Technical Report

IRI-TR-15-07

Autonomous navigation framework for a car-like robot

Sergi Hernandez Juan
Fernando Herrero Cotarelo

July, 2015

Institut de Robòtica i Informàtica Industrial

Abstract

This technical report describes the work done to develop a new navigation scheme for an autonomous car-like robot available at the Mobile Robotics Laboratory at IRI. To plan the general path the robot should follow (i.e. the global planner), a search based planner algorithm, with motion primitives which take into account the kinematic constraints of the robot, is used. To actually execute the path and avoid dynamic obstacles (i.e the local planner) a modification of the DWA algorithm is used, which takes into account the kinematic constraints of the ackermann configuration to generate and evaluate possible trajectories for the robot.
The whole navigation scheme has been integrated into the ROS middleware navigation framework and tested on the real robot and also in a simulator.

Institut de Robòtica i Informàtica Industrial (IRI)
Consejo Superior de Investigaciones Científicas (CSIC)
Universitat Politècnica de Catalunya (UPC)
Llorens i Artigas 4-6, 08028, Barcelona, Spain
Tel (fax): +34 93 401 5750 (5751)
http://www.iri.upc.edu

Corresponding author:
Sergi Hernandez
tel: +34 93 401 0857
shernand@iri.upc.edu
http://www.iri.upc.edu/staff/shernand

1 Introduction

The Mobile Robotics Laboratory at IRI has the car-like robot shown in Fig. 1. This robot was developed by Robotnik and even though it had no autonomous navigation capability, it could be remotely operated using an external control pad.

Figure 1: Picture of the real ackermann based robot available at the Mobile Robotics Laboratory at IRI.

The main features of the robot are shown in Table 1.

The ROS middleware used at IRI has a navigation framework with standard local and global planners, that works quite well for holonomic and quasi-holonomic robots (i.e. robots that can turn in place), but it does not take into account any kind of constraints necessary for other kinds of robot architectures, like the ackermann (or car-like) configurations. The kinematic constraints associated with an ackermann configuration are introduced in section 2.

For the global planner, a search base planner algorithm from SBPL (see [3]) has been used with primitives carefully generated to take into account the maximum turn radius of the robot. This algorithm has also already been integrated to the ROS framework, which reduced the required work. Section 3 briefly describes this algorithm and give detailed information on the generation of the motion primitives.

No local planner for ackermann architectures was found that could be adapted to our needs, so it was decided to modify the DWA (Dynamic Window Approach) algorithm already used by ROS to take into account the ackermann constraints. All the necessary changes to take into account the non-holonomic constraints are presented in section 4.

Finally, section 5 summarizes the integration of the modified global and local planners into

Table 1: Main features of the car-like robot at IRI.

Feature	Value
Weight	400 kg
Dimensions	$2.5x1.2x1.9\,m$
Max. speed	30 km/h
Max. steer angle	0.45 rad
Max. slope	30 %
Wheel base	1.65 m
Track	1.2 m
Wheel diameter	0.4329 m

the ROS framework, and section 6 shows some of the results and possible future upgrades.

2 Ackermann kinematics

The motion of an ackermann based robot, like a car, can be described on first approximation by the translational speed and the steering angle, and also by the steering speed. Both translational and steering accelerations are only considered when generating trajectory estimations to take into account that the velocities and angles can not change instantaneously.

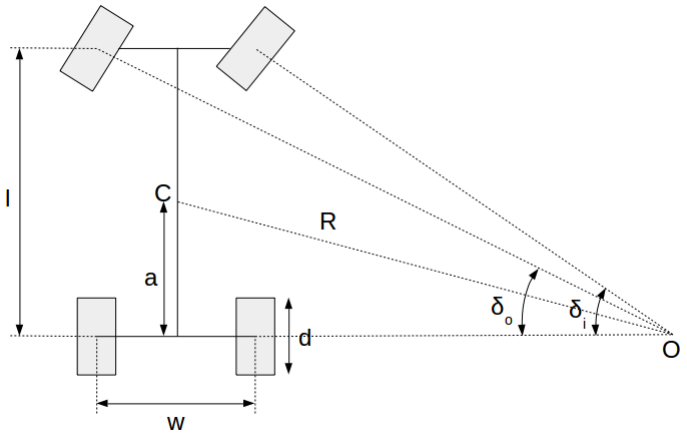


Figure 2: Sketch of an ackermann based robot with all its main parameters.

Fig. 2 shows a sketch of an ackermann architecture. The important parameters of this kind of configurations are:

- wheel base (l): is the distance between the front and rear axles of the robot.
- track (w): is the distance between the left and right wheels. In this document, it is assumed that this distance is the same for the front and rear axles.
- wheel diameter (d):
- center of mass (C): position of the center of mass of the robot, which is considered the point of rotation of the robot.

The inner and outer front wheels have different steering angles to avoid slipping (δ_i and δ_o respectively), and the resulting steering angle for the whole robot (taken at the center of mass) can be computed as:

$$cot(\delta) = \frac{cot(\delta_o) + cot(\delta_i)}{2}. \tag{1}$$

Also, the rear wheels have different angular speeds when the robot is turning (ω_i for the inner wheel and ω_o for the outer one), and the equivalent translational speed for the whole robot (taken at the center of mass) can be computed as:

$$v_t = d\pi \frac{w_o + w_i}{2}. \tag{2}$$

In general, a single actuator is used for each of the traction and steering functions, and therefore the difference in the steering angle between the front wheels, and also the difference in angular speed for the rear wheels, is handled mechanically by the steering mechanism and the differential respectively. This is the case with the robot shown in Fig. 1.

The position of an object in a two dimensional space is completely defined by the position of its center of mass (x, y) and its orientation θ, and a trajectory in a two dimensional space is given by a temporal sequence of these variables. Therefore, in order to estimate the trajectory the robot will follow given the translational speed and the steering angle, first it is necessary to compute the turn radius of the center of mass:

$$R = \sqrt{a^2 + fracl^2 tan^2(\delta)}. \tag{3}$$

The distance traveled by a vehicle in a given amount of time Δt for a given translational speed v_t can be computed as the circular arc. However, the angle of the circular sector is unknown a priori, so the distance is approximated by:

$$L = v_t \Delta\, t. \tag{4}$$

The error of this approximation increases with the speed of the robot and the interval of time considered. However, for small periods and relatively low speeds, as is the case with the robot shown in Fig. 1, this error can be ignored.

Then, the change in the robot's orientation and its relative displacement at the i-th iteration, and also its absolute pose, can be computed as:

$$\Delta\, \theta_i = L_i / R_i, \theta_i = \sum_i \Delta\, \theta_i \tag{5}$$

$$\Delta\, x_i = L_i cos(\delta_i), x_i = \sum_i \Delta\, x_i \tag{6}$$

$$\Delta\, y_i = L_i sin(\delta_i), y_i = \sum_i \Delta\, y_i. \tag{7}$$

With the parameters presented in Table 1, the minimum turn radius for the robot shown in Fig. 1 is $R_{min} = 3.5\, m$.

3 Global planner: search based planner

To generate a feasible path to go from the current position to a desired goal the kinematic constraints of the robot have to be taken into account. Most current global planners use discrete representations of the robot state which reduces the computational complexity at the expense of reducing completeness. However, this discrete representations make it difficult to comply with the maneuverability limitations of most robots.

The state of the robot is encoded by its position (x,y) and its heading (θ) to ensure the generated trajectory is smooth (without sudden changes in the robot's heading). However, the velocity and acceleration of these parameters are not considered.

Some algorithms have been developed to overcome this problem. In particular, a search based algorithm, using a state lattice ([3]), is used for the global planning of the car-like robot presented in section 1. In this case, the non-holonomic planning query is formulated as search in a graph called *state lattice*.

The nodes of the state lattice are a discretized set of all reachable configurations of the robot (both in position and heading) and its edges are feasible motions that connect two possible configurations. In this case not all adjacent nodes (configurations) will be connected, only the nodes whose position and heading coincide with the initial and final positions and headings of a kinematicaly complying trajectory will be connected.

By regularly sampling the state space it is possible to use a reduced set of position invariant feasible trajectories (called motion primitives) that can be used for any node in the state lattice. This set of motion primitives is called the control set. This fact allows us to build the trajectories between adjacent nodes off-line and get the graph connectivity.

Then, for each planning query, a partial graph is built on-line to try to find a path connecting the current and goal configurations. When the solution path in the graph is found, the actual trajectory is built by appending the feasible motions associated to each of the edges of the path.

The number of primitives used for the control set directly affects the computational complexity of the graph search algorithm, so there exist a trade-off between keeping the computational complexity low while exploiting the maneuverability of the robot as much as possible. Each motion primitive is assigned a cost that will be taken into account when searching for the best path.

3.1 Generation of the motion primitives

To generate the motion primitives, first the state space must be discretized, both in position and heading. By doing so, the desired goal state may not be reachable, but in this case the closest state will be selected. All the turn motion primitives will change the heading state of the robot only in one discretized step, while the change in the position states is not limited.

Each motion primitive is build from three separate segments as shown in fig. 3a. One straight segment with variable length l_1, an arc segment with a variable radius R and a second straight segment also with variable length l_2. Fig. 3b shows the possible final states of the robot for a given turn radius and final heading, with $l_1 = 0$ and $l_2 = \infty$.

Eqs. 8 and 9 state how the robot initial and final states (x_i,y_i,θ_i and x_f,y_f,θ_f respectively) are related to the trajectory parameters (l_1, l_2 and R):

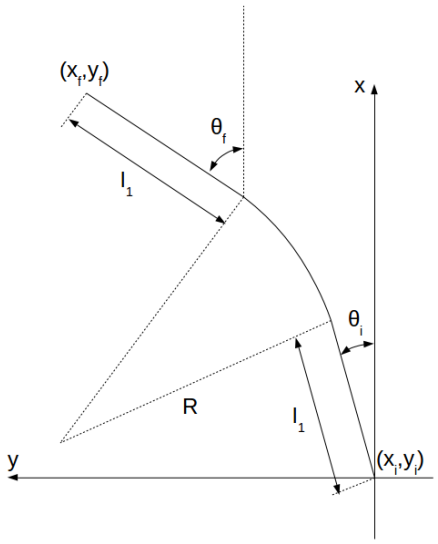
$$x_f = x_i + l_1 cos(\theta_i) + R(sin(\theta_f) - sin(\theta_i)) + l_2 cos(\theta_f), \tag{8}$$

$$y_f = y_i + l_1 sin(\theta_i) - R(cos(\theta_f) - cos(\theta_i)) + l_2 sin(\theta_f). \tag{9}$$

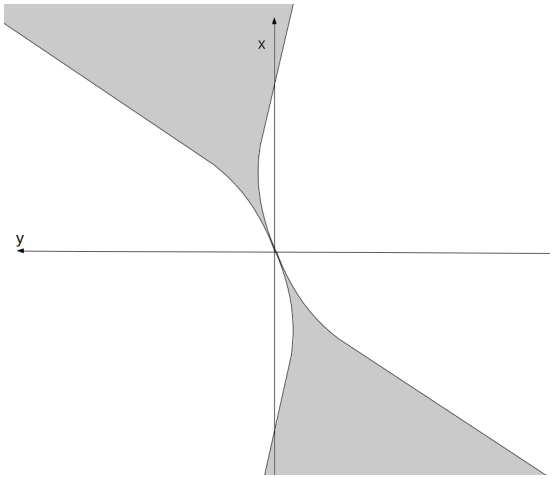
Since the motion primitives are position invariant, from now on we will take $x_i = 0$ and $y_i = 0$. Depending on whether the turn is to the left or the right and whether the motion is forward or backward, the allowed ranges of values for the trajectory parameters (l_1, l_2 and R) change.

Therefore, if the final position of the robot is to the left of the original position (left turn), the turn radius R in Eqs. 8 and 9 will be positive, and vice versa:

$$R_{min} \leq R \leq \infty, \; when\, left\, turn$$
$$-\infty \leq R \leq -R_{min}, \; when\, right\, turn$$



(a) Trajectory template used to generate the motion primitives, build from two straight segments (l_1 and l_2) connected by a circular segment (R).



(b) Possible final states of the robot (grayed) for a given values of initial (θ_i) and final heading (θ_f) with $l_1 = 0$ and $l_2 = \infty$.

Figure 3

where R_{min} is the minimum allowed turn radius, which must be always greater or equal than the minimum mechanical achievable turn radius.

Similarly, if the final position is behind the original one (backward motion), the lengths of the straight segments (l_1 and l_2) in Eqs. 8 and 9 will be negative, and vice versa:

$$0 \leq l_1, l_2 \leq \infty, \; when\, forward\, motion$$
$$-\infty \leq l_1, l_2 \leq 0, \; when\, backward\, motion$$

The lower bound on the length of the straight segments for a forward motion (upper bound for a backward motion) must be always taken into account to ensure smooth paths, but there is no constraint on the other bound. By limiting the maximum (minimum) segment length, the solution space shown in Fig. 3b is reduced.

So, the problem of generating the motion primitives can be stated as finding feasible sets of l_1, l_2 and R for each of the desired final states of the robot, which in turn can be stated as a linear programming problem with equality and inequality constraints. The cost function used for the optimization process tries to minimize (maximize) only the length of both straight segments, the radius is not taken into account.

A set of MATLAB scripts, listed in Appendix A, are provided to automatically generate the motion primitives, plot them on screen to check them out and also save them into a text file with the appropriate format to be used in the *sbpl_lattice_planner* ROS node, presented later in section 5. The main function call to generate the primitives is:

```
p=generate_primitives(resolution,num_angles,points,costs,Rmin);
```

The input arguments to this function are:

- resolution (m): is the minimum distance between to adjacent states both in x and y in the state lattice. The default value is $0.2\,m$.

- num_angles (integer): is the number of discrete heading states of the state lattice. The default value is 16 which corresponds to a minimum heading increment of 0.3927 *rad*. It appears that the *sbpl_lattice_planner* ROS node only supports this value.
- points (vector): a vector with all the desired final configurations (x,y and θ) with an initial heading. The format of each element of this vector must be:

```
points(i,:) = [desired_final_x(m) desired_final_y(m) desired_final_θ(rad)];
```

  Each final configuration is internally discretized using the *resolution* and *num_angles* parameters to map it to the state lattice.

  All configurations in this vector are rotated to each of the possible headings of the state lattice (determined by the *num_angles* parameter) in order to generate all the motion primitives. There is no constraint in the number of elements in this vector, but keep in mind that the total number of primitives that will be generated is proportional to the size of this vector and the number of discrete heading states.

- costs (vector): a vector with the cost associated to each one of the final configurations in *points*. The size of this vector must be equal to the number of final configurations provided in *points*, and all values must be positive integers.
- Rmin (m): is the minimum desired turn radius in meters. This value is internally used by the optimization process to find feasible trajectory parameters to reach each of the final configurations.

The *generate_primitives* MATLAB script returns a structure with all the generated primitives and also additional information necessary to generate the output file to be used in the *sbpl_lattice_planner* ROS node. The format of this structure is:

- resolution (m): is the minimum distance between to adjacent states both in x and y in the state lattice. Its value coincides with the one provided as an input argument.
- num_angles (integer): is the number of discrete heading states of the state lattice. Its value coincides with the one provided as an input argument.
- num_prim (integer): is the number of primitives for each possible heading in the state space. Its value coincide with the size of the *points* input vector.
- num_samples (integer): is the number of points for each trajectory segment. This value is set to 32 by default, and can only be changed modifying the scripts.
- trajectories (structure): a structure with the particular information for each of the motion primitives that have been generated. The size of this structure is *num_prim* by *num_angles*, and may contain empty elements in case that one or more of the desired final configurations are not reachable. In this case an error is reported on screen, but the process continues with the next one.

  The parameters of this structure are:

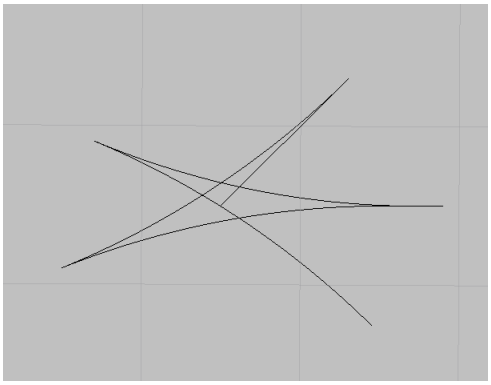
  - start_angle (integer): is the index of the discretized initial heading. Its value is an integer between 0 and $num\_angles - 1$.
  - id (integer): is an identifier of the primitive. Its value is an integer between 0 and $num\_prim - 1$.

- endpose (vector): is the final configuration of the robot in the discretized state lattice, that is in units of the spatial and angular resolutions.
- points (vector): a vector of size *num_samples* with all the robots states to reach the final configuration from the initial one. The values in this vector are meters and radians, and are the ones used to generate the final path the robot should follow.
- cost (integer): the cost associated with the motion primitive. This value is the one provided in the *costs* input vector.

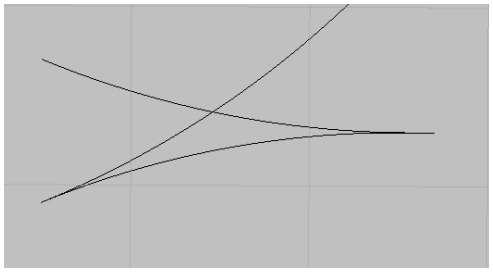
This structure can be used in combination with the *save_primitives* MATLAB script to generate the text file with all the necessary information for the *sbpl_lattice_planner* ROS node. The filename provided to this function will be used as it is, and will overwrite any existing file with the same name in the execution folder.

3.2 Trajectory splitting

The global path returned by the global planner may contain sudden changes in the heading of the robot, which correspond to maneuvers introduced by the kinematic constraints of the robot. Fig. 4 show a few examples of such maneuvers.



(a) Example of a trajectory to turn 90 degrees almost in place.



(b) Example of a trajectory to change the heading more than the maximum turn angle of the robot would allow.

Figure 4: Example of some trajectories with maneuvers that must be executed in order to reach the desired goal position

This maneuvers are critical to reaching the goal, so they must be executed. Standard local planners will try to reach the farthest point within their planning window, which may skip some of these maneuvers. To avoid that, the trajectory is split in segments between maneuver points, so that the robot is forced to execute each of the maneuvers.

To detect this maneuver points, the slope of the trajectory is computed at each point, and when a change of more than $\frac{\pi}{2}$ is detected, the trajectory is split. Therefore, the original path generated by the global planner presented in section 3 is divided in several smaller segments between maneuver points, and are those segments (as partial goals) that are sent to the local planner.

4 Local planner: modified DWA

The path returned by the global planner has been generated only taking into account a static version of the environment (walls, trees, trash cans, etc.), but the robot, in general, will have to navigate in a dynamic environment with people and other obstacle moving around. The task of

the local planner is to avoid collisions with all this dynamic obstacles while trying to follow as much as possible the path lay out by the global planner.

One of the most commonly used local planner is the Dynamic Window Approach (DWA) [1]. This algorithm takes into account the dynamic limitations of the robot, in terms of velocities and accelerations, to compute the next control command for a given time interval. Only those commands that will be achievable in the desired time interval and that will allow the robot to stop safely will be considered.

From all the remaining feasible commands, the one maximizing an objective function is chosen. This objective function includes a measure of the progress towards the goal, the distance to the obstacles and the forward velocity among other parameters.

The search for feasible commands is carried out in the state space, which is determined by the motion control variables of the robot (translational and rotational speeds in general). However, the control variables for an ackermann based robot are the translational speed and the steering angle, and therefore, the original algorithm has to be modified in order to adapt it to the desired robot architecture.

In the next few sections, the changes and additions to the original Dynamic Window Approach algorithm are introduced. Section 4.1 describes how to find the window of feasible motion commands in state space, and section 4.2 describes how to generate a set of trajectories within this window. Finally, section 4.2.1 introduces a new cost function which is specially useful for the ackermann configuration.

4.1 Window generation

The first step is to find out which are the maximum translational velocity and steering angle (i.e. the dynamic window) that can be achievable in the given time interval, taking into account that the final translational and steering velocities must be 0. After finding the boundaries of the dynamic window, a finite number of samples will be taken in each of the two dimensions of the window in order to generate all the trajectory candidates (see section 4.2).

Given that the number of samples is finite, and in general small in order to reduce the overall computational complexity of the algorithm, it makes no sense to use a fixed time interval to compute the dynamic window because, when the robot is close to a goal, only a small subset of all the computed trajectories will be feasible to reach the goal (those with small translational speeds).

Therefore, a variable time interval is used in terms of the distance to the goal d and the current translational speed v_t of the robot, as shown in Eq. 10. Maximum and minimum values for this time interval can also be configured depending on the application.

$$T_{sim} = \frac{d}{v_t} \tag{10}$$

To compute the boundaries of the dynamic window, the current state of the robot, in terms of current translational velocity and current steering angle and velocity, is needed. Also the dynamic parameters of the robot are needed, that is the maximum translational acceleration and deceleration and the maximum steering velocity, acceleration and deceleration.

Fig. 5 shows some examples of possible velocity profiles for the translational velocity, where the maximum and minimum velocities are set to $v_{max} = 10m/s$ and $v_{min} = -10m/s$ respectively, and the acceleration and deceleration are both set to $acc_{max} = 5\, m/s^2$. The blue and red traces are computed for a time interval of $T_{sim} = 10\, s$, and the yellow and purple ones are computed for a time interval of $T_{sim} = 4\, s$. The initial speed in both cases is set to $v_i = -5\, m/s$.

If the time required to accelerate to the maximum (minimum) velocity T_{acc} and to decelerate to a complete stop T_{dec} are smaller than the desired time internal, the boundaries of the dynamic

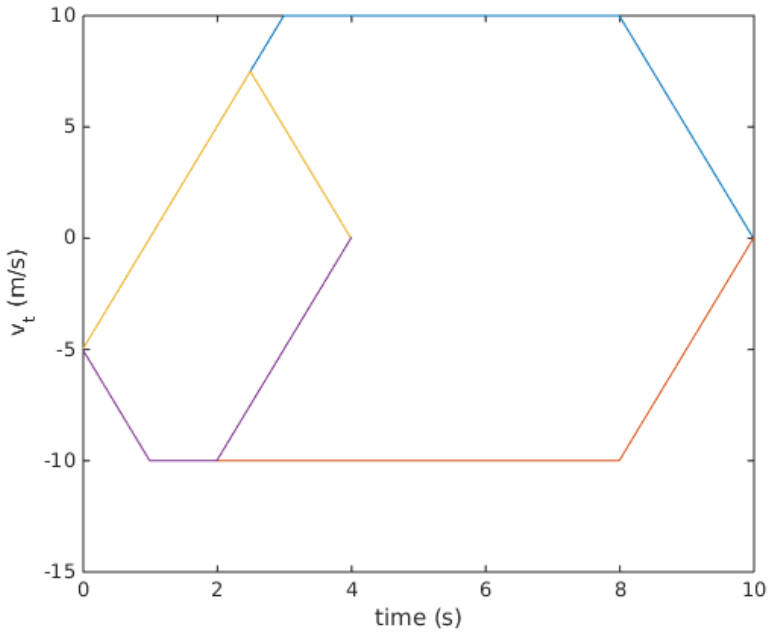


Figure 5: Example of some velocity profiles used to find the boundaries of the dynamic window for the translational velocity.

window are the maximum and minimum velocity (red, blue and purple traces on Fig. 5). However, if the given time interval is not enough to accelerate to the maximum (minimum) velocity, one or both of the boundaries of the dynamic window must be reduced (yellow trace in Fig. 5).

Eqs. 11 and 12 can be used to compute the maximum and minimum velocities respectively.

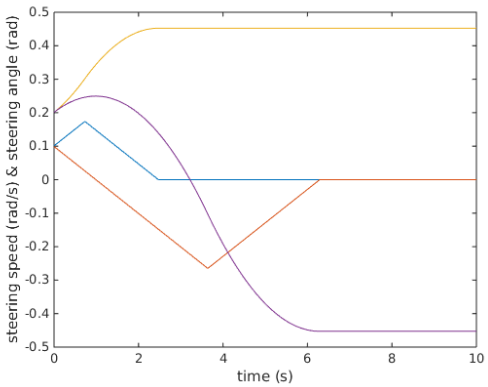
$$v_{max}^{dwa} = \begin{cases} \frac{T_{sim} acc_{max}}{2} + \frac{v_i}{2} & \text{if } T_{acc} + T_{dec} \geq T_{sim} \\ v_{max} & \text{if } T_{acc} + T_{dec} < T_{sim} \end{cases}, \tag{11}$$

$$v_{min}^{dwa} = \begin{cases} -\frac{T_{sim} acc_{max}}{2} + \frac{v_i}{2} & \text{if } T_{acc} + T_{dec} \geq T_{sim} \\ v_{min} & \text{if } T_{acc} + T_{dec} < T_{sim} \end{cases}. \tag{12}$$

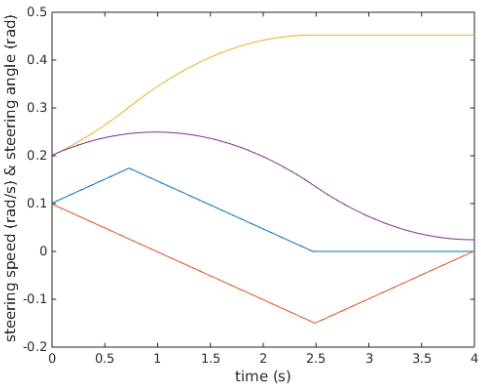
For the steering angle window boundaries, a similar procedure is followed. In this case, it is somewhat more complex to compute the boundaries because we have to deal with both angles and velocities. Figs. 6a and 6b show both the velocity profiles and steered angles for two different time intervals ($T_{sim} = 10s$ and $T_{sim} = 4s$ respectively) for identical initial conditions, $\delta_i = 0.2rad$ and $\dot{\delta}_i = 0.1\ rad/s$. In these examples, the maximum steering speed is $\dot{\delta}_{max} = 0.5\ rad/s$ (which is never reached), and the maximum steering angle is $\delta_{max} = 0.45\ rad$.

First, the boundaries for the steering speed are computed using the same procedure used for the translational speed (Eqs. 11 and 12), and a preliminary velocity profile is generated. If the final steered angles with these preliminary velocity profiles are within the physical limitations of the robots, these angles are used as boundaries of the dynamic window (See the red velocity profile of Fig. 6b and its corresponding steering angle in purple).

In the case that the steered angles go beyond the physical limitations, the steering speed is further reduced and the physical limits of the robot are taken as the boundaries of the dynamic window (See the red and blue velocity profiles in Fig. 6a and their corresponding steering angles in purple and yellow respectively). Note that the velocity profiles in figs 6a and 6b are only used



(a) Example of a dynamic window equivalent to the maximum steering range of the robot, $\Delta\delta = 2\delta_{max}$



(b) Example of a dynamic window considerably reduced due to the time interval used. All other parameters are the same.

Figure 6: Two examples of dynamic windows for the steering angles for two different time intervals with identical initial conditions $\delta_i = 0.2\,rad$ and $\dot{\delta}_i = 0.1\,rad/s$. The velocity profiles are also shown in blue and purple for the upper and lower boundaries respectively.

for finding the boundaries of the dynamic window, but they do not represent any control action applied to the steering wheels.

Both the initial steering velocity $\dot{\delta}_i$ and the initial steering angle δ_i are taken into account when computing the dynamic window boundaries, which provides a more accurate estimation of the dynamic window.

4.2 Trajectory generation

Once the boundaries of the dynamic window have been computed as explained in section 4.1, it is time to generate a set of trajectory candidates to be evaluated.

The boundaries define a two dimensional subspace with all the feasible values of translational speed and steering angle. Since it is not computationally feasible to evaluate all the possible candidate pairs, an uniform sampling is performed in both dimensions, and a reduced set of candidate pair is generated.

For each pair of steering angle and translational speeds, the resulting trajectory is generated for the desired time interval using the kinematic and dynamic constraints of the robot. Each trajectory is then evaluated with a set of cost functions. The usual costs functions used to evaluate each trajectory are:

- **oscillation**: This cost function penalizes trajectories that would change the motion direction in order to avoid oscillations.
- **obstacles**: This cost function eliminates the trajectories that would collide with an obstacle, either an static one from the map or a dynamic one detected by the sensors.
- **path**: This cost function evaluates the trajectory in terms of how close it is to the planned path.
- **goal**: This cost function evaluates the trajectory in terms of how close the final position reached by the trajectory is to the global (or local) goal.

Each cost function assign a cost to the trajectory, and the total cost assigned to it is the weighted sum of all these costs. Some of the cost functions may discard the trajectory without assigning any cost (trajectories that would collide with an obstacle for example).

After all the candidate trajectories have been evaluated, the one with the lowest cost value is selected to be executed on the robot for the current iteration. The whole process is repeated for each control iteration until the robot reaches its target position or no feasible trajectory can be found to continue.

4.2.1 Heading cost function

Due to the motion limitations introduced by the kinematic constraints of an ackermann based robot, it is useful to introduce a new cost function to be evaluated. This cost function compares the heading of the robot in several points along the candidate trajectory with the heading of the desired path, and assigns a cost proportional to the angular difference in all evaluated points (the greater the error, the greater the cost). This cost function is intended to minimize the heading error of the robot along the path, and therefore minimize the need of re-planning required.

In general the number of points in the global path segment and the number of points in the candidate trajectory do not coincide, because the former is fixed by the user when generating the motion primitives (see section 3.1), and the later depends on the chosen resolution. For this reason, for each evaluation point in the candidate trajectory, it is necessary to find the closest point in the current global path segment.

Once the two closest points are found, a vector representing the slope of each curve is generated by using the current and the previous points, v_{seg} for the global path segment and v_{traj} for the candidate trajectory. With theses two vectors, the heading difference for a single point is computed as shown in Eq. 13:

$$\Delta\theta = atan2\left(\frac{v_{seg} \times v_{traj}}{v_{seg} \cdot v_{traj}}\right). \tag{13}$$

The heading differences at all evaluation points are accumulated and then multiplied by a scale factor.

5 ROS integration

This section summarizes the work done to integrate the global and local planner introduced in sections 3 and 4 respectively into the ROS middleware framework used at IRI. First a general overview of the navigation framework used in ROS is presented in section 5.1, and then section 5.2 deals with the global planner and section 5.3 with the local planner.

The software is publicly available through the SVN server at IRI:

> https://devel.iri.upc.edu/pub/labrobotica/ros/iri-ros-pkg_hydro/metapackages/iri_navigation/iri_ackermann_local_planner

and can be used together with the developed car simulator:

> https://devel.iri.upc.edu/pub/labrobotica/ros/iri-ros-pkg_hydro/metapackages/car_robot

in order to test the navigation framework proposed in this document.

5.1 ROS navigation framework

The ROS navigation framework is built around a simple package named *move_ base*. This package implements the basic sequence of events necessary to navigate the robot from an arbitrary initial position, through a dynamic environment, until it reaches the desired goal position, or the action is canceled because the target position is unreachable.

Rather than integrating both the local and global planners into this package, which will require to replicate the basic code for each combination of local and global planners, this package uses the concept of *plug-in*, which allows it to use any number of local and global planner that comply with a simple generic interface, as shown in Fig. 7.

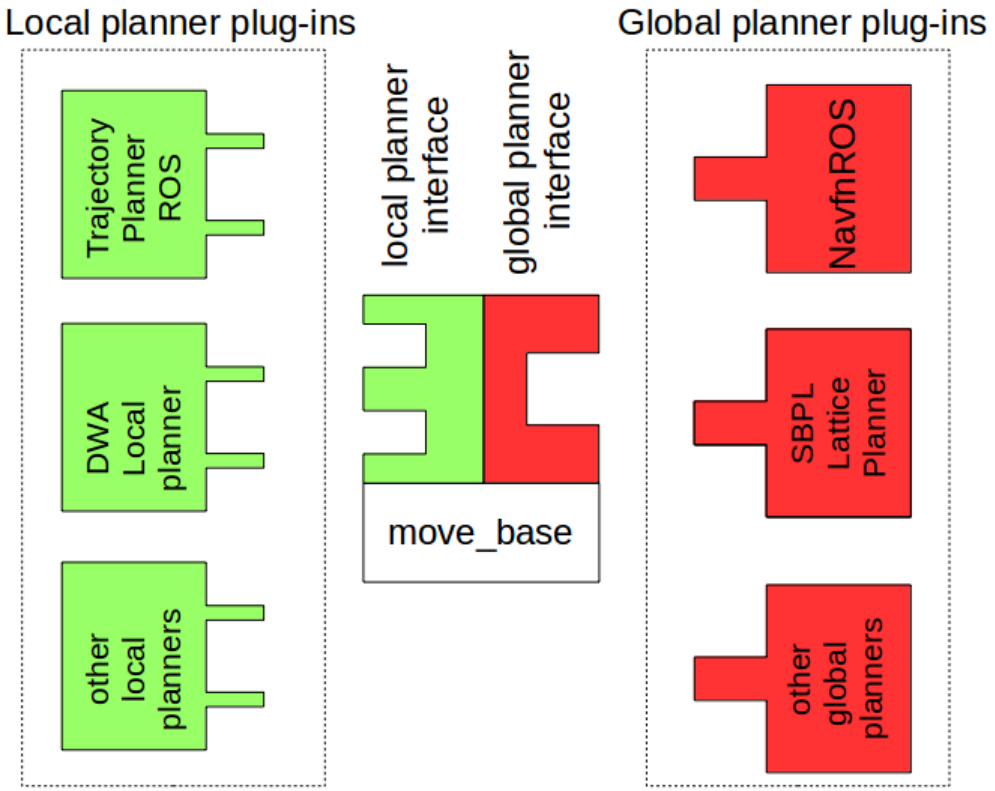


Figure 7: Simplified structure of the move_base package with the global and local planners *plug-in* interfaces.

By using *plug-ins*, the task of testing different planners in similar set-ups is greatly simplified, and also allows the user to customize them to their particular needs. The necessary interface for the global planner is listed here (see the BaseGlobalPlanner class in the nav_core package for more details):

- **initialize**: This function is called at construction time to initialize all the necessary parameters of the global planner. It returns either false or true depending on whether the initialization failed or not respectively.

- **makePlan**: This function is called each time a new navigation goal is received. Given the current position and the desired target this function should return a plan. Alternatively, this function can also return a cost associated to the generated plan. This function returns

either true if the path have been generated successfully, or false if it was impossible to find a feasible path.

The necessary interface for the local planner is listed here (see the BaseLocalPlanner class in the nav_core package for more details):

- **computeVelocityCommands**: This function is periodically called to get a new velocity command for the robot. This function returns true if a feasible motion command has been found and false otherwise.
- **initialize**: This function is called at construction time to initialize all the necessary parameters of the local planner. It returns either false or true depending on whether the initialization failed or not respectively.
- **isGoalReached**: This function is periodically called to check whether the target position has been reached (it returns true) or not (it returns false).
- **setPlan**: This function is called once for each new navigation target or when re-planning is necessary, immediately after the makePlan function of the global planner returns a valid plan. This function return true or false depending on whether the new global plan could be set properly or not, respectively.

The *move_base* package executes the simple state machine shown in Fig. 8. At the Initialize/idle state, the corresponding initialize functions of the global and local planner are called. If no initialization error is reported, the *move_base* package waits for a new navigation request.

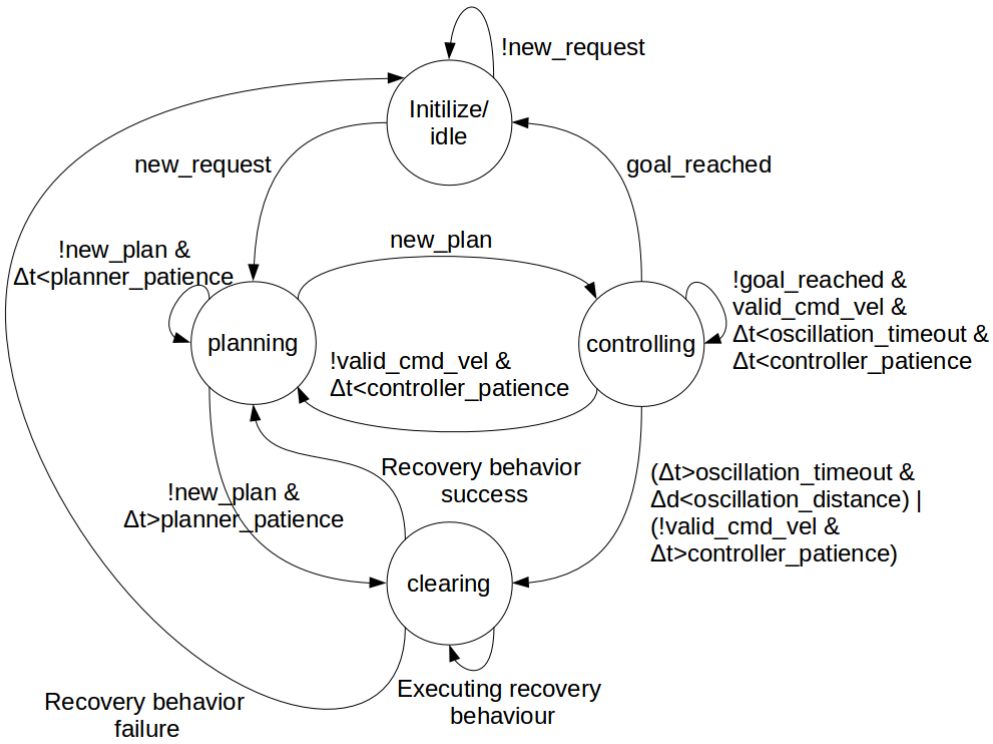


Figure 8: Simplified state machine executed by the move_base package.

When a new navigation request is received, a new plan is requested to the global planner through the *makePlan* function in the *planning* state. If a new plan is not returned within a

specified amount of time (*planner_patience*), the planning is aborted, and the state changes to *clearing*, where the robot will try to execute any recovery behavior available in order to clear the internal costmap.

If any the recovery behaviors have been successful in clearing the costmap of old obstacles, the state returns to *planning* in order to try to find a path to the desired target. Otherwise, the state changes to *idle* and the navigation request is aborted.

On the other hand, if a plan is found in time, the state changes to *controlling* and the *setPlan* function is called to load the global plan into the local planner. In this state, the *isGoalReached* and *computeVelocityCommands* are called at each iteration to detect whether the robot has arrived to the target position or not, and to compute a new motion command if not.

In the *controlling* state, if a new valid motion command can not be found, the state changes to *planning* to try to find an alternative plan to reach the goal. If the impossibility of finding a valid motion command persists for a given amount of time (*controller_patience*) the state changes to the *clearing* state to clear the cost map.

Finally, when the goal is reached, the state returns to *idle* where the navigation action is successfully ended top notify the user that the goal has been reached. At any point, if a new navigation request is received, the current action is canceled, and the process starts over with the new goal.

5.2 Global planner

As introduced before in section 3, the global planner used for the car-like robot is a lattice planner developed by [3] which has already been integrated into ros (*sbpl_lattice_planner* ROS node). Therefore, the only necessary step has been to generate a compatible motion primitives file which take into account the kinematic constraints of the robot.

The trajectory splitting presented in section 3.2, although being part of the global planner, it has been implemented inside the local planner for simplicity, as will be presented in the next section.

As of ROS Groovy, the *sbpl_lattice_planner* ROS package is no longer maintained by ROS, and its correct operation in newer versions of the ROS framework depends on a branch of the original repository created by Johannes Meyer [2].

5.3 Local planner

Integrating the local planner into ROS has been more difficult, because it has been necessary to develop a new *plug-in* compatible with the local planner interface of the move_base package. To develop this new plug-in we used the *dwa_local_planner* as a starting point, and made the necessary changes and additions to adapt it to the ackermann configuration.

The *dwa_local_planner* uses the modular structure shown in Fig. 9, which in turn uses several standard modules provided by the *base_local_planner* package.

Given the relatively high computational cost of generating a global plan using the search based algorithm introduced in section 3, it is not feasible to search for a new global plan the first time a valid motion command can not be found, as shown in Fig. 8. Therefore, the behavior of this state machine is slightly modified so that a new global plan is generated only when no valid motion command has been found for several iterations. The number of iterations the algorithm will wait can be configured using the *controller_patience* parameter.

Also, all the parameters of the kinematic and dynamic constraints of and ackermann based robot can be changed, so that it would be easy to adapt it to several different implementations. The kinematic parameters are *axis_distance*, *wheel_distance* and *wheel_radius*, and the dynamic parameters are *max_trans_vel*, *min_trans_vel*, *max_trans_acc*, *max_steer_angle*, *min_steer_angle*, *max_steer_vel*, *min_steer_vel* and *max_steer_acc*.

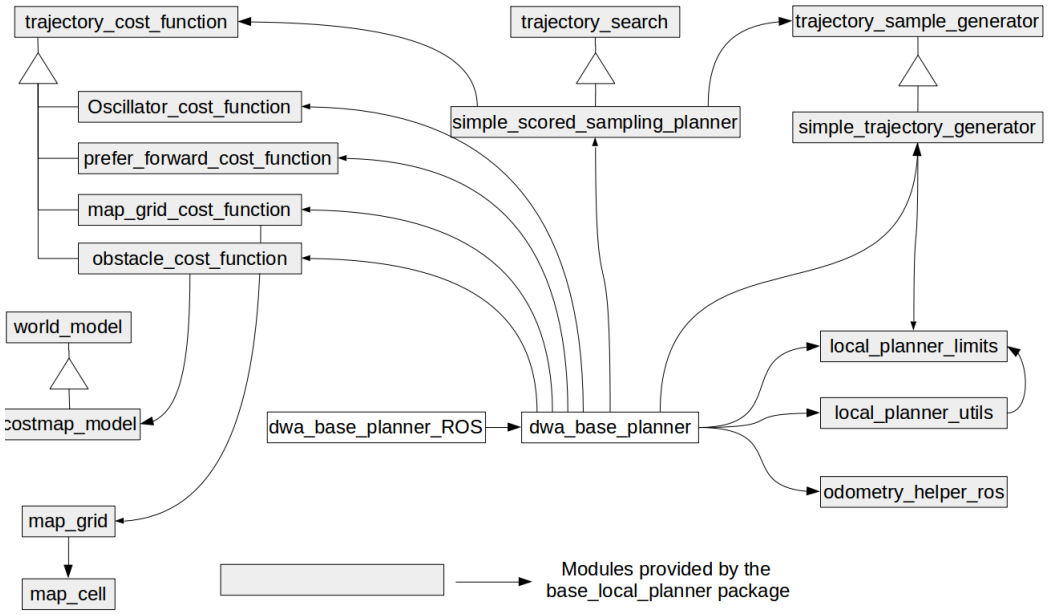


Figure 9: Software structure of the *dwa_local_planner* ROS package with its most relevant modules.

The main changes to the original *dwa_local_planner* are described in the next few sections.

Trajectory splitting

When the *setPlan* function of the local planner interface is called, the original input plan is split as explained in section 3.2 and stored internally as local goals. The fact that a single global goal can be split in several local goals make it necessary to change a little bit the behavior of the local planner interface, as shown in the flow chart in Fig. 10.

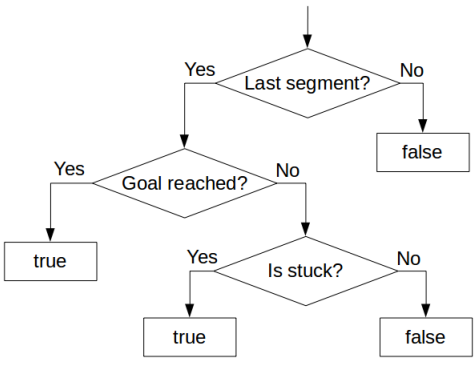


Figure 10: Flow chart of the *isGoalReached* function taking into account the trajectory splitting and the possibility that the robot get stuck before reaching the goal.

In this case, when the *isGoalReached* function of the local planner interface is called, it first checks whether the segment that is being executed is the last one or not. In the case that the robot is not executing the last segment of the whole global plan, this function will always return false. On the other hand, the generic function provided by the *base_local_planner* is used to

check whether the robot has reached the final local goal, and therefore the final target position. If so, true is returned and the whole navigation action will finish successfully.

However, because the accumulation of errors in the execution of each of the segments or sudden changes in position and orientation forced by a localization system, the local or global goal may never be reached, because the kinematic constraints of an ackermann based robot make it impossible to further reduce the position and orientation errors. In this case the robot is *stuck* (see Fig. 11 for an example of this situation).

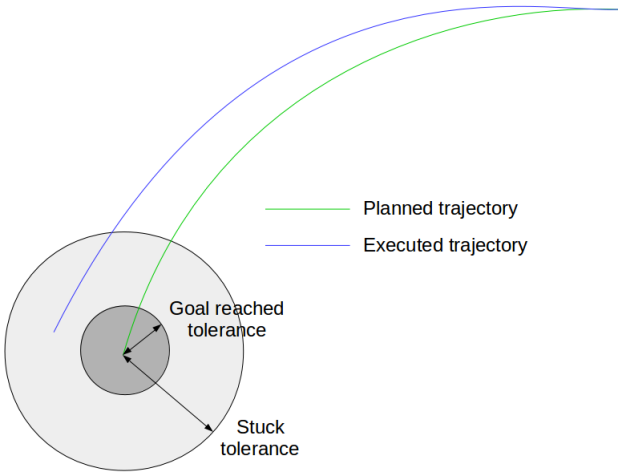


Figure 11: An example of a situation in which the robot would get stuck trying to complete a trajectory segment due to an initial orientation error.

To solve this problem, a *stuck* detector has been added. This detector checks the translational velocity of the robot and the distance to the desired goal. If the translational speed is either small or it changes in direction continuously, and the distance to the goal is bigger than the goal reached tolerance but smaller than a predefined value, the robot is considered to be stuck.

In this case, as shown in Fig. 10, the *isGoalReached* function will either return true if it is the last segment, or jump to the next trajectory segment, even though the real goal reached condition is not satisfied, which will allow the navigation framework to continue with its normal operation.

Motion commands and odometry

For holonomic and quasi-holonomic robots, the motion commands are defined by the velocities in the relative x and y axis and the turn rate, and the corresponding fields in the *Twist* message of the *cmd_vel* topic are filled. However, as seen in section 2, the motion commands for an ackermann-based robot must include the translational speed, the steering angle and, optionally, the steering speed.

The best solution would be to generate a new topic message specific for car-like robots and publish it instead of the standard *Twist* message, but this would require to modify the ROS navigation framework. A simpler solution has been chosen, in which some of the fields of the Twist message are overloaded to carry the ackermann specific information. Table 2 shows the overloaded *Twist* message.

The information returned by the *Odometry* message does not directly represent the state of an ackermann based robot. It is possible to use this information to estimate the real state of the robot, but the performance of the planner may be degraded due to errors in the state estimation. Therefore, some of its fields are overloaded to carry ackermann specific information. In this case

Table 2: Overloaded fields of the *Twist* message for *cmd_ vel* topic of an ackermann based robot.

Twist message		
linear	x	translational speed
	y	always 0
	z	not used
angular	x	not used
	y	not used
	z	steering angle

however, only message fields normally not used by wheeled mobile robots are overloaded because the odometry information may be used by other ROS nodes.

Table 3 shows the overloaded *Twist* fields inside the *Odometry* message. All other fields of the *Odometry* message are not modified.

Table 3: Overloaded fields of the *Twist* field inside the *Odometry message* for ackermann based robot.

Twist inside Odometry message		
linear	x	unchanged
	y	unchanged
	z	translational speed
angular	x	steering angle
	y	steering speed
	z	unchanged

Trajectory sample generator

This class inherits from the *TrajectorySampleGenerator* class provided by the *base_ local_ planner* ROS node, and performs the following actions:

- find the best time interval for the current iteration in terms of the distance to the global or local goal and the maximum translational speed of the robot as explained in section 4.1. The computed time interval is limited by a maximum and minimum values specified as configuration parameters. See parameters *max_ sim_ time* and *min_ sim_ time*.
- find the boundaries of the dynamic window at each iteration taking into account the current state of the robot (translational speed and steering angle and speed) as explained in section 4.1.
- given the number of samples for both the translational speed and steering angle and the boundaries of the dynamic windows, generate the actual trajectory candidates for each pair of the control parameters in order to evaluate all of them using the cost functions, as explained in section 4.2.

Cost functions

A new cost function has been added, that inherits from the *TrajectoryCostFunction* class provided by the *base_ local_ planner* ROS node, and implements the trajectory evaluation procedure introduced in section 4.2. The number of points to be evaluated and the final scale factor of this

cost function can be changed by the user by the *heading_points* and *hdiff_scale* configuration parameters.

The number of points to evaluate must be chosen carefully because of its relatively high computational cost, and the fact that it will be called for each trajectory candidate at each iteration.

Configuration parameters

This section summarizes and provides a brief description of the configuration parameters specific to an ackermann based robots, but the standard navigation framework parameters are not listed here.

- **max_trans_vel** (double, default: $0.3\,m/s$): the maximum allowed forward speed of the robot in m/s.
- **min_trans_vel** (double, default: $-0.3\,m/s$): the maximum allowed backward speed of the robot in m/s.
- **max_trans_acc** (double, default: $1.0\,m/s^2$): both the maximum translational acceleration and deceleration the robot is capable in m/s^2.
- **max_steer_angle** (double, default: $0.45\,rad$): maximum steer angle in the counter-clockwise direction in rad.
- **min_steer_angle** (double, default: $-0.45\,rad$): maximum steer angle in the clockwise direction in rad.
- **max_steer_vel** (double, default: $1.0\,rad/s$): maximum steering speed in rad/s.
- **min_steer_vel** (double, default: $-1.0\,rad/s$): minimum steering speed in rad/s
- **max_steer_acc** (double, default: $0.36\,rad/s^2$): both maximum steering acceleration and deceleration in rad/s^2.
- **axis_distance** (double, default: $1.65\,m$): distance in m between the front and back axles of the robot.
- **wheel_distance** (double, default: $0.3\,m/s$): distance in m between both wheels in the same axle.
- **wheel_radius** (double, default: $0.4329\,m$): diameter of the wheels in m.
- **max_sim_time** (double, default: $10\,s$): maximum allowed time interval for the Dynamic Window Approach algorithm in s.
- **min_sim_time** (double, default: $1.7\,s$): minimum allowed time interval for the Dynamic Window Approach algorithm in s.
- **planner_patience** (int, default: 2): number of iterations the local planner can not find a valid motion command before trying to find a new global plan.
- **hdiff_scale** (double, default: 1.0): scale factor used to weight the heading cost function presented in section 4.2.1.
- **heading_points** (int, default: 8): number of equally spaced points taken on the candidate trajectories to evaluate its heading.

All of these parameters can be set at the beginning by assigning a value in the launch file, and also they all can be changed at any time by using the the dynamic reconfigure feature of ROS.

6 Conclusions

The overall navigation framework for ackermann based robots presented in this technical report has been implemented and tested both in the real robot shown in Fig. 1 and in simulation. The proposed framework is successful in finding a path to reach a desired goal, both in reduced spaces and large open spaces.

However, most of the times the resulting global path is sub-optimal in terms of the number of maneuvers necessary to achieve the goal, specially when the goal position is close to the current position of the robot. The global planner tends to use short segments (made of a single motion primitive) instead of longer segments (made of several similar motion primitives) which would reduce the number of maneuvers required.

It has been shown that the type of motion primitives used and the costs assigned to them play a crucial role in the quality of the resulting global paths, and also that different tasks may require different control sets, that is, the best motion primitives to find paths in reduced spaces may not generate good quality paths for large open spaces, and vice-versa. Therefore, a method to switch the motion primitives depending on the environment conditions may be necessary in a general case.

Regarding the local planner, it is capable of effectively follow the global path executing each of the maneuvers, but its ability to avoid dynamic obstacles is quite limited, and in general the whole framework ends up finding a new path when an unexpected obstacle is found. This problem is mainly caused by the kinematic and dynamic limitations of the robot, but the algorithm could be modified in order to overcome it, for example, by increasing the dynamic window time interval, and thus allowing the local planner more time to react to changing conditions.

The stuck detector, is quite useful in paths with several consecutive maneuvers in order to continue the execution of the path even when the accumulated error is bigger than the allowed tolerance, however it is not perfect, and some times is incapable of properly detecting the stuck condition, mainly when the localization system makes big pose correction.

The computational complexity of the overall framework (global and local planners) is quite high, and prevents it from being used in rapidly changing environments such as roads with moving traffic.

A Global planner MATLAB scripts

This Appendix includes the source code of the MATLAB scripts used to generate and handle the motion primitives for the search based algorithm used the global planner for the car-like robot at IRI. These scripts are included in the *car_rosnav* ROS package that can be downloaded from the SVN server at IRI:

> https://devel.iri.upc.edu/pub/labrobotica/ros/iri-ros-pkg_hydro/metapackages/car_robot/car_rosnav

A.1 Find feasible trajectory parameters

The MATLAB script shown in Listing 1 uses a simple linear optimization method to find the parameters of a trajectory from the current pose of the robot to the desired one, if any exists. See section 3.1 for details on the problem formulation, the equality and inequality constraints used, as well as for the cost function.

Listing 1: MATLAB function to find the trajectory parameters for the current and final robot poses

```
function [l1 R l2 status]=find_traj_params(theta_i,theta_f,delta_x, ...
          delta_y,Rmin)

A_eq=[[cos(theta_i) sin(theta_f)-sin(theta_i) cos(theta_f)]; ...
      [sin(theta_i) -(cos(theta_f)-cos(theta_i)) sin(theta_f)]];
b_eq=[delta_x;delta_y];
A_ineq=zeros(3,3);
b_ineq=zeros(3,1);
f=zeros(3,1);
nx=cos(theta_i);
ny=sin(theta_i);
norm=sqrt(delta_x^2+delta_y^2);
if(nx*delta_x/norm+ny*delta_y/norm>0)
    % positive l1
    A_ineq(1,:)=[-1000 0 0];
    f(1)=1;
else
    % negative l1
    A_ineq(1,:)=[1000 0 0];
    f(1)=-1;
end
if(nx*delta_x/norm+ny*delta_y/norm>0)
    % positive l2
    A_ineq(3,:)=[0 0 -1];
    f(3)=1;
else
    % negative l2
    A_ineq(3,:)=[0 0 1];
    f(3)=-1;
end
```

```
nx=cos(theta_i+pi/2);
ny=sin(theta_i+pi/2);
if(nx*delta_x/norm+ny*delta_y/norm>0)
    % positive R
    A_ineq(2,:)=[0 -1 0];
    b_ineq(2)=-Rmin;
    f(2)=0;
else
    % negative R
    A_ineq(2,:)=[0 1 0];
    b_ineq(2)=-Rmin;
    f(2)=0;
end
[x,fval,exitflag] = linprog(f,A_ineq,b_ineq,A_eq,b_eq);
if exitflag~=1
    warning(['Impossible to find a feasible solution for initial heading '...
        ,num2str(theta_i),', final heading ',num2str(theta_f), ...
        ' and displacement (',num2str(delta_x),',',num2str(delta_y),')']);
    l1=0;
    R=0;
    l2=0;
    status=0;
else
    l1=x(1);
    R=x(2);
    l2=x(3);
    status=1;
end
```

A.2 Generate trajectory points

The MATLAB script shown in Listing 2 generates the actual trajectory points for the corresponding initial and final poses and the trajectory parameters l_1, l_2 and R. See section 3.1 for details of its operation.

Listing 2: MATLAB function to generate the trajectory points from the trajectory parameters

```
function points=generate_traj(l1,R,l2,x_i,y_i,theta_i,x_f,y_f, ...
                              theta_f,numofsamples)

% compute the total length to move
L=abs(l1)+abs(l2)+abs(R*(theta_f-theta_i));
%generate samples
dtheta=theta_i;
length2=0;
points = zeros(numofsamples,3);
for i = 1:numofsamples
    dL = L*(i-1)/(numofsamples-1);
    if(dL < abs(l1))
        if(l1>0)
            points(i,:) = [x_i + dL*cos(theta_i) ...
```

```
                y_i + dL*sin(theta_i) ...
                theta_i];
        else
            points(i,:) = [x_i - dL*cos(theta_i) ...
                y_i - dL*sin(theta_i) ...
                theta_i];
        end
    else
        if(dL<(L-abs(l2)))
            if(theta_i<theta_f)
                dtheta = dtheta+(L/(numofsamples-1))/abs(R);
                points(i,:) = [x_i + l1*cos(theta_i) + R*(sin(dtheta) - ...
                    sin(theta_i)) y_i + l1*sin(theta_i) - R*(cos(dtheta) -...
                    cos(theta_i)) dtheta];
            else
                dtheta = dtheta-(L/(numofsamples-1))/abs(R);
                points(i,:) = [x_i + l1*cos(theta_i) + R*(sin(dtheta) - ...
                    sin(theta_i)) y_i + l1*sin(theta_i) - R*(cos(dtheta) -...
                    cos(theta_i)) dtheta];
            end
        else
            if(l2>0)
                points(i,:) = [x_i + l1*cos(theta_i) + R*(sin(theta_f) - ...
                    sin(theta_i)) + length2*cos(theta_f) y_i + ...
                    l1*sin(theta_i) - R*(cos(theta_f) - cos(theta_i)) + ...
                    length2*sin(theta_f) theta_f];
                length2=length2+(L/(numofsamples-1));
            else
                points(i,:) = [x_i + l1*cos(theta_i) + R*(sin(theta_f) - ...
                    sin(theta_i)) + length2*cos(theta_f) y_i + ...
                    l1*sin(theta_i) - R*(cos(theta_f) - cos(theta_i)) + ...
                    length2*sin(theta_f) theta_f];
                length2=length2-(L/(numofsamples-1));
            end
        end
    end
end

% compute final pose error
errorxy = [x_f - points(numofsamples,1) ...
    y_f - points(numofsamples,2)];
interpfactor = [0:1/(numofsamples-1):1];
points(:,1) = points(:,1) + errorxy(1)*interpfactor';
points(:,2) = points(:,2) + errorxy(2)*interpfactor';
```

A.3 Generate motion primitives

The MATLAB script shown in Listing 3 calls the *find_traj_params* (see Listing 1) and *generate_traj* (see Listing 2) for all the provided final configurations and all the possible orientations, and generates an structure with all feasible motion primitives.

Listing 3: MATLAB function to generate all the motion primitives.

```
function primitives=generate_primitives(resolution,num_angles,points,...
                    costs,Rmin)

primitives.resolution=resolution;
primitives.num_angles=num_angles;
primitives.num_prim=size(points,1);
primitives.num_samples=32;
primitives.trajectories=[];

for angleind = 1:num_angles
    %iterate over primitives
    for primind = 1:size(points,1)
        %current angle
        currentangle = (angleind-1)*2*pi/num_angles;
        primitives.trajectories((angleind-1)*size(points,1)+...
            primind).start_angle=angleind-1;% in discretized states
        primitives.trajectories((angleind-1)*size(points,1)+...
            primind).id=primind-1;

        traj_points(primind,1)=points(primind,1)*cos(currentangle)-...
            points(primind,2)*sin(currentangle);
        traj_points(primind,2)=points(primind,1)*sin(currentangle)+...
            points(primind,2)*cos(currentangle);
        traj_points(primind,3)=points(primind,3)+currentangle;

        % find the closest point on the resolution grid
        res_points(primind,1)=round(traj_points(primind,1)/resolution)*...
            resolution;
        res_points(primind,2)=round(traj_points(primind,2)/resolution)*...
            resolution;
        res_points(primind,3)=traj_points(primind,3);

        primitives.trajectories((angleind-1)*size(points,1)+...
            primind).endpose=zeros(1,3);
        primitives.trajectories((angleind-1)*size(points,1)+...
            primind).endpose(1:2)=round(res_points(primind,1:2)./resolution);
        primitives.trajectories((angleind-1)*size(points,1)+...
            primind).endpose(3)=round(rem(res_points(primind,3)*...
            num_angles/(2*pi),num_angles));
        primitives.trajectories((angleind-1)*size(points,1)+...
            primind).cost=costs(primind);

        if(points(primind,3)==0)% straight segments
            l1=points(primind,1)/2;
            l2=points(primind,1)/2;
            R=0;
            status=1;
        else
```

```
            [l1 R l2 status]=find_traj_params(currentangle,...
                res_points(primind,3),res_points(primind,1),...
                res_points(primind,2),Rmin);
        end
        if(status==1)
            primitives.trajectories((angleind-1)*size(points,1)+...
                primind).points=generate_traj(l1,R,l2,0,0,currentangle,...
                res_points(primind,1),res_points(primind,2),...
                res_points(primind,3),primitives.num_samples);

            plot(primitives.trajectories((angleind-1)*size(points,1)+...
                primind).points(:,2),primitives.trajectories((angleind-1)*...
                size(points,1)+primind).points(:,1));
            hold on;
        else
            primitives.trajectories((angleind-1)*size(points,1)+...
                primind).points=[];
        end
    end
end
```

A.4 Save motion primitives

The MATLAB script shown in Listing 4 saves the motion primitives into an output file compatible with the *sbpl_lattice_planner* ROS node once the primitives have been successfully generated.

Listing 4: MATLAB function to save the motion primitives into a file.

```
function save_primitives(output_filename,primitives)

file = fopen(output_filename, 'w');

fprintf(file, 'resolution_m: %f\n', primitives.resolution);
fprintf(file, 'numberofangles: %d\n', primitives.num_angles);
fprintf(file, 'totalnumberofprimitives: %d\n', ...
        primitives.num_prim*primitives.num_angles);

for primind = 1:primitives.num_angles*primitives.num_prim
    if(isempty(primitives.trajectories(primind).points)==0)
        fprintf(file, 'primID: %d\n', primitives.trajectories(primind).id);
        fprintf(file, 'startangle_c: %d\n', ...
            primitives.trajectories(primind).start_angle);
        fprintf(file, 'endpose_c: %d %d %d\n', ...
            primitives.trajectories(primind).endpose(1), ...
            primitives.trajectories(primind).endpose(2), ...
            primitives.trajectories(primind).endpose(3));
        fprintf(file, 'additionalactioncostmult: %d\n', ...
            primitives.trajectories(primind).cost);
        fprintf(file, 'intermediateposes: %d\n', ...
            primitives.num_samples);
        for interind = 1:primitives.num_samples
```

```
            fprintf(file, '%.4f %.4f %.4f\n', ...
                primitives.trajectories(primind).points(interind,1),...
                primitives.trajectories(primind).points(interind,2),...
                primitives.trajectories(primind).points(interind,3));
        end;
    end
end
fclose(file);
```

References

[1] Dieter Fox, W. Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation*, 4(1), 1997.

[2] Johannes Meyer. Branch repository for the *sbpl_lattice_planner* for ros hydro and newer versions of ros. `https://github.com/meyerj/sbpl_lattice_planner`, July 2015.

[3] sbpl. Search-based planning lab. `http://www.sbpl.net/`, March 2015.

IRI reports

This report is in the series of IRI technical reports.
All IRI technical reports are available for download at the IRI website
`http://www.iri.upc.edu`.