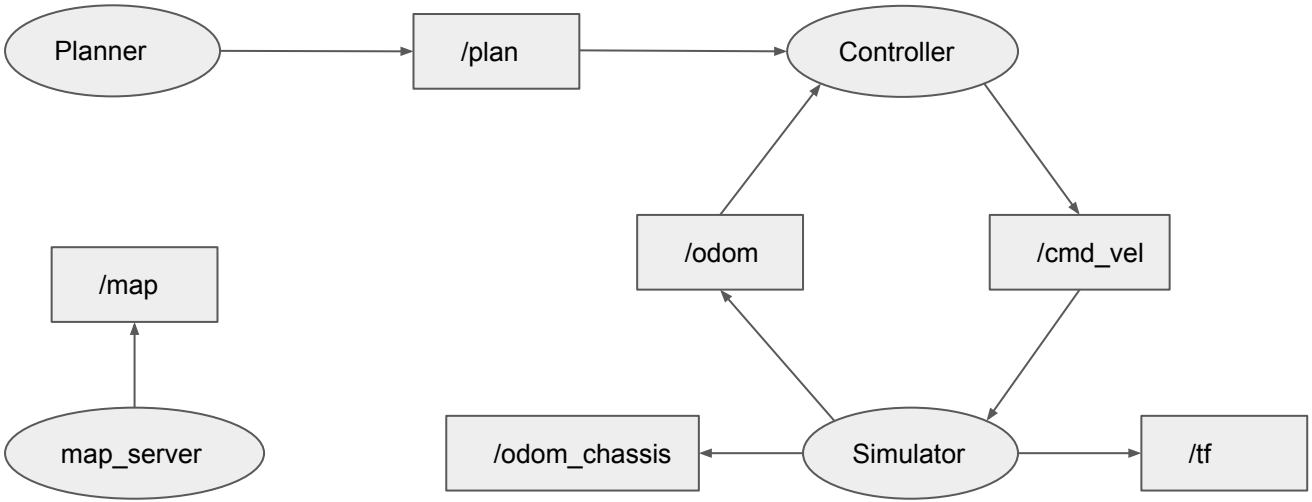
Control of mobile robots

Group 6
Andrea Bonatti 10520865
Samuele Camnasio 10566482

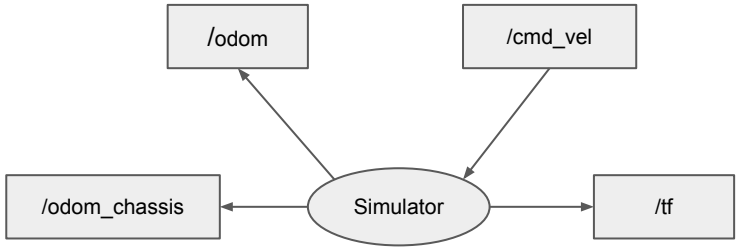
Node network:

Here the nodes that are used for both the control of the car and the visualization of the results via Rviz



Simulator node structure:

The simulator node has two different implementations, one for the kinematic model and one for the dynamic model of the car. Regardless of which implementation is used, the interface remains the same.
The node receives the */cmd_vel* as a Twist message with linear velocity and steering angle. The Simulator then publishes the odometry via the */odom* topics an Odometry message, that is subscribed by the controller, and the */odom_chassis* topic, which is used by Rviz. The Simulator also links the frame of the car to the world via the */tf* topic. Since that there are no constraints on the simulation environment (we don't use Gazebo and as such we are not required a publishing rate of 30 Hz) and we have no need for external sensors, the node rate and the publishing rate have been both set at 100Hz which can be easily handled by the node.

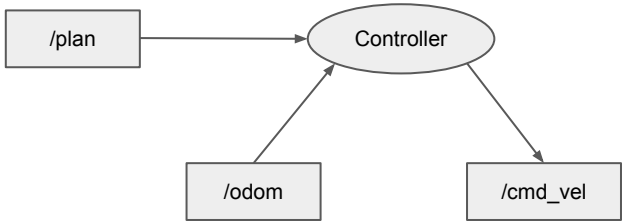


Planner node structure:

The planner outputs, only once at the start, the position of the various waypoints that the robot has to follow. The points are transmitted via the */plan* topic. There are 2 planner nodes, with the same structure, to publish the 2 different trajectories, computed with sPRM and RRT respectively.

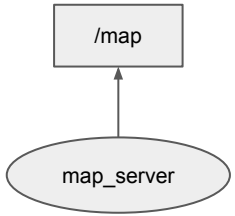
Controller node structure:

The controller receives the set of points from the planner via the */plan* topic. It then computes the commands for velocity and the steering angle that the car has to follow to reach each waypoint and their sub-objective based on the current position of the robot in the world frame, read from the /odom topic. The commands are published over the */cmd_vel* topic. The node and publishing frequence have been set at 50Hz, obviously lower than the publishing rate of the simulator.



Map server:

We used the map_server node of ROS to display over Rviz the map of the maze where the car moves.



Assumptions

To simplify the model of the car we have done a series of a priori assumptions on the kinematic and dynamic model and on the world where the car operates.

Model of the car and interaction with the world:
The model of the car is simplified as a bicycle model with “no apparent dimensions” this means that while we consider the length of the frame of the bicycle in the dynamic/kinematic models, it does not occupy any space in the world environment, it will be treated as a “point moving in the space”. Collision with walls have been avoided by tuning various parameters (mainly in the planner and in the controller). There is no need for localization and mapping since we consider that the odometry is always correct and immune to noise.

Kinematic model:
The scheme of the kinematic model can be later seen in the following pages but it’s important to specify that since the steering angle is a control input (and not a state variable) the model accepts instantaneous variations in its value.

Dynamic model:
The interaction wheel-ground has been considered following the Fiala model. The longitudinal force of front wheel has been considered null since the car is rear wheel drive. The slip angle β of the car has been considered “small” to avoid contributions of the longitudinal force of the rear wheel in the equation of the rate of change of the slip angle. Having ignored all contributions of the longitudinal forces (and without having information about them in the model) we consider the friction circle constraint satisfied. Each wheel has a load of half the gravitational force of the car.

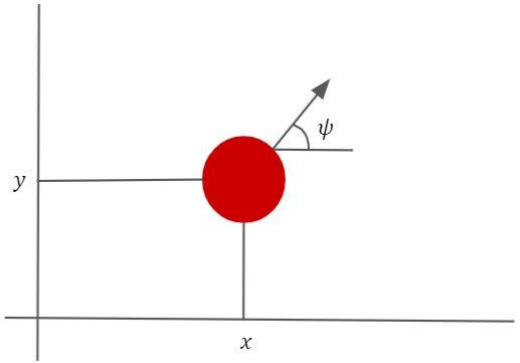
Planner:
The planner publishes only the first 25 poses of the trajectories, due to the limitations of ROS with Simulink.

Controller:
As previously specified, the car can is seen as a single point, this lower the complexity of the controller since all it has to consider is a single point with a certain pose. Moreover we have forced the steering angle to be limited in [-45°,+45°].

Kinematic simulator

The node is subscribed to the */cmd_vel* topic which is composed by Twist message where the linear field has the velocity, while the angular field has the steering angle δ. The node publishes two different odometries. The first one published on the */odom* topic has the pose of the car simplified as a single point without a frame as in figure. The yaw ψ is the sum of the yaw of the frame of the car and the slip angle β so that the velocity is always applied in with the same verse and direction of the state vector.

$$[x,\ y,\ \psi]^T$$



The second odometry is published over */odom_chassis*, while it has the same xy components, the angle is only the yaw of the frame. This distinction is useful to visualize on Rviz the two different vectors of the "heading" of both the frame and the velocity vector.

This node is also responsible to publish the clock for the synchronization with the other nodes

Kinematic model

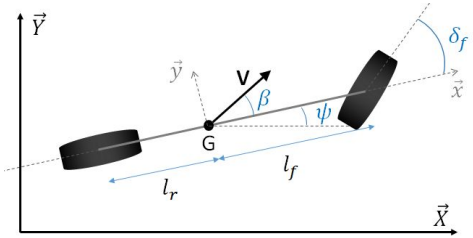
The model equations are as follows:

$$\frac{dx_g}{dt} = V\cos(\psi + \beta)$$

$$\frac{dy_g}{dt} = V\sin(\psi + \beta)$$

$$\frac{d\psi}{dt} = \frac{V\cos(\beta)\tan(\delta)}{l_r + l_f}$$

$$\beta = atan(\frac{l_r \tan(\delta)}{l_r + l_f})$$



Dynamic simulator

The Dynamic simulator subscribes and publishes on the same topics as the Kinematic simulator (*/cmd_vel, /odom, /odom_chassis, /clock, /tf*). The difference lies in the model.

Dynamic model

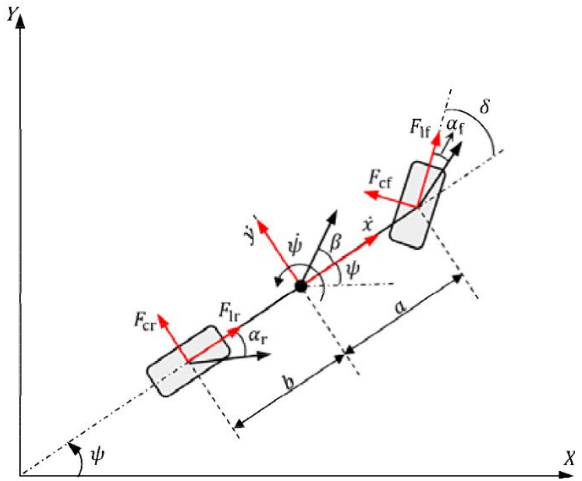
Starting from the Newton equations for the translational motion of the center of mass and adding the kinematic state of the model we find:

$$\frac{d^2\psi}{dt^2} = \frac{aF_{cf}\cos(\delta) - bF_{cr}}{I_z}$$

$$\frac{d\beta}{dt} = \frac{F_{yf}\cos(\delta) + F_{yr} - m\frac{d\psi}{dt}v}{mv}$$

$$\frac{dx}{dt} = V\cos(\psi+\beta)$$

$$\frac{dy}{dt} = V\sin(\psi+\beta)$$



As said in the assumptions, we have considered null the contribution of the longitudinal forces on the tires, so only the lateral forces influence the behaviour of the car, and we have considered small β.

Dynamic model

To model the wheel-ground interaction we have used the Fiala tire model:

$$F_c = \begin{cases} -C_\alpha \tan(\alpha) + \dfrac{{C_\alpha}^2 \tan(\alpha)\left|\tan(\alpha)\right|}{3\mu F_z} - \dfrac{{C_\alpha}^3 \tan^3(\alpha)}{27\mu^2 {F_z}^2} & |\alpha| < a\tan\left(\dfrac{3\mu F_z}{C_\alpha}\right) \\ -\mu F_z sgn(\alpha) \end{cases}$$

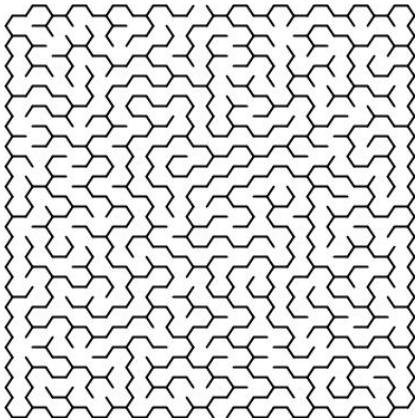
Simplifying then the slip angles of the forward and backward wheel as:

$$\alpha_f \approx \beta + \frac{a\dfrac{d\psi}{dt}}{v} - \delta$$

$$\alpha_r \approx \beta - \frac{b\dfrac{d\psi}{dt}}{v}$$

Planning

The maze is made by 20x20 hexagonal cells, with a size large enough to allow the bicycle to move properly inside of it.



Preprocessing

Maze is obtained from http://www.mazegenerator.net/ as a .png file.
A relative .yaml file, needed also to publish the /*map* topic, is handwritten to describe the map attributes:

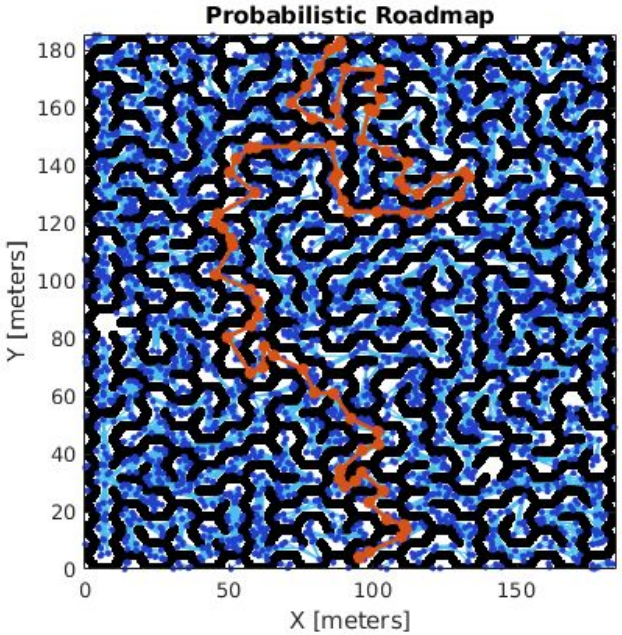
- origin coordinates: [0.0 ,0.0, 0.0]
- occupied threshold: 0.65
- free threshold: 0.196
- resolution: 0.5, to have a large enough maze
- negate: 0, not needed

To be used in the planning algorithm it is loaded in matlab and turned into grayscale with rgb2gray function.

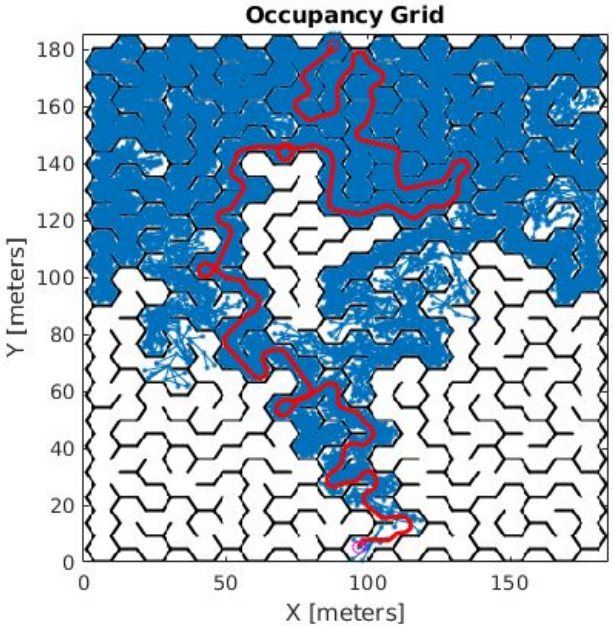
Notes and considerations: the trajectories are computed on a not very performant laptop, this may lead to a high very computational time, that might be smaller on better PCs, the focus of the result is not the exact time but mainly the comparison between the 2 algorithms, which says sPRM is much faster with this kind of map. In general our experiments said that sPRM is overall better and more reliable than RRT in obtaining a trajectory for this maze.

Example of solutions obtained with 2 different algorithms

Simplified Probabilistic RoadMaps



Rapidly-exploring Random Trees



Parameters setting and considerations

Simplified Probabilistic RoadMaps (sPRM):

- good solutions found with a relative small number of nodes, around 3000
- reasonable computational time (a few minutes)
- connection distance: best solution found with ‘15’, which well fits the maze dimensions, with smaller values more nodes are required, increasing computational time
- occupied location for inflate: best solution found with ‘1’, to have a reasonable distance from the obstacles(wall), probably it would have been safer to have a larger value, but it made the computation of a solution much less feasible, fortunately the experimental results are still good and provide a reliable solution for our bicycle

Rapidly-exploring Random Trees (RRT)

- finding a solution requires a high value for the ‘max tree nodes’, and ‘max iterations’, due to the large size of the maze, the best results were obtained by keeping these 2 parameters with the same order of magnitude.
- due to the high number of nodes and iterations, a large computational time is required, this also made not so easy to make experiments and obtain feasible solutions
- min turning radius in the state space: 2.5, to have a solution which is feasible for our bicycle models, with respect to the angles of curvature performed. Smaller values, like 1.0, allows to find a solution with a not too high number of nodes, but the solution found will be not feasible for our bicycle.
- validation distance: 1.0, good to obtain connections feasible with the constraints(walls), having greater values often provides unacceptable solutions(walls cross)
- max connection distance: small values require a larger number of nodes to find a solution. Results obtained with values between ‘15’ and ‘50’, after experiments 50 is the best found value since it avoids requiring too much nodes
- max iterations: in order of 100k, if smaller ‘max tree nodes’ up to 1 million iterations requires, best solution with 850k
- max tree nodes: in order of 100k, best solution with 800k
- goal bias: 0.25, the default value is 0.05, i’ve tried to gradually increase it and noticed that it improved the finding of a solution,
- number of steps in interpolation: values between 150 and 300, 300 is a good solution but due to the limitation of publishing only 25 steps the simulation in rviz does not go very far and so the result displayed is not very meaningful to see. For this reason the result provided in the generated node is the one with 150 interpolation steps, which performs worse, especially with the dynamic model, trepassing some wall at some points, but allows to see the motion for a longer period

Algorithms comparison

	sPRM	RRT
Computational time:	reasonable, few minutes	high, many minutes, increasing a lot with number of nodes and iterations
number of nodes required	reasonable	very high to obtain a good solution
steps in the obtained path	~77	between 150 and 300, depends on the final interpolation, more the nodes more the smoothness of the solution
quality of solutions	generally good	quite good with a proper ‘minTurningRadius’, if instead it is not well set paths can have some unfeasible curve for a bicycle. Some points are too close to the obstacles. Sometimes small useless loops are created.
smoothness of solution	quite good	good, due to the many number of states.
sensitivity to parameters tuning	not too much, less options with respect to RRT, easier to tune	a lot, ‘validationDistance’ can provide solutions that violate the walls constraints, ‘minTurningRadius’ can provide solutions that looks ok but are then not feasible when used in the simulation

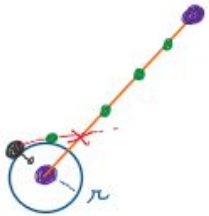
Controller

The controller node is responsible to publish the commands for the car to follow moving inside the maze, it subscribes to the */plan* topic and publishes the */cmd_vel* topic. The node is composed by two principal components, one in charge of the interpolation of the trajectory and the other of both the feedback linearization and proportional tracking controller.

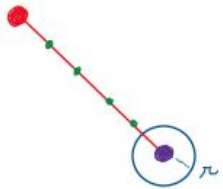
Trajectory interpolator

The trajectory interpolator is in charge of creating “sub-objective” over the entire trajectory and switching between them as the car reaches them. Since the car is unable to perform tight corners (which during testing caused the car to remain stuck) the interpolator has to smooth over the trajectory. Moreover since the path is inside a maze and speed is not required, the interpolator will set small step objective to keep the difference between the position of point P and the current objective small. In Rviz the current step objective is a green dot, the last waypoint and the current waypoint (both extracted from the */plan*) are respectively Red and Purple.

For the first objective, once the robot has reach a position inside a certain radius from the current waypoint (the purple one), the interpolator will find the next waypoint in list and the equation of the line between them. It then selects the point at a distance of 1m from the old waypoint (red cross in the picture) and sets the middle of the segment that connects this point to the current car position as the current step objective (green point).



For the second objective the interpolator will update the step objective (green point) every time the car reaches a certain distance from it, finding the next one on the line that connects the old waypoint to the current one.



Feedback linearizing law

The feedback linearizing law has been designed over the kinematic model of the bicycle and the result is:

$$
\begin{aligned}
v &= v_{xp}\, cos(\vartheta) + v_{yp}\, sin(\vartheta) \\
\delta &= a\tan(\frac{l_r + l_r}{\varepsilon} \frac{v_{yp}\, cos(\vartheta) - v_{xp}\, sin(\vartheta)}{v_{xp}\, cos(\vartheta) + v_{yp}\, sin(\vartheta)})
\end{aligned}
$$

Trajectory controller

The proportional trajectory tracking controller with velocity feedforward we used is:

$$
\begin{aligned}
x_p &= x + \varepsilon\, \cos(\vartheta) \\
y_p &= y + \varepsilon\, \sin(\vartheta) \\
x_{pd} &= x_d + \varepsilon \cos(\vartheta) \\
y_{pd} &= y_d + \varepsilon \sin(\vartheta) \\
v_{xp} &= \dot{x_d} + KP(x_{pd} - x_p) \\
v_{yp} &= \dot{y_d} + KP\left(y_{pd} - y_p\right)
\end{aligned}
$$

Controller parameters setting

- Proportional gain (KP): needs to be a positive value, was set by experimental tuning to 0.5, values close to it are also fine, increasing it provides a faster convergence to the next point, decreasing it results in a slower convergence
- Distance of the point (epsilon): needs to be any small positive value, like 0.5, that provides good results. The smaller the value the more the trajectory is followed strictly
- Ball radius: range within which the next waypoint is considered as reached. 2 was a good value due to the dimension of the bicycle and the experimental tuning we did. Slower values(like ‘1’) give slower and less smooth motions,but still feasible. Higher values can give smoother motions but if too high(like ‘5’) can also introduce violations of walls constraints

Tracking controller validation

The controller acts generally well with the planned paths with both the kinematic and dynamic model simulators.

in particular, the trajectories provided by sPRM behave very well with the interpolation parameters we chosen and are not very sensitive to small changes, of course increasing the value of epsilon a lot makes the bicycle pass farther from the strict trajectory, and this causes sometimes an undesired crossing of obstacles. So in general smaller epsilon are more reliable values.

RRT instead has a more sensitive behavior, due to the less good quality of certain solutions, changing the controller parameters can cause the impossibility to perform some maneuvers, leaving the bicycle stuck. this is also given by a selection of parameters in the planning algorithm that was not appropriate for our bicycle model. Also in this case a small epsilon makes the movement more faithful to the received trajectory, but increasing it a bit can sometimes help in situations where the path has some not nice corner by providing smoother movements. Even the best obtained trajectories have some points that are too close to the walls, so the simulation does not provide a perfect performance, in this case 0.5 is the best value for epsilon, since with both higher or lower values the collision with the obstacle gets accentuated.

In conclusion sPRM trajectories are better managed by the controller, since it’s easy to obtain a nice behavior by setting the controller parameters and providing a good planned path, which is much easier to find with respect to the RRT algorithm.