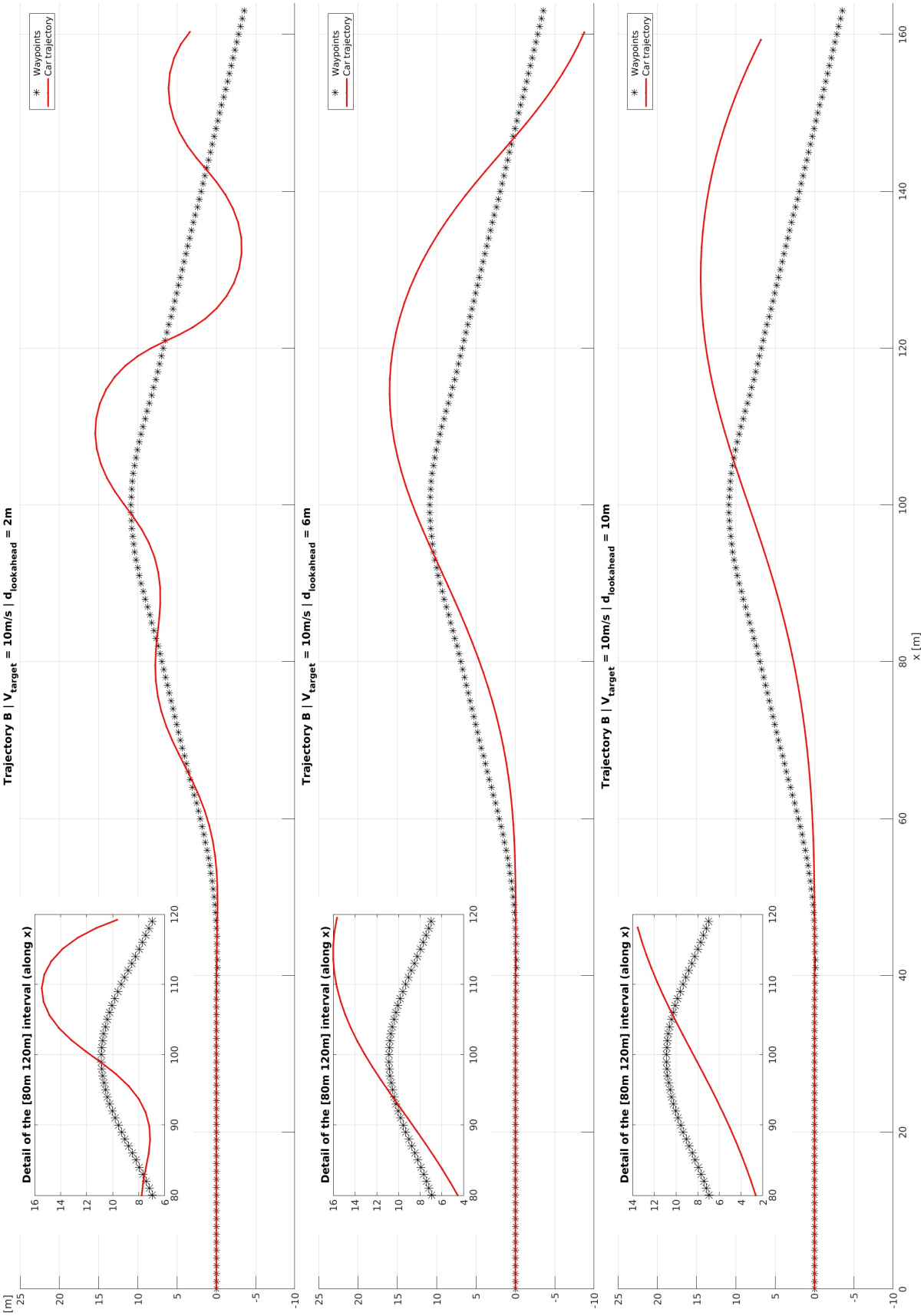

Trajectory B | V_target = 10m/s | d_lookahead = 2m
Trajectory B | V_target = 10m/s | d_lookahead = 6m
Trajectory B | V_target = 10m/s | d_lookahead = 10m
Waypoints
Car trajectory
Detail of the [80m 120m] interval (along x)
y [m]
x [m]

Figure A2

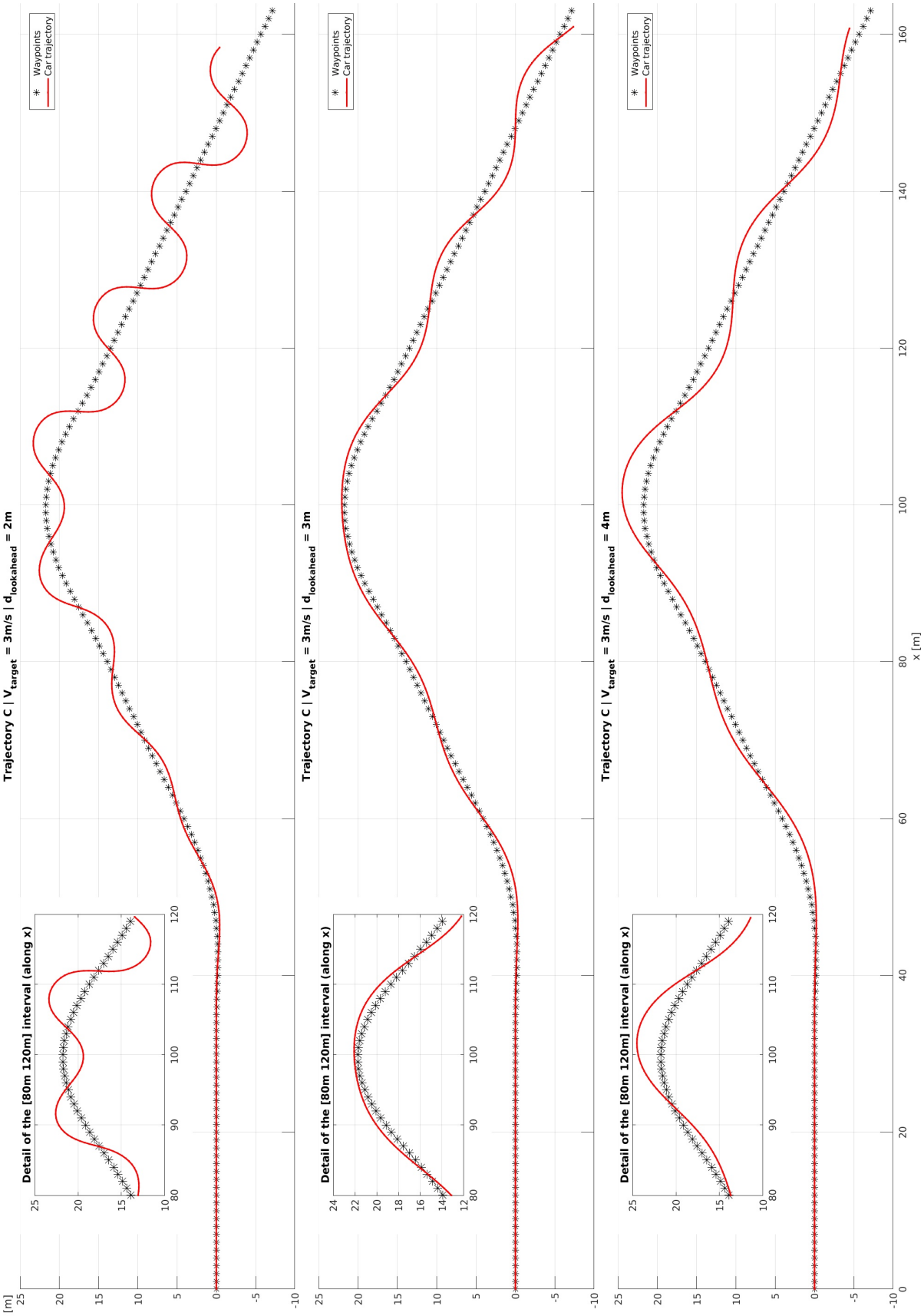

Trajectory C | V_target = 3m/s | d_lookahead = 2m
Trajectory C | V_target = 3m/s | d_lookahead = 3m
Trajectory C | V_target = 3m/s | d_lookahead = 4m
Waypoints
Car trajectory
Detail of the [80m 120m] interval (along x)
y [m]
x [m]

Figure A3

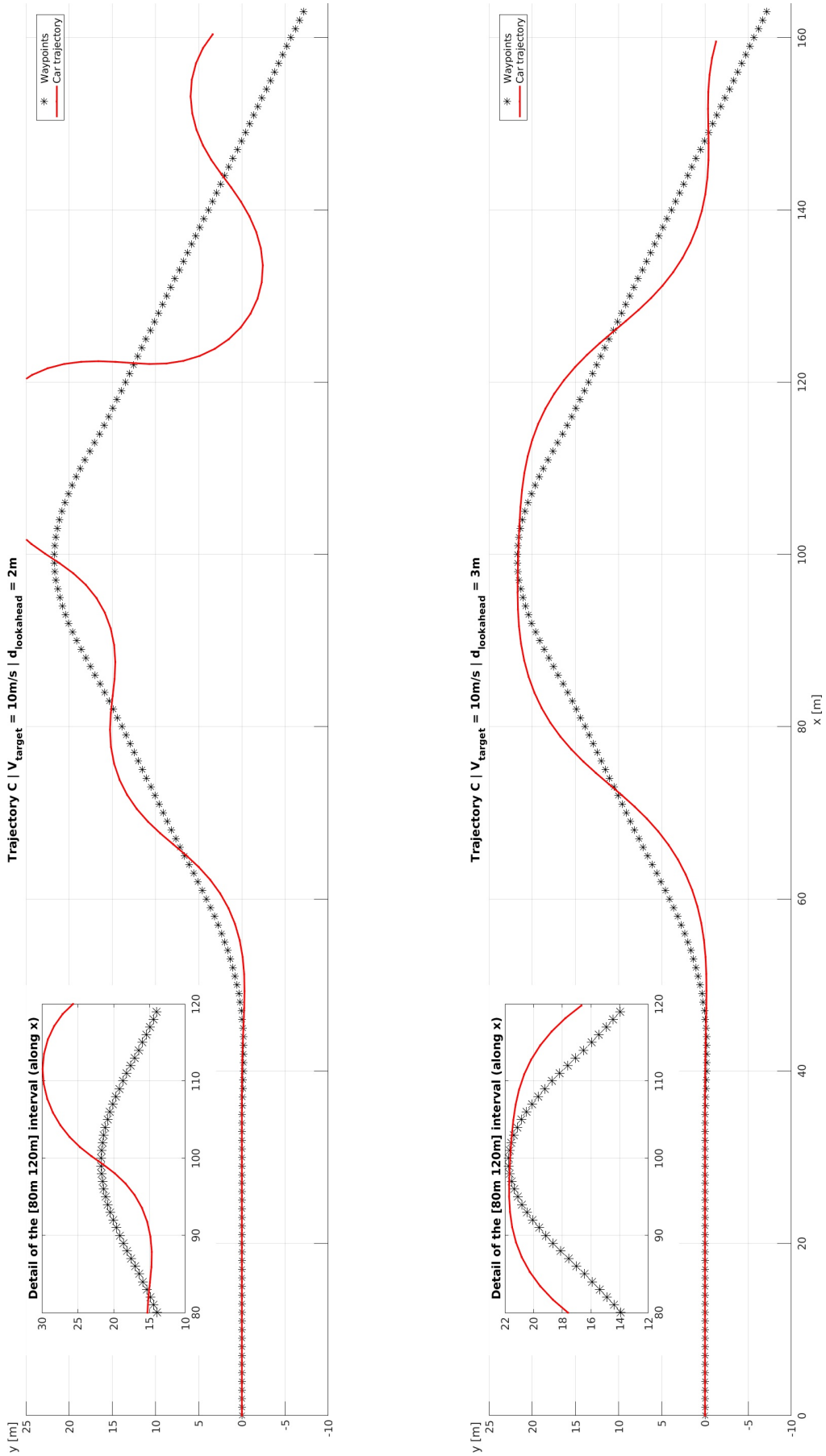

Trajectory C | V_target = 10m/s | d_lookahead = 2m
Waypoints
Car trajectory
Detail of the [80m 120m] interval (along x)
y [m]
Trajectory C | V_target = 10m/s | d_lookahead = 3m
Waypoints
Car trajectory
Detail of the [80m 120m] interval (along x)
y [m]
x [m]

Figure A4

References

[1] S. **Barrera**, V. **Comito**.
A ROS-Based Platform for Autonomous Vehicles: Foundations and Perception.
Polytechnic University of Turin, 2018.

[2] D. **González**, J. **Pérez**, V. **Milanés**, F. **Nashashibi**.
A Review of Motion Planning Techniques for Automated Vehicles.
IEEE Transactions on Intelligent Transportation Systems, 2016.

[3] **ROS** website.
`http://www.ros.org/`

[4] M. **Quigley**, K. **Conley**, B. **Gerkey**, J. **Faust**, T. **Foote**, J. **Leibs**.
ROS: an Open-Source Robot Operating System.
ICRA Workshop on Open Source Software, 2009.

[5] J. **Beltrán**.
BirdNet: a 3D Object Detection Framework from LIDAR Information.
IEEE 21st International Conference on Intelligent Transportation Systems, 2018.

[6] R. **Mur-Artal**, J D. **Tardós**.
ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras.
IEEE Transactions on Robotics, 2017.

[7] C. **Kerl**, S. **Jürgen**, C. **Daniel**.
Dense Visual SLAM for RGB-D Cameras.
IEEE/RSJ International Conference on Intelligent Robots and Systems, 2013.

[8] **YOLO** website.
`https://pjreddie.com/darknet/yolo/`

[9] J. **Redmon**.
You Only Look Once: Unified, Real-Time Object Detection.
Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016.

[10] R. **Coulter**.
Implementation of the Pure Pursuit Path Tracking Algorithm.
Camegie Mellon University, 1990.

[11] J. **Morales**, J. L. **Martínez**, M. A. **Martínez**, A. **Mandow**.
Pure-Pursuit Reactive Path Tracking for Nonholonomic Mobile Robots with a 2D Laser Scanner.
EURASIP Journal on Advances in Signal Processing, 2009.

[12] J. M. **Snider**.
Automatic Steering Methods for Autonomous Automobile Path Tracking.
Robotics Institute, Pittsburgh, PA, 2009.

[13] P. E. **Hart**, N. J. **Nilsson**, B. **Raphael**.
A Formal Basis for the Heuristic Determination of Minimum Cost Paths.
IEEE Transactions on Systems Science and Cybernetics SSC4, 1968.

[14] P. **Polack**, F. **Altché**, B. **D'Andréa-Novel**, A. **De La Fortelle**.
The Kinematic Bicycle Model: a Consistent Model for Planning Feasible Trajectories for Autonomous Vehicles?
IEEE Intelligent Vehicles Symposium (IV), Los Angeles, 2017.

[15] J. **Kong**, M. **Pfeiffer**, G. **Schildbach**, F. **Borrelli**.
Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design.
IEEE Intelligent Vehicles Symposium (IV), Seoul, 2015.

[16] J. G. **Ziegler**, N. B. **Nichols**.
Optimum Settings for Automatic Controllers.
Rochester, N. Y., 1942.

[17] P. **Cominos**, N. **Munro**.
PID Controllers: Recent Tuning Methods and Design to Specification.
IEE Proceedings-Control Theory and Applications, 2002.

[18] J. B. **Rawlings**, D. Q. **Mayne**.
Model Predictive Control: Theory and Design.
Madison, Wisconsin: Nob Hill Pub., 2009.

[19] F. **Borrelli**, P. **Falcone**, T. **Keviczky**, J. **Asgari**, D. **Hrovat**.
MPC-Based Approach to Active Steering for Autonomous Vehicle Systems.
International Journal of Vehicle Autonomous Systems, 2005.