POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

DWA vs custom trajectory tracking controller: a comparison in ROS

Project for the Control of Mobile Robots course
Computer Science and Engineering

Giuseppe Chiari, 10576799
Leonardo Gargani, 10569221
Serena Salvi, 10607377

Supervisor:
Luca Bascetta

Academic year:
2022/2023

Abstract:
The Dynamic Window Approach (DWA) is an online collision avoidance strategy for mobile robots. It incorporates the dynamics of the robot by reducing the search space to only the velocities reachable within a short time interval.
In this work we first present a comparison between the DWA algorithm from the paper and its implementation in Robot Operating System (ROS).
Then, a further comparison is made between the implementation above and a custom trajectory tracking controller, which is composed of an inner linearisation law (based on the kinematic model) and an outer tracking law (based on a proportional integral controller with velocity feed-forward).

Table of Contents

1. Introduction

1.1. Project overview

This project aims at comparing the behavior of a robot when controlled in two different ways: first with a trajectory tracking controller, and then with the Dynamic Window Approach (DWA).

The software performs a simulation of a differential drive robot, using its kinematic model, and implements the two controllers. To test the quality of the results, an eight-shaped trajectory has been chosen as the reference one to be followed.

Everything runs on *ROS Melodic* on *Ubuntu 18.04*.

1.2. Background on differential drive

$$\begin{cases} \omega_R = \frac{v+\omega \cdot d/2}{r} \\ \omega_L = \frac{v-\omega \cdot d/2}{r} \end{cases} \qquad b = c \tag{1}$$

2. DWA overview

This section contains a brief overview of DWA as presented in the original paper[1]. Basically, it is an approach to perform collision avoidance in mobile robots, while dealing with the constraints imposed by limited velocities and accelerations.

Given a certain goal point to be reached by the robot, DWA tries to find the optimal linear and angular velocities to go there. This is mainly done in two steps: search space reduction and objective function optimization.

2.1. Search space reduction

This approach consists in reducing the search space to those velocities which are reachable under the dynamic constraints and are safe with respect to obstacles.
One of the core concepts of DWA is the so-called search space. It can be seen as a two-dimensional space where each point represents a tuple (v, ω) of velocities where v is the linear velocity of the robot and ω is the angular velocity.

An initial reduction of this space is obtained by searching only in circular trajectories of the robot.
In fact, at each time instant the velocities (v, ω) are considered constant for the next n time intervals making up the simulated trajectory. The search is repeated after each time interval.

The search space is further reduced by considering only all the admissible velocities, which correspond to the velocities allowing the robot to stop before it reaches the closest obstacle.
Finally, given the limited accelerations of the robot, all the velocities that can't be reached within a short time interval are left out too.

2.2. Objective function optimization

The remaining velocities are fed into the following objective function to be maximized:

$$G(v, \omega) = \sigma(\alpha \cdot heading(v, \omega) + \beta \cdot dist(v, \omega) + \gamma \cdot vel(v, \omega))$$

This function trades off the following aspects:

- *heading*, which is a measure of progress towards the goal location;
- *dist*, which is the distance to the closest obstacle on the trajectory;

[1] D. Fox, W. Burgard, S. Thrun (1997) *The Dynamic Window Approach to Collision Avoidance*

- *vel*, which is the forward velocity of the robot.

Each one of the three quantities above is multiplied to its own weight (α, β, γ), and the resulting quantity is passed to a smoothing function (σ).

At this point the tuple (v, ω) leading to the highest score is chosen and the robot picks those velocities.

3. DWA in ROS

3.1. From ROS wiki

DWA is already implemented in ROS in the `dwa_local_planner`[2] package.

As stated in the ROS Wiki:

> "This package provides a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx,dy,dtheta velocities to send to the robot."

This package is ought to be used as the planner for `move_base`[3] (package providing an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base) within the navigation stack.

In the Wiki the following cost function to score each trajectory is presented:

```
cost = path_distance_bias * (distance to path from the endpoint of the trajectory)
       + goal_distance_bias * (distance to local goal from the endpoint of the trajectory)
       + occdist_scale * (maximum obstacle cost along the trajectory in obstacle cost (0-254))
```

However, reading through the source code of the DWA ROS library, it is said that the above function is *"used for visualization only, total_ costs are not really total costs"* [4]. This is because ROS uses a slightly different cost function that will be described in the next paragraph.

3.2. ROS libray source code

In order to use DWA in our experiment, we call the `DWAPlannerROS::computeVelocityCommands()` function which leads to the following cascading function calls:

1. `DWAPlannerROS::computeVelocityCommands()`[5] called by `diffdrive_traj_ctrl.cpp` at line 78;
2. `DWAPlannerROS::dwaComputeVelocityCommands()`[6] called by function above at line 302;
3. `DWAPlanner::findBestPath()`[7] called by function above at line 209;
4. `SimpleScoredSamplingPlanner::findBestTrajectory()`[8] called by function above at line 317;
5. `SimpleScoredSamplingPlanner::scoreTrajectory()`[9] called by function above at line 105.

At this point, inside `scoreTrajectory()` the different cost elements are evaluated to score the trajectory. Each cost element is an instance of the abstract `base_local_planner::TrajectoryCostFunction` class.

These elements are contained in a `std::vector<TrajectoryCostFunction*>` vector and, in the case of DWA, they are 7 in total:

[2] https://wiki.ros.org/dwa_local_planner
[3] https://wiki.ros.org/move_base
[4] https://docs.ros.org/en/melodic/api/dwa_local_planner/html/dwa__planner_8cpp_source.html
[5] https://docs.ros.org/en/melodic/api/dwa_local_planner/html/dwa__planner__ros_8cpp_source.html
[6] https://docs.ros.org/en/melodic/api/dwa_local_planner/html/dwa__planner__ros_8cpp_source.html
[7] https://docs.ros.org/en/melodic/api/dwa_local_planner/html/dwa__planner_8cpp_source.html
[8] https://docs.ros.org/en/melodic/api/base_local_planner/html/simple__scored__sampling__planner_8cpp_source.html
[9] https://docs.ros.org/en/melodic/api/base_local_planner/html/simple__scored__sampling__planner_8cpp_source.html

- `base_local_planner::OscillationCostFunction oscillation_costs_`, which penalizes trajectories where the robot oscillates;
- `base_local_planner::ObstacleCostFunction obstacle_costs_`, which penalizes trajectories where the robot occupies illegal positions with its footprint;
- `base_local_planner::MapGridCostFunction goal_front_costs_`, which prefers trajectories that make the nose go towards (local) nose goal;
- `base_local_planner::MapGridCostFunction alignment_costs_`, which prefers trajectories that keep the robot nose on nose path;
- `base_local_planner::MapGridCostFunction path_costs_`, which prefers trajectories on global path;
- `base_local_planner::MapGridCostFunction goal_costs_`, which prefers trajectories that go towards (local) goal;
- `base_local_planner::TwirlingCostFunction twirling_costs_`, which prefers trajectories that don't spin.

The considered trajectories by DWA are those taken from a equi-distant discretization of the velocities that the robot can assume. For this purpose ROS has a `base_local_planner::SimpleTrajectoryGenerator`[10] class.
WEIGHTED SUM... (descrivi la funzione di costo implementata nella libreria)

4. Trajectory tracking controller

...

5. Setup of the experiment

5.1. The robot

In our experiment we chose to simulate a small differential drive robot.

In particular, it is characterized by two main dimensions (specified as YAML parameters in the code):
- `d` = 15 cm, which is the distance between the two motorized wheels;
- `r` = 3 cm, which is the radius of the two motorized wheels.

The precise footprint is a pentagon, just for convenience, so that when looking at it we are able to determine the orientation of the robot. However, this is not a decisive detail since it has o influence on the robot's behavior.

5.2. The map

Regarding the map, it is important to highlight the different setting we have with respect to the usual DWA use.
There are two main differences:
- in our setting there are no obstacles, neither fixed nor moving;
- the robot does not have any sensor.

As a consequence, we don't need both the local map and the global map, but only the local one.
Moreover, this map needs to be empty so it is generated starting from a totally white image.

5.3. The trajectory

The robot must follow a precise trajectory, which is used to perform all benchmarks.
In our case we chose an eight-shaped trajectory with a dimension of 2 x 1 meters.

In order to make DWA compute the velocities of the robot, we must feed it a goal.
This means that the complete trajectory has to be "discretized" in multiple points. Each one of these points is passed to DWA as the current goal, and once it is reached the next point is set as the new goal. You will find a detailed explanation in the following sections.

[10] https://docs.ros.org/en/melodic/api/base_local_planner/html/classbase__local__planner_1_1SimpleTrajectoryGenerator.html

6. Implementation

6.1. Architecture overview

Our implementation is composed of three packages: one simulator (`diffdrive_kin_sim`) and two controllers (`diffdrive_kin_ctrl` and `diffdrive_dwa_ctrl`).
The two controllers are interchangeable and are meant to always be used together with the simulator, one at a time.

6.2. Service

6.2.1 `generate_desired_path_service`

The 8-shaped trajectory that the robot must follow is generated inside `eight_traj_gen.cpp` executed in the launch file.
As a result, the two vectors contained in the `GenerateDesiredPathService.srv` message are populated with the coordinates of all the points of the trajectory.
As soon as the service is called, during the initialization (in the `Prepare()` function) of both controllers, the complete trajectory is made available also to the controllers.

6.3. Nodes

6.3.1 `diffdrive_kin_sim_node`

This node is subscribed to the `/robot_input` topic and reads the wheels velocities.
Given those it simulates the movement of the robot and publishes the new position in both `/robot_state` and `/odom` topics. We have decided to keep these two topics separate in order to accommodate both controllers.

The actual simulator is an object of the `diffdrive_kin_ode` class, which is initialized as follow:

```
simulator = new diffdrive_kin_ode(dt);
```

and contains the integration logic to update the robot pose.

6.3.2 `diffdrive_kin_trajctrl_node`

This node is dedicated to compute the angular velocities of the two wheels given the current position of the robot and its next point in the trajectory.

During the initialization phase it queries the `generate_desired_path_service` service and stores the complete trajectory.
Then, every time a message is published on the `/robot_state` topic this node updates its internal values of the robot pose.

The `PeriodicTask()` that gets executed uses a PID controller and a linearization law (implemented in the `diffdrive_kin_fblin` class) to compute both the linear and the angular velocities of the robot.
As a last step, since the simulated robot is a differential drive one, the angular velocities of the two wheels are computed starting from the (v, ω) above, and the new results are published on the `/robot_input` and `/controller_state` topics.

6.3.3 `diffdrive_dwa_trajctrl_node`

This node is used to interface the DWA library in ROS with the simulator.
This is due to the fact that natively DWA is used inside ROS Navigation Stack, thus it expects an odometry source and a costmap.

In this specific implementation only a global costmap and a planner (DWA) are used.
As the controller explained above, during the initialiation phase the service is queried. Once the trajectory is received, one point every `skipped_goals` is set as the new plan for DWA.

Once the plan is set, the DWA controller tries to reach the new goal position by computing the necessary pair of (v, ω) velocities. As soon as the goal is reached with a certain tolerance, the next point in the trajectory is set as the new goal.

Lastly, the (ω_r, ω_l) velocities are computed and published on the `/robot_input` and `/controller_state` topics. The implementation of DWA and all the functions that have been used is provided by the following two files in the ROS library:

- `dwa_planner.cpp`
- `dwa_planner_ros.cpp`

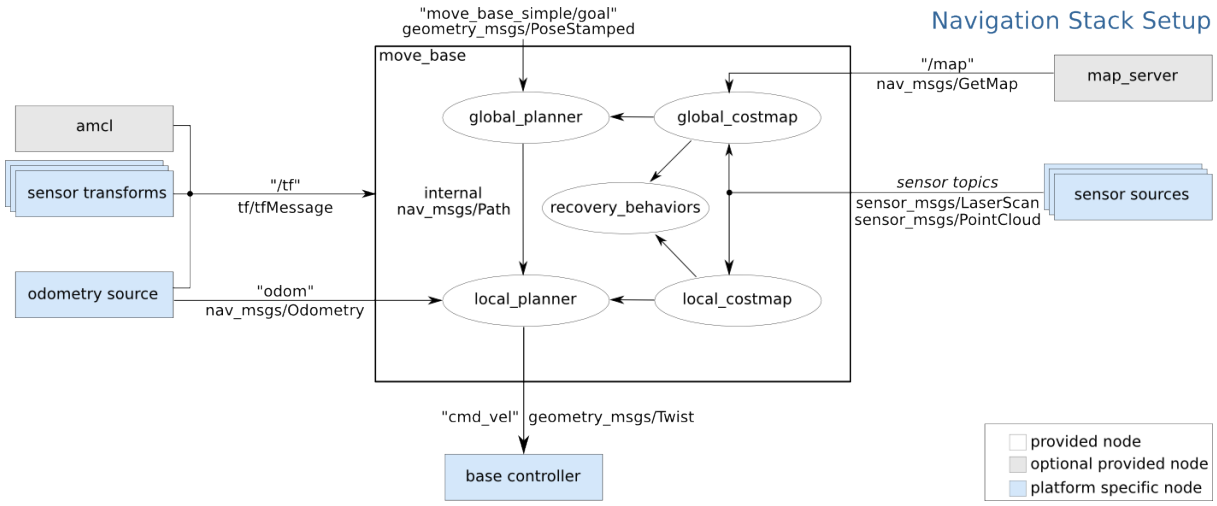


Figure 1: ROS Navigation Stack.

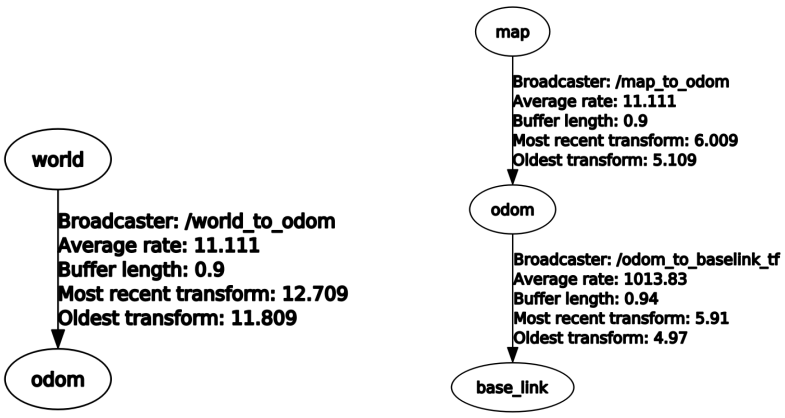
6.3.4 `odom_to_baselink_tf_node`

This node is needed in order to link the `odom` and the `base_link` coordinate frames through a simple dynamic tf.

6.4. Frames

Here is a list with a brief description of all the frames:

- `map`, which is the coordinate system where the empty map (provided by the map server) is;
- `odom`, which represent the global reference system, that in this particular case its origin matches the one of the `map` frame;
- `base_link`, which represrents reference system moving around together with the robot.



(a) `rqt_tf_tree` of the custom controller. (b) `rqt_tf_tree` of DWA standalone.

Figure 2: Frames in the two approaches.

6.5. Topics

Here is a list with a brief description of the topics that our nodes are subscribed to and published into:

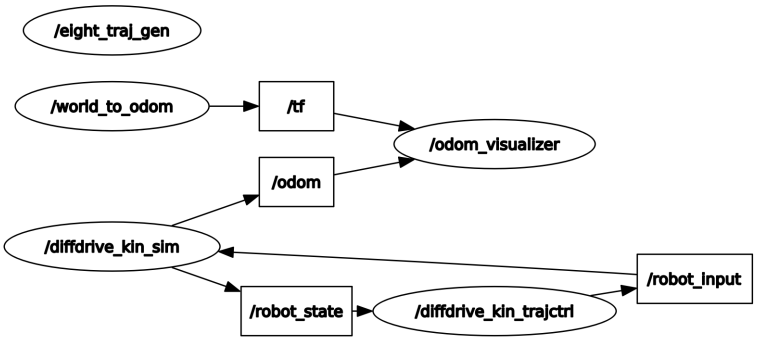
- `/clock`, used to synchronize all the nodes in the simulation;
- `/odom`, used to pass the odometry information of the robot to DWA;
  - publishers:
    - `diffdrive_kin_sim`
  - subscribers:
    - `odom_to_baselink_tf`
- `/robot_state`, used to pass the odometry information of the robot to the custom controller.
  - publishers:
    - `diffdrive_kin_sim`
  - subscribers:
    - `diffdrive_kin_trajctrl`
- `/controller_state`, used to publish details about the controller for visualization purposes;
  - publishers:
    - `diffdrive_kin_trajctrl`
    - `diffdrive_dwa_trajctrl`
  - subscribers:
    - *none*
- `/robot_input`, used to communicate (ω_r, ω_l) computed by the controllers;
  - publishers:
    - `diffdrive_kin_trajctrl`
    - `diffdrive_dwa_trajctrl`
  - subscribers:
    - `diffdrive_kin_sim`

6.6. Launch files

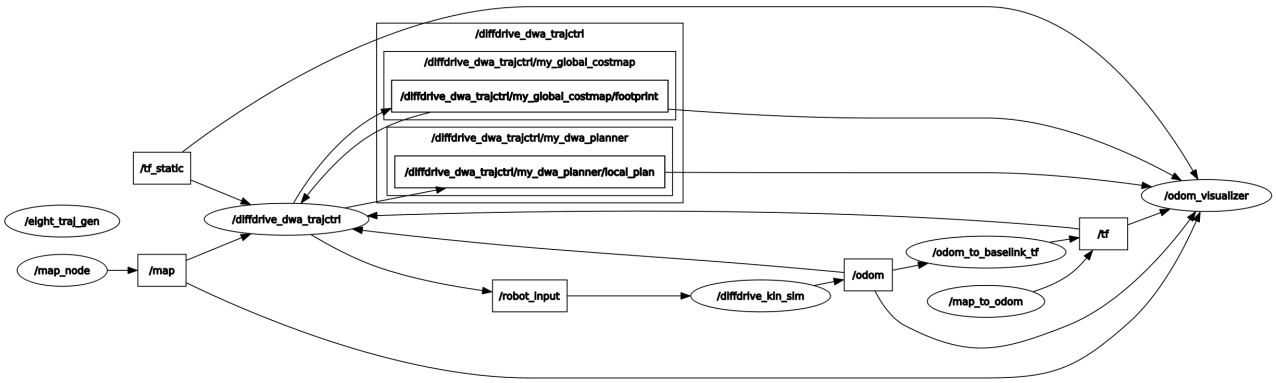
Here is a list with a brief description of all the launch files:

- `diffdrive_kin_trajctrl.launch`, used to start the following nodes:
  - `diffdrive_kin_sim`
  - `diffdrive_kin_trajctrl`
  - `eight_traj_gen`
  - `world_to_odom` (static tf linking `world` and `odom` frames)
  - `odom_visualizer` (*RVIZ* node to visualize data)

- `diffdrive_dwa_trajctrl.launch`, used to start the following nodes:
  - `diffdrive_kin_sim`
  - `diffdrive_dwa_trajctrl`
  - `eight_traj_gen`
  - `odom_to_baselink_tf`
  - `map_node` (*map_ server* node used to provide the map)
  - `map_to_odom` (static tf linking `map` and `odom` frames)
  - `odom_visualizer` (*RVIZ* node to visualize data)



(a) `rqt_graph` of the custom controller.



(b) `rqt_graph` of DWA standalone.

Figure 3: Architecture of the two approaches.

7. Parameters

7.1. List of parameters

Parameters have a crucial role for the correct functioning of the nodes introduced above.

In this section the most relevant ones are explained:

- `diffdrive_kin_sim.yaml`:
  - `d`, distance between the two wheels;
  - `r`, radius of the two wheels.

- `eight_traj.yaml`:
  - `a`, amplitude of the eight-shaped trajectory;
  - `w`, ratio $\frac{2 \cdot \pi}{T}$ where T is the time duration of each lap;
  - `trajectory_length`, length of the eight-shaped trajectory;
  - `increment_step`, increment step when discretizing the trajectory.

- `diffdrive_kin_trajctrl.yaml`:
  - `Kp`, proportional gain of the PID controller;
  - `Ki`, integral gain of the PID controller;
  - `Kd`, derivative gain of the PID controller.

- `diffdrive_dwa_trajctrl.yaml`:
  - `skipped_goals`, number of points to skip when feeding the trajectory to DWA.

- `dwa_planner_params.yaml`:
  - `min_vel_y`, minimum linear velocity along the y-axis;
  - `max_vel_y`, maximum linear velocity along the y-axis;

- `min_vel_x`, minimum linear velocity along the x-axis;
- `max_vel_x`, minimum linear velocity along the x-axis;
- `acc_lim_theta`, maximum angular acceleration;
- `vth_samples`, number of uniformly-sampled values of ω that DWA considers when simulating;
- `vx_samples`, number of uniformly-sampled values of v that DWA considers when simulating;
- `path_distance_bias`, weighting for how much the controller should stay close to the given path;
- `goal_distance_bias`, weighting for how much the controller should attempt to reach its local goal;
- `xy_goal_tolerance`, tolerance (in meters) in the x & y distance when reaching a goal;
- `yaw_goal_tolerance`, tolerance (in radians) in yaw/rotation when reaching a goal.

7.2. Tuning

7.2.1 Eight-shaped trajectory

An eight-shaped trajectory can be be described with the following analytical equations:

$$x = \mathtt{a} \cdot \sin(\mathtt{w} \cdot t)$$

$$y = \mathtt{a} \cdot \sin(\mathtt{w} \cdot t) \cdot \cos(\mathtt{w} \cdot t)$$

In the equations above it is possible to tune two parameters: `a` and `w`. In the presented implementation they are set to `a` = 1 and `w` = 1.

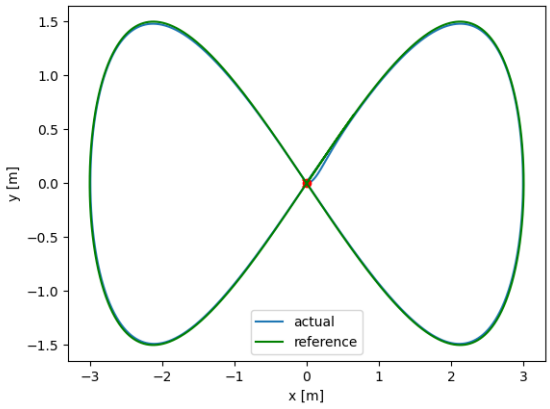


Figure 4: Trajectory when `a = 3.0`.

7.2.2 Custom controller

The custom controller is a trajectory tracking one composed of an inner linearisation law and an outer tracking law. The outer tracking law is based on a PID controller with tunable parameters: K_p, K_i, K_d.

The step response when using a PID controller is computed as follows:

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

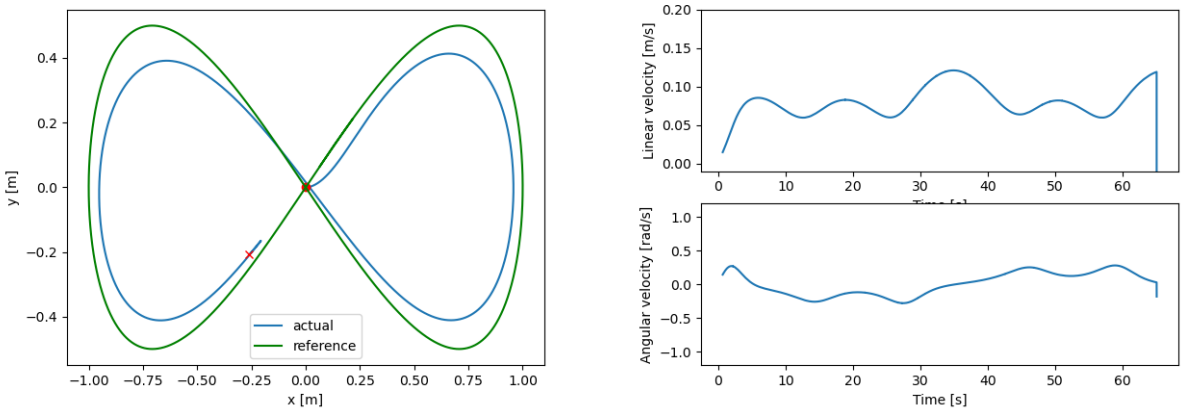
Effects of *increasing* a parameter independently

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

Figure 5: PID parameters tuning.

Various tests have been carried out to achieve the optimal behaviour, in particular to minimize the overshoot and the settling time while keeping an overall good stability in the PID response.

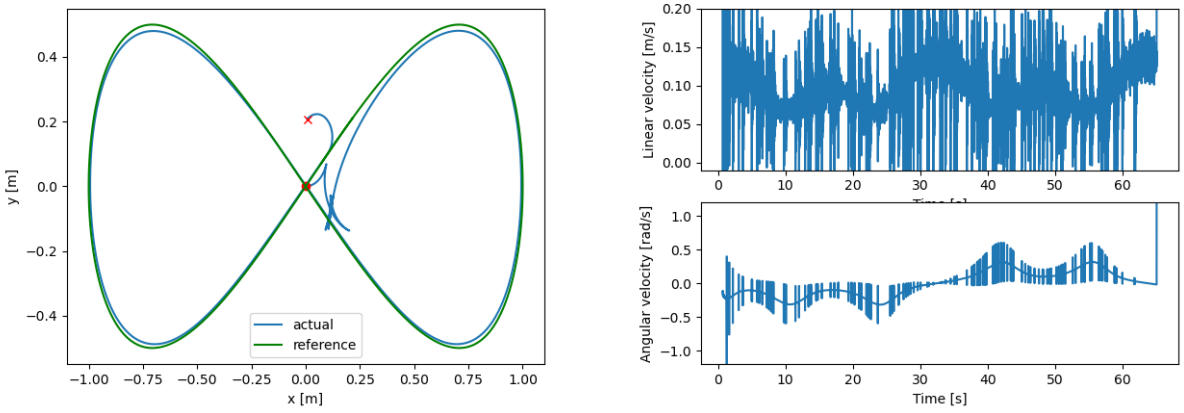
In the presented implementation they are set to $K_p = 0.8$, $K_i = 0.8$, $K_d = 0$.



(a) Trajectory.

(b) Linear and angular velocities.

Figure 6: Behavior with $K_p = 0.2, K_p = 0.8, K_p = 0.0$.



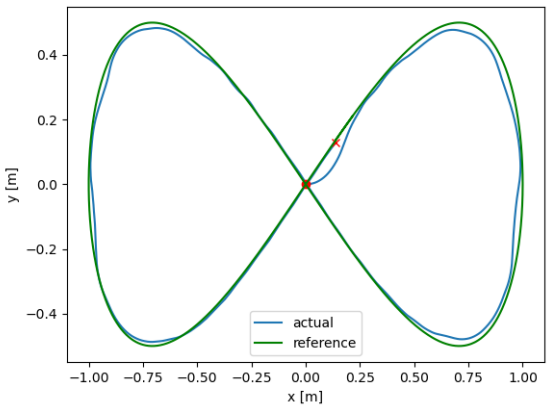
(a) Trajectory.

(b) Linear and angular velocities.

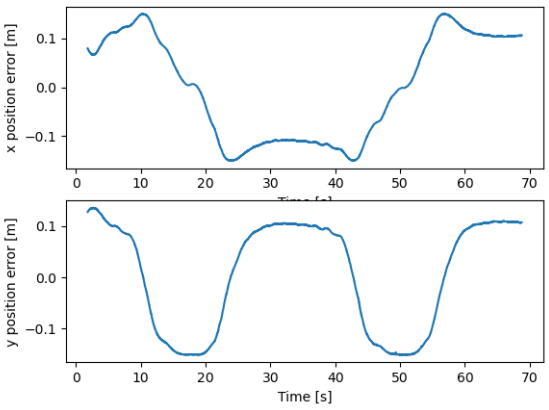
Figure 7: Behavior with $K_p = 0.8, K_p = 3.2, K_p = 1.0$.

7.2.3 DWA

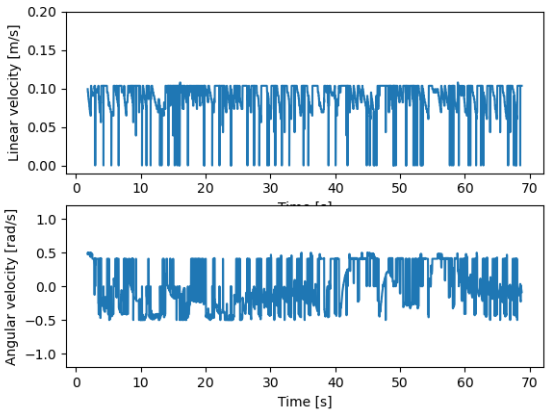
When the trajectory is generated, the eight shape is discretized over numerous points.
Due to the large number of those, they are very close to each other. This makes the robot misbehave as soon as a new goal is set in DWA. For this reason it is best to skip some of the points in the trajectory. This is accomplished through the `skipped_goals` parameters, which is set to 15.



(a) Trajectory.

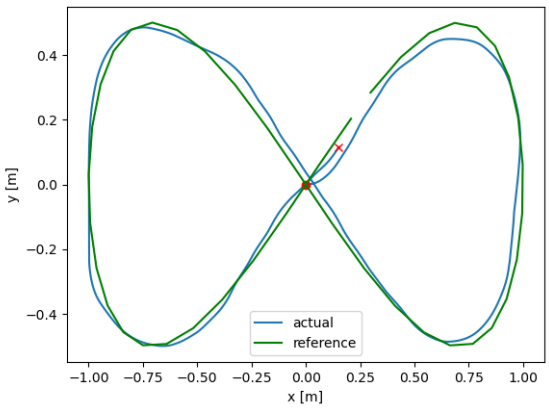


(b) Position Error.

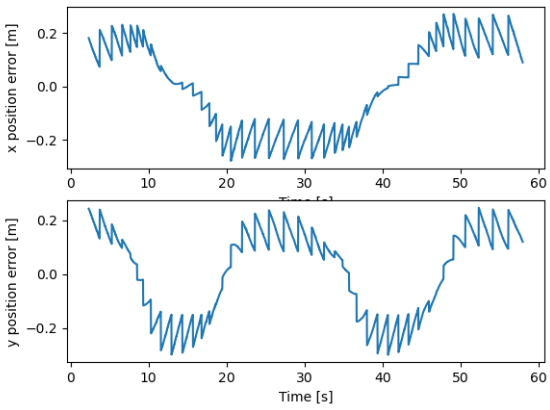


(c) Linear and angular velocities.

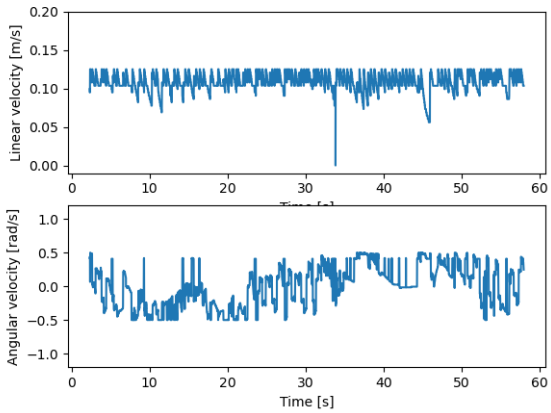
Figure 8: Behavior with `skipped_goals = 0`.



(a) Trajectory.



(b) Position Error.



(c) Linear and angular velocities.

Figure 9: Behavior with `skipped_goals = 150`.

In order to simulate a differential drive robot, the following parameters have been set:

- `min_vel_y` $= 0$ and `max_vel_y` $= 0$, to forbid lateral movement along the y-axis;
- `min_vel_x` $= 0$, to forbid backward movement along the x-axis.

Another parameter related to the constraints on the movement of the robot is `acc_lim_theta`. To achieve the best trade-off betweeen the ability to correctly follow the trajectory and the realistic constraints on the steering capabilities of the robot, the most suitable value for it is 10.

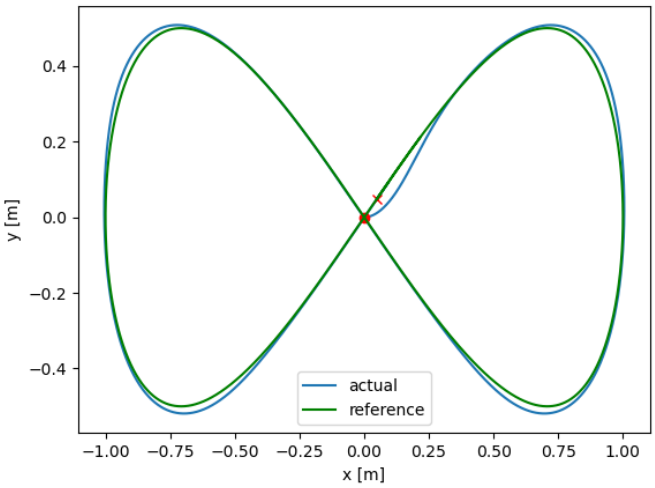
Since DWA discretely sample in the robot's control space, parameters `vth_samples` and `vx_samples` play a key role in allowing a finer grained choice of velocities. To achieve the best trade-off betweeen a wide choice of possible (v, ω) and computational effort, the optimal values have been found to be `vth_samples = 100` and `vx_samples = 30`.

Finally, DWA provides a certain tolerance within which the goal is considered to be reached. In particular this can be done through two parameters `yaw_goal_tolerance` and `xy_goal_tolerance`. The former has been set to `6.3` because it is irrelevant the direction of the robot when reaching the goal (thus $6.3 \simeq 2\pi$ in radians). The latter has been set to `0.15` because it is a good deal between the dimension of the robot and an approximation of the trajectory.

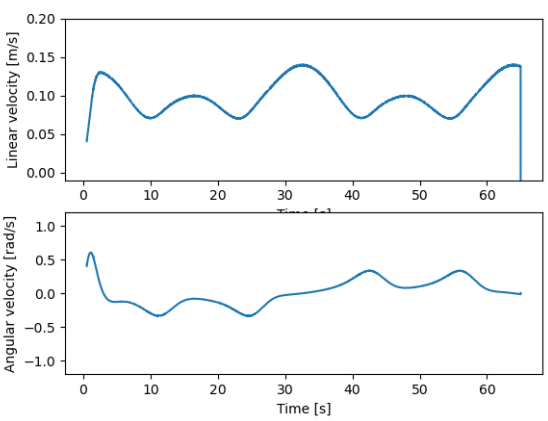
8. Experimental Results

8.1. Tuned trajectory tracking controller

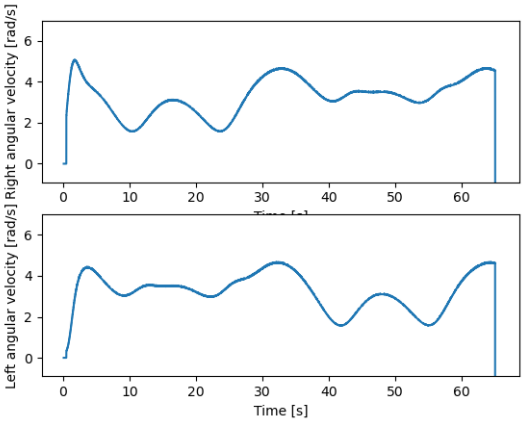
Here are the final results obtained with the optimal parameters in the trajectory tracking controller:



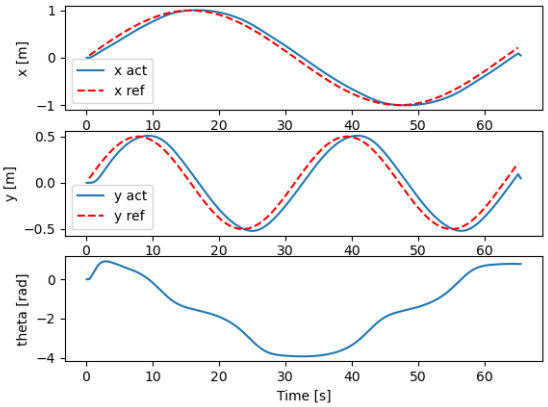
(a) Trajectory.



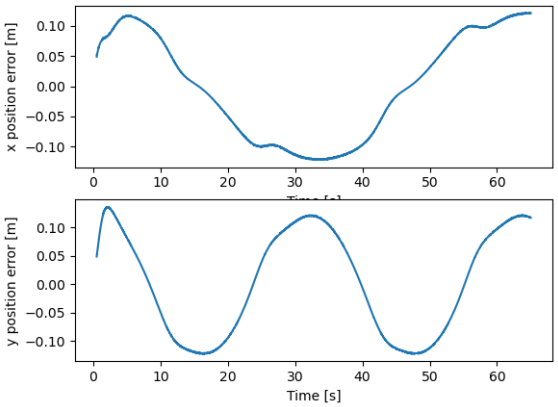
(b) Linear and angular velocities.



(c) Wheels velocities.



(d) Pose.

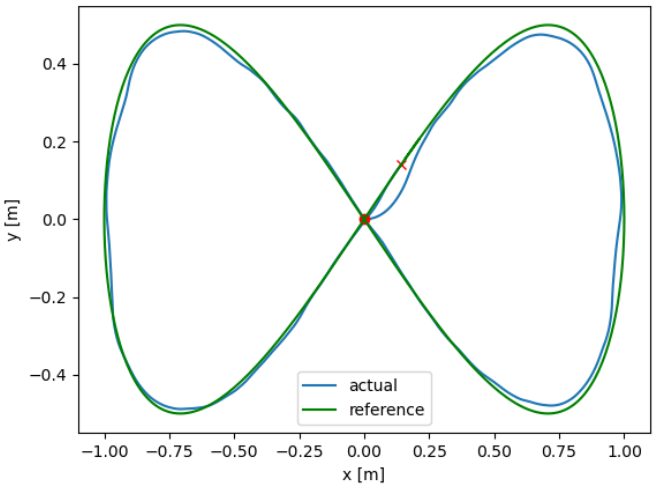


(e) Position Error.

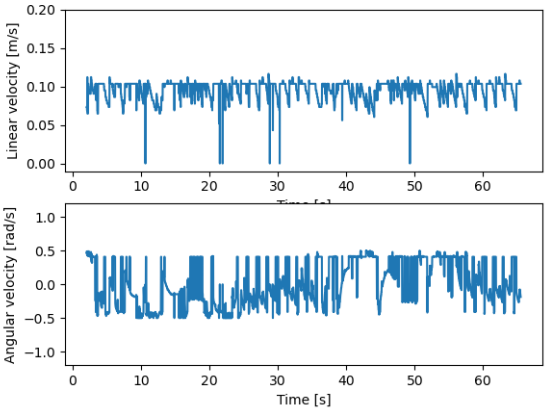
Figure 10: Behavior with the tuned custom controller.

8.2. Tuned DWA

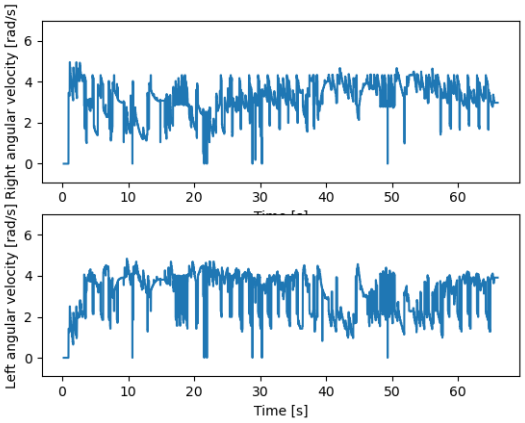
Here are the final results obtained with the optimal parameters in DWA:



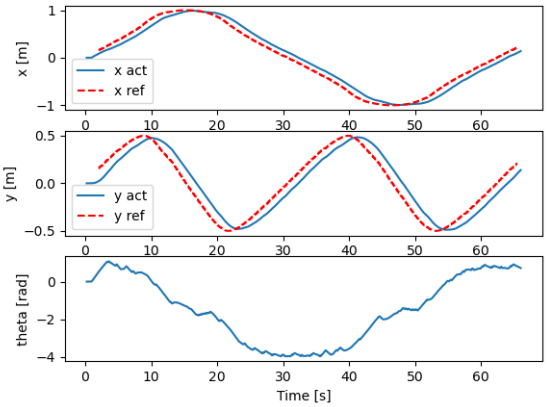
(a) Trajectory.



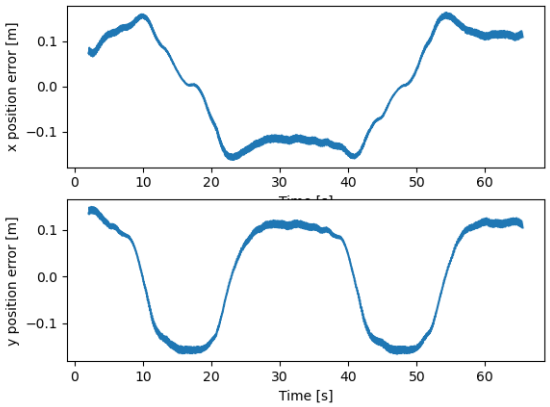
(b) Linear and angular velocities.



(c) Wheels velocities.



(d) Pose.



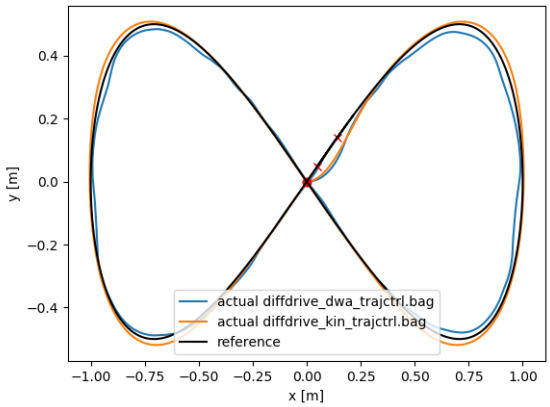
(e) Position Error.

Figure 11: Behavior with tuned DWA.

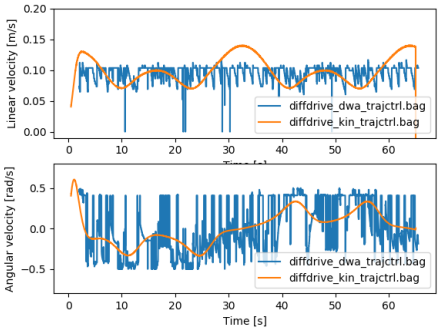
8.3. Comparison

Below there is a direct comparison between the behavior of the trajectory tracking controller and DWA.

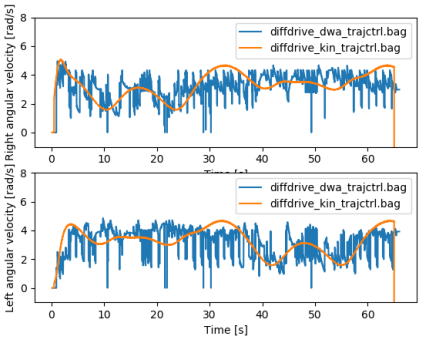
Generally, the custom controller performs better than DWA due to the sudden changes (spikes) in the velocities of the latter. On the other hand, the performance of the two controllorers is quite similar in following the eight-shaped trajectory.
It is possible to achieve better smoothness in velocities by increasing the value of `skipped_goals`. This is because when increasing the distance between two consecutive goals DWA tends to choose a velocity closer to the current one, while the trajectory would be followed in a less precise way.



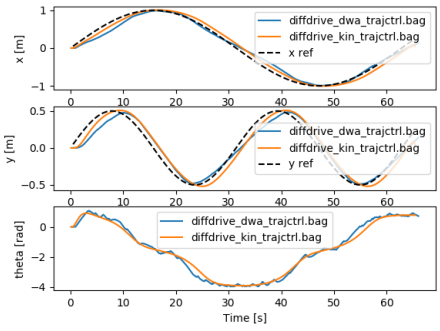
(a) Trajectory comparison.



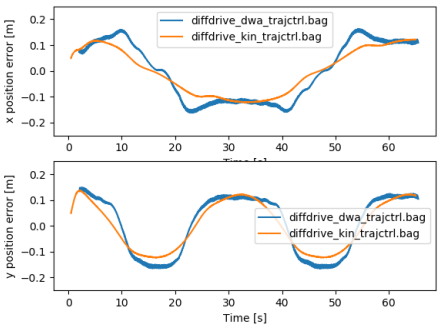
(b) Linear and angular velocities comparison.



(c) Wheels velocities comparison.



(d) Pose comparison.



(e) Position Error comparison.

Figure 12: Comparison between the two approaches.

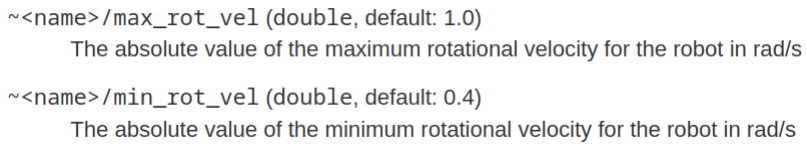
9. Encountered problems

9.1. Deprecated parameters

During the tuning phase of DWA a trial and error approach has been carried out, finding that a few parameters are named differently in the in the ROS library [11] with respect to the ROS Wiki [12].
Such renaming was not advertised by any error nor warning, so it was difficult to spot where the problem was when changing parameters and seeing no effect in the robot behavior.
In fact, the warning would be raised only if DWA was instantiated inside the `nav_core` architecture, which is not the case when using DWA standalone.

The following pair of parameters is an example:

- `min_rot_vel` renamed into `min_vel_theta`;
- `max_rot_vel` renamed into `max_vel_theta`.



(a) ROS Wiki description.

```
// Warn about deprecated parameters -- remove this block in N-turtle
nav_core::warnRenamedParameter(private_nh, "max_vel_trans", "max_trans_vel");
nav_core::warnRenamedParameter(private_nh, "min_vel_trans", "min_trans_vel");
nav_core::warnRenamedParameter(private_nh, "max_vel_theta", "max_rot_vel");
nav_core::warnRenamedParameter(private_nh, "min_vel_theta", "min_rot_vel");
nav_core::warnRenamedParameter(private_nh, "acc_lim_trans", "acc_limit_trans");
nav_core::warnRenamedParameter(private_nh, "theta_stopped_vel", "rot_stopped_vel");
```

(b) ROS library source code warning.

Figure 13: Deprecated parameters.

9.2. DWA used standalone

Usually DWA is used inside a standard architecture which is the Navigation Stack in ROS.
However the goal of this project was to integrate DWA in the architecture with the simulator used for the custom controller. Thereby DWA has been rearranged in a standalone mode.

To accomplish this, it has been necessary to provide to DWA all the information it normally expects on certain topics such as `/odom`, `/goal`, and `/map`.

Unfortunately there is no documentation for this kind of setup on the ROS Wiki, and the few lines of code that are present refer to some deprecated libraries (i.e. `tf` is the one mentioned in the example, while `tf2` is the actual supported one).

[11] https://docs.ros.org/en/melodic/api/dwa_local_planner/html/dwa__planner__ros_8cpp_source.html
[12] https://wiki.ros.org/dwa_local_planner

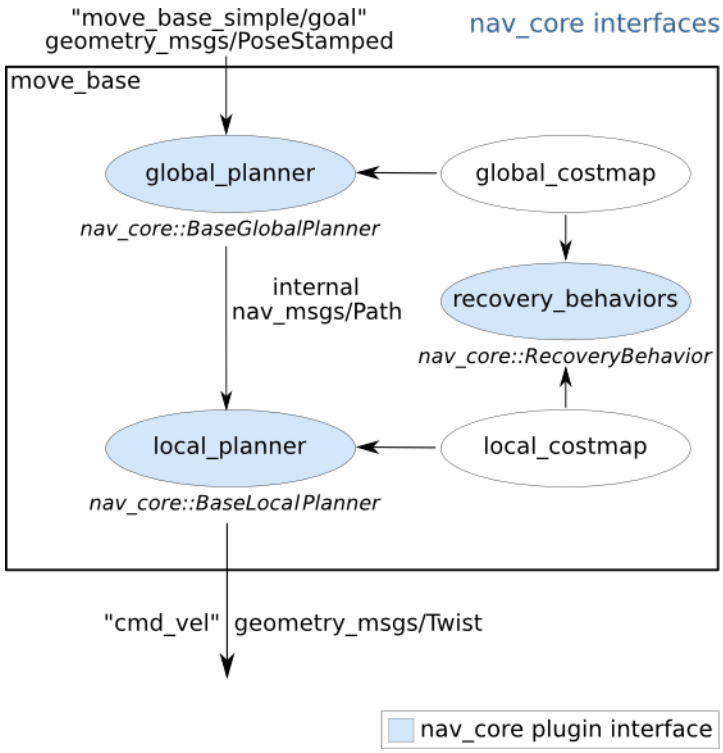


Figure 14: `nav_core` package architecture.

9.3. Multiple goals

In a standard scenario DWA needs only the one final goal and computes the trajectory on its own.
In this project the trajectory is predefined and must be forced changing continuously the goal, otherwise the robot would go straight to the final point of the overall eight-shaped trajectory.

10. Usage of the code

10.1. Installation

First of all, all the software written in this project has been tested on *Ubuntu 18.04 LTS* and *ROS Melodic*.

In order to install ROS and all the required packages, execute:

```
sudo apt instal ros-melodic-desktop-full
sudo apt install ros-melodic-costmap-2d ros-melodic-base-local-planner ros-melodic-dwa-local-
    planner ros-melodic-map-server
```

10.2. Setup and compilation

Clone or download the repository into your home (~) folder.

If you use bash, add the following line to the end of your .bashrc:

```
source ~/ROS_trajectory_tracking_controller/devel/setup.bash
```

If you use zsh, add instead this other line to the end of your .zshrc:

```
source ~/ROS_trajectory_tracking_controller/devel/setup.zsh
```

Enter the project root directory and compile everything with:

```
cd ROS_trajectory_tracking_controller
catkin_make
```

10.3. Simulation and results

The following instructions let you perform and visualize a simulation.

10.3.1 Custom controller

This simulation shows the behavior of the custom controller when an 8-shaped trajectory is set.

[terminal 1] Start the simulation:

```
roslaunch diffdrive_kin_ctrl diffdrive_kin_trajctrl.launch
```

[terminal 2] Enter the `script/` folder and record the simulation:

```
cd ~/ROS_trajectory_tracking_controller/src/diffdrive_kin_ctrl/script
rosbag record -a -O diffdrive_kin_trajctrl.bag
```

Wait about 30 seconds so that the simulation can be performed.

[terminal 2] Stop the recording with Ctrl-C.

[terminal 1] Stop the simulation with Ctrl-C.

[terminal 2] Visualize the results:

```
python plot_results.py diffdrive_kin_trajctrl.bag
```

10.3.2 DWA

This simulation shows the behavior of DWA when an 8-shaped trajectory is set.

[terminal 1] Start the simulation:

```
roslaunch diffdrive_dwa_ctrl diffdrive_dwa_trajctrl.launch
```

[terminal 2] Enter the script/ folder and record the simulation:

```
cd ~/ROS_trajectory_tracking_controller/src/diffdrive_kin_ctrl/script
rosbag record -a -O diffdrive_dwa_trajctrl.bag
```

Wait about 30 seconds so that the simulation can be performed.

[terminal 2] Stop the recording with Ctrl-C.

[terminal 1] Stop the simulation with Ctrl-C.

[terminal 2] Visualize the results:

```
python plot_results.py diffdrive_dwa_trajctrl.bag
```

10.3.3 Compare two simulations

If you wish, you can also compare the results of two different simulations.
Given the two bag files named diffdrive_dwa_trajctrl.bag and diffdrive_kin_trajctrl.bag, visualize the comparison:

```
python plot_comparison.py diffdrive_dwa_trajctrl.bag diffdrive_kin_trajctrl.bag
```

11. Conclusions