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

A differential wheeled robot is a mobile robot whose movement is based on two separately driven wheels, having a rotational velocity ω_r, ω_l.

Let's define r as the radius of the wheels, d as the distance between the two wheels, and θ as the orientation of the robot. Then, its representation in the space is the following:

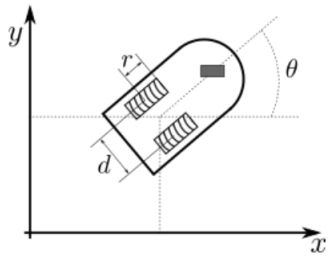


Figure 1: Differential drive robot

The kinematic model of a differential drive vehicle is:

$$\begin{cases} \dot{x} = \frac{\omega_r+\omega_l}{2} r\cos(\theta) \\ \dot{y} = \frac{\omega_r+\omega_l}{2} r\sin(\theta) \\ \dot{\theta} = \frac{\omega_r-\omega_l}{d} r \end{cases}$$

The linear and angular velocities of the vehicle are related to wheel velocities by the simple equations:

$$\begin{cases} \omega_r = \frac{v+\omega\cdot d/2}{r} \\ \omega_l = \frac{v-\omega\cdot d/2}{r} \end{cases}$$

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.

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

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;
- *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.

[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

3.2. ROS libray source code

In order to use DWA standalone, it is possible to use 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, `scoreTrajectory()` evaluates the different cost elements of a single trajectory in order to give it a score. This score will be used to determine the best trajectory among all the evaluated ones.
Each cost element is an instance of the abstract `base_local_planner::TrajectoryCostFunction` class, and takes into account a different aspect of the trajectory.

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

- `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 final score of each trajectory is then computed as the sum of all the cost elements above.

It is worth mentioning that the trajectories evaluated by DWA are obtained through the `base_local_planner::SimpleTrajectoryGenerator`[10] class, which generates trajectories by performing an equi-distant discretization of the velocities that the robot can assume.

4. Trajectory tracking controller

4.1. Unicycle model control

When designing the controller it is possible to work on the unicycle (thus not differential) model, for simplicity:

$$\begin{cases} \dot{x} = v \cdot \cos(\theta) \\ \dot{y} = v \cdot \sin(\theta) \\ \dot{\theta} = \omega \end{cases}$$

In fact, the differential model can be easily reduced to the unicycle one with the transformations:

$$\begin{cases} v = \frac{\omega_r + \omega_l}{2} r \\ \omega = \frac{\omega_r - \omega_l}{2} r \end{cases}$$

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

With this simple substitution the differential model is obtained:

$$\begin{cases} \dot{x} = \frac{\omega_r + \omega_l}{2} r \cos(\theta) \\ \dot{y} = \frac{\omega_r + \omega_l}{2} r \sin(\theta) \\ \dot{\theta} = \frac{\omega_r - \omega_l}{d} r \end{cases}$$

Going back to the unicycle, which is simpler to work on, we must consider a point P on the robot chassis for control purposes, since the contact point of the wheel is affected by a nonholonomic constraint.

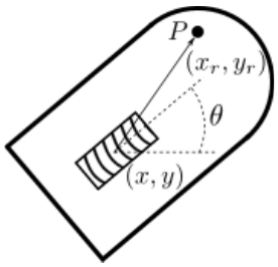


Figure 2: Point P on the robot chassis.

Therefore all the goal points that the robot must reach will be translated into goals for the point P, as explained in the next few paragraphs.

4.2. Controller's architecture

The developed controller works by taking only the reference points (making up the full trajectory), and then computing everything else on its own.

A comprehensive schematic representation of the whole controller is shown in the diagram below:

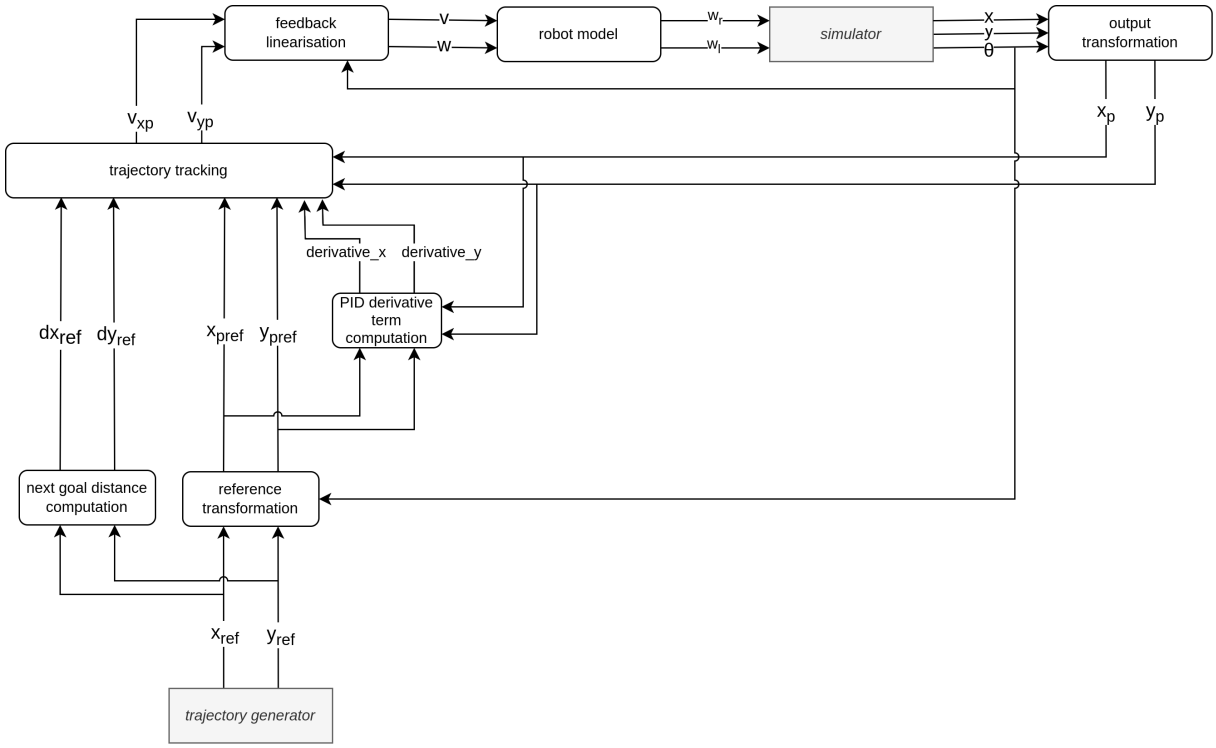


Figure 3: Scheme of the custom controller.

Let's break down the operations performed by each one of the blocks in the diagram.

The "trajectory tracking" block computes the linear velocity of point P:

$$v_{Px} = dx_{ref} + K_p \cdot error_{Px} + K_i \cdot error_{Px} \cdot RunPeriod + K_d \cdot derivative\_x$$

$$v_{Py} = dy_{ref} + K_p \cdot error_{Py} + K_i \cdot error_{Px} \cdot RunPeriod + K_d \cdot derivative\_y$$

The "feedback linearisation" block computes the linear and angular velocities of the robot:

$$v = v_{Px} \cdot \cos(\theta) + v_{Py} \cdot \sin(\theta)$$

$$\omega = \frac{v_{Py} \cdot \cos(\theta) - v_{Px} \cdot \sin(\theta)}{P\_dist}$$

The "robot model" block computes the angular velocities of the wheels:

$$\omega_r = \frac{v + \omega \cdot \frac{d}{2}}{r}$$

$$\omega_l = \frac{v - \omega \cdot \frac{d}{2}}{r}$$

The "output transformation" block computes the new position of point P:

$$x_P = x + P\_dist \cdot \cos(\theta)$$

$$y_P = y + P\_dist \cdot \sin(\theta)$$

The "next goal distance computation" block computes the distance between the old goal and the new goal:

$$dx_{ref} = x_{ref,t+1} - x_{ref,t}$$

$$dy_{ref} = y_{ref,t+1} - y_{ref,t}$$

The "reference transformation" block computes the reference position for point P (goal for point P):

$$x_{Pref} = x_{ref} + P\_dist \cdot \cos(\theta)$$

$$y_{Pref} = y_{ref} + P\_dist \cdot \sin(\theta)$$

The "PID derivative term computation" block computes the derivative term for the PID controller:

$$error\_x = x_{Pref} - x_P$$

$$error\_y = y_{Pref} - y_P$$

$$derivative\_x = \frac{error\_x - error\_x_{prev}}{RunPeriod}$$

$$derivative\_y = \frac{error\_y - error\_y_{prev}}{RunPeriod}$$

In the scheme there are also two blocks that are outside of the controller:

- the "simulator" block, which updates the pose of the robot given the angular velocities of the wheels;
- the "trajectory generator" block, which creates a vector of reference positions for the robot (goals for the center of the robot).

5. Setup of the experiment

5.1. The robot

The differential drive robot that has been chosen for the simulation is characterized by the following two dimensions, which are specified as YAML parameters in the code:

- `d` = 15 cm, which is the distance between the two wheels;
- `r` = 3 cm, which is the radius of the two wheels.

The precise footprint of the robot is a pentagon, just for convenience, so that it is easy to determine its orientation when looking at it in *Rviz*. However, this is a minor detail since it has no influence on the robot's behavior.

5.2. The map

The map is a component required by DWA to be able to work. Differently from the usual setting in which DWA is used, three main differences characterize this experiment:

- there are no obstacles in the map, neither fixed nor moving, and the robot does not have any sensor;
- the robot must follow a predefined trajectory, no global plan has to be computed.

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

5.3. The trajectory

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

In order to make DWA compute the velocities of the robot, a goal is required. This means that the complete trajectory needs to be "discretized" in multiple points, each one representing a goal. These points are then passed to DWA one at a time and set as the current goal. Once a point is reached the next one is set as the new goal. A more-detailed explanation can be found in the following sections.

6. Implementation

6.1. Architecture overview

The code used for this experiment is divided into three packages: one simulator (`diffdrive_kin_sim`) and two controllers (`diffdrive_kin_ctrl` and `diffdrive_dwa_ctrl`).
The two controllers, one implementing a trajectory tracking law and the other implementing DWA, are interchangeable and are meant to always be used together with the simulator, one at a time.

6.2. Services

A ROS service named `generate_desired_path_service`, implemented in `eight_traj_gen.cpp`, is used to generate the eight-shaped trajectory that the robot must follow.

First, the two vectors contained in the `GenerateDesiredPathService.srv` message are populated with the coordinates of all the points of the trajectory.
Then, as soon as the service is called during the initialization (`Prepare()` function) of both controllers, the complete trajectory is made available also to them.

6.3. Nodes

6.3.1 `diffdrive_kin_sim_node`

This node is subscribed to the `/robot_input` topic and reads the angular velocities of the wheels.
Given those it simulates the movement of the robot and publishes the new position on both `/robot_state` and `/odom` topics. These two topics are kept separated in order to accommodate both controllers.

The actual simulator is an object of the `diffdrive_kin_ode` class, which is initialized with

```
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 computing the angular velocities of the two wheels given the current position of the robot and its next point in the trajectory, using a custom trajectory tracking controller.

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 linearisation law (implemented in the `diffdrive_kin_fblin` class) to compute both the linear and the angular velocities (v, ω) 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`

Similarly to the custom controller explained before, this node computes the angular velocities of the two wheels.

This node is used to interface the DWA library in ROS with the custom simulator (in order to compute the angular velocities of the two wheels, just like the previous controller). This is due to the fact that natively DWA is used inside ROS Navigation Stack, thus it expects an odometry source and a costmap.

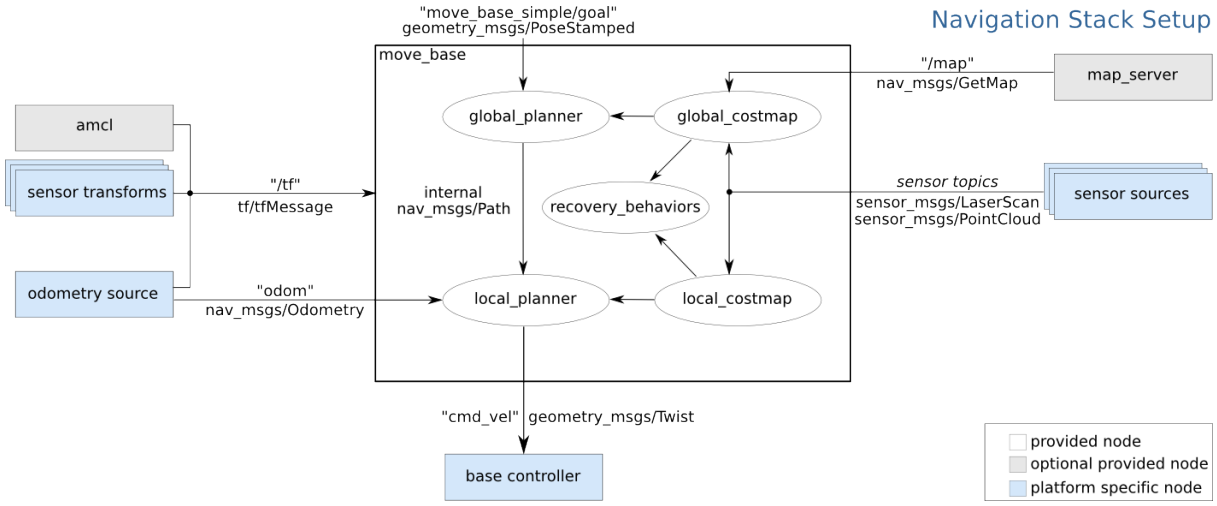


Figure 4: ROS Navigation Stack.

In this specific implementation of this experiment only a global costmap and a planner (DWA) are used, for the reasons explained before.

During the initialization phase the service is queried. The trajectory is received and one point every `skipped_goals` is set as the new plan for DWA (note: the plan here is made of just one point).

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 of the two wheels 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_local_planner/dwa_planner.cpp`
- `dwa_local_planner/dwa_planner_ros.cpp`

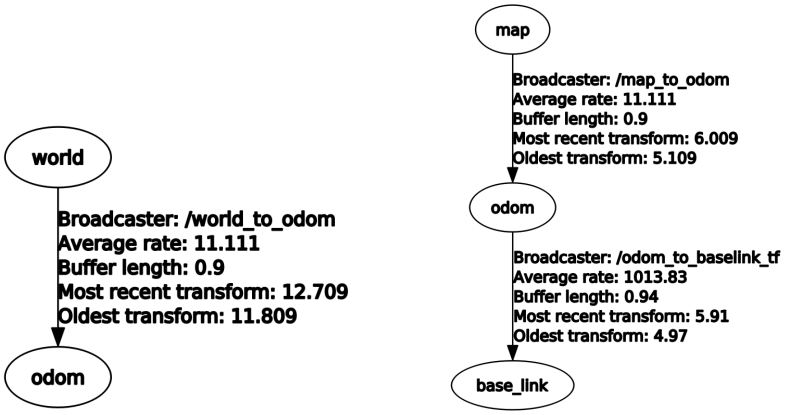
6.3.4 `odom_to_baselink_tf_node`

This is a very simple node. It links the `odom` and the `base_link` coordinate frames through a dynamic tf, since the robot footprint (which is in the `base_link` frame) must be somehow related to the environment (the robot starts moving in the `odom` frame).

6.4. Frames

Here is a list of all the frames, with a short description:

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



(a) Frames of the custom controller. (b) Frames of DWA standalone.

Figure 5: Frames (obtained with `rqt_tf_tree`) of the two approaches.

6.5. Topics

Here is a list of the topics used by nodes for publishing or subscribing, with a short description :

- `/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 communicate the odometry information of the robot to the custom controller;
  - publishers:
    - `diffdrive_kin_sim`
  - subscribers:
    - `diffdrive_kin_trajctrl`

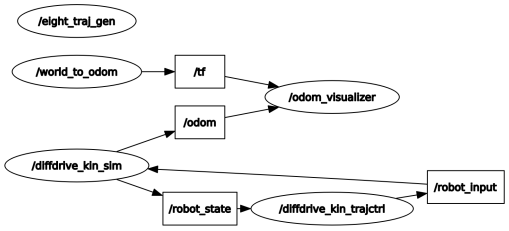
- `/controller_state`, used to publish the details about the controller for visualization purposes;
  - publishers:
    - `diffdrive_kin_trajctrl`
    - `diffdrive_dwa_trajctrl`
  - subscribers:
    - *none*
- `/robot_input`, used to communicate (ω_r, ω_l) computed by the controllers;
  - publishers:
    - `diffdrive_kin_trajctrl`
    - `diffdrive_dwa_trajctrl`
  - subscribers:
    - `diffdrive_kin_sim`

6.6. Launch files

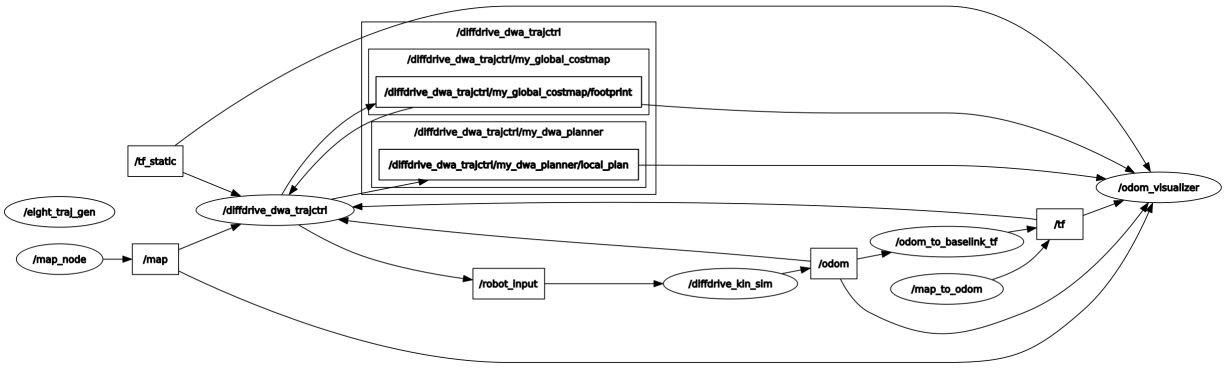
Here is a list of the two launch files, with a short description:

- `diffdrive_kin_trajctrl.launch`, used to simulate the behavior of the robot with the custom controller by starting the following nodes:
  - `diffdrive_kin_sim` (simulator)
  - `diffdrive_kin_trajctrl` (trajectory tracking controller)
  - `eight_traj_gen` (trajectory generator)
  - `world_to_odom` (static tf linking `world` and `odom` frames)
  - `odom_visualizer` (*Rviz* node to visualize real-time data)

- `diffdrive_dwa_trajctrl.launch`, used to simulate the behavior of the robot with DWA by starting the following nodes:
  - `diffdrive_kin_sim` (simulator)
  - `diffdrive_dwa_trajctrl` (DWA)
  - `eight_traj_gen` (trajectory generator)
  - `odom_to_baselink_tf` (dynamic tf linking `odom` and `base_link` frames)
  - `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) Architecture of the custom controller.



(b) Architecture of DWA standalone.

Figure 6: Architecture (obtained with `rqt_graph`) 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. Some are strictly required for the implementation to be started, others can be tuned to slightly change the behavior of the robot when following the trajectory.

In this section the most relevant ones are listed, with a short description:

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

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

- `diffdrive_dwa_trajctrl/config/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`, maximum 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. Parameters tuning

7.2.1 Eight-shaped trajectory

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

$$\begin{cases} x = \mathtt{a} \cdot \sin(\mathtt{w} \cdot t) \\ y = \mathtt{a} \cdot \sin(\mathtt{w} \cdot t) \cdot \cos(\mathtt{w} \cdot t) \end{cases}$$

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

Here is what happens when `a` is increased (the eight-shape becomes larger, look at the scale):

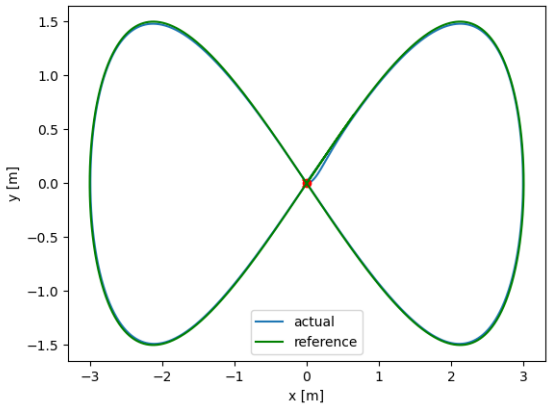


Figure 7: Trajectory when `a = 3.0`.

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

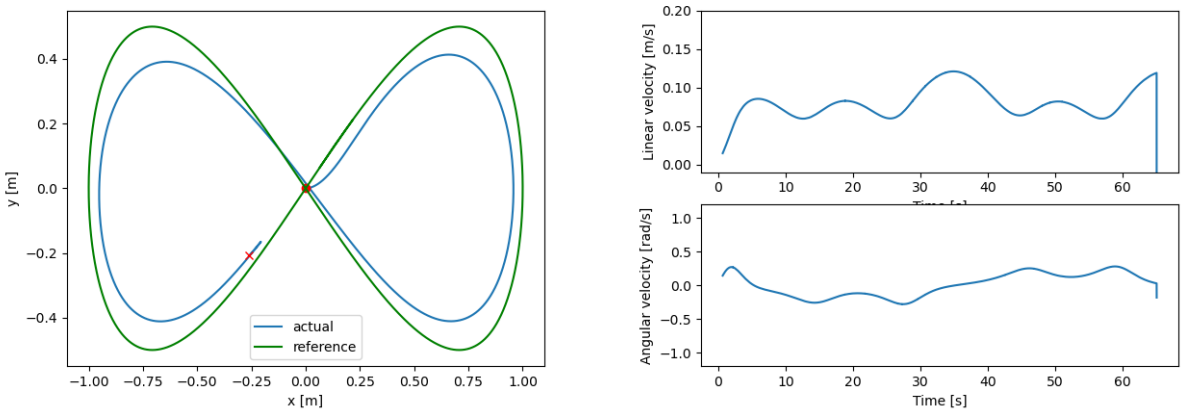
Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

Figure 8: PID parameters tuning.

Various tests have been carried out to achieve the optimal behavior, 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 $K_p = 0.8$, $K_i = 0.8$, $K_d = 0$.

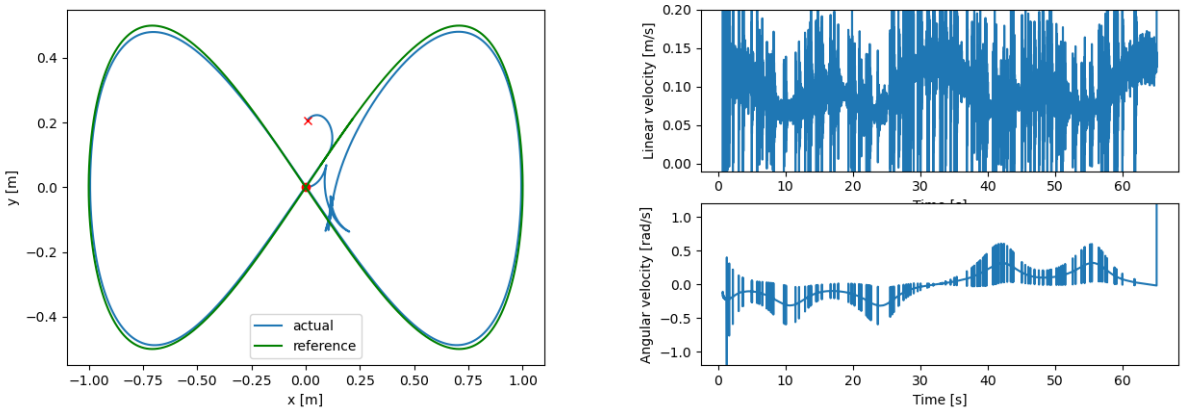
Here are two examples of poor performance when the PID parameters are not correctly tuned:



(a) Trajectory.

(b) Linear and angular velocities.

Figure 9: Behavior with $K_p = 0.2, K_i = 0.8, K_d = 0.0$.



(a) Trajectory.

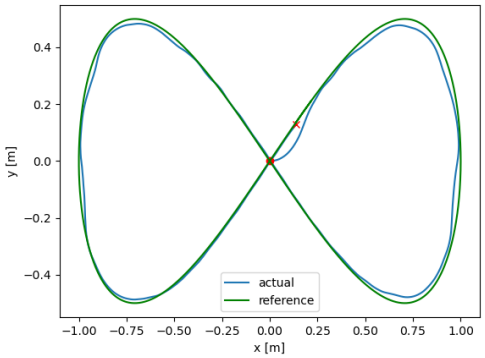
(b) Linear and angular velocities.

Figure 10: Behavior with $K_p = 0.8, K_i = 3.2, K_d = 1.0$.

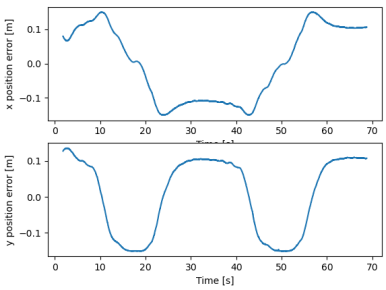
7.2.3 DWA

When the trajectory is generated, the eight shape is discretized over numerous points.
Due to their large number, they are very close to each other. This makes the robot misbehave as soon as a new goal is set in DWA: the distance between two consecutive goals must be higher. 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.

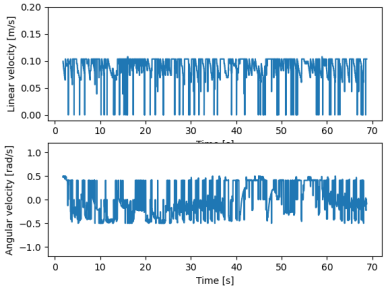
If `skipped_goals = 0` is set, then the number of spikes in the velocities of the robot increases:



(a) Trajectory.



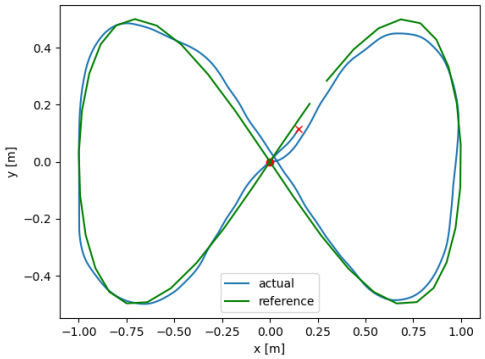
(b) Position Error.



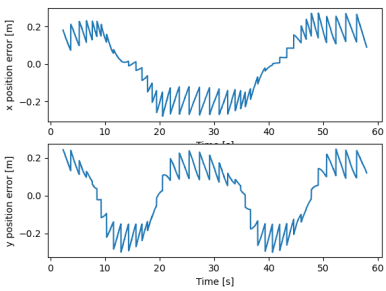
(c) Linear and angular velocities.

Figure 11: Behavior with `skipped_goals = 0`.

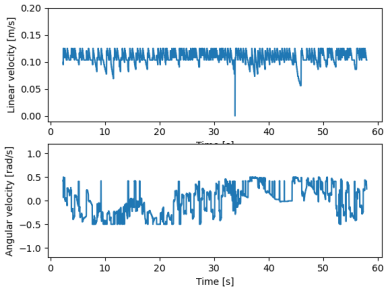
If instead `skipped_goals = 150` is set, then the error in the position while following the trajectory increases:



(a) Trajectory.



(b) Position Error.



(c) Linear and angular velocities.

Figure 12: Behavior with `skipped_goals = 150`.

Since the robot to be simulated is a differential drive one, the following parameters must be 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 chosen value for it is 10.

Since DWA discretely samples 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 the computational effort needed to perform the simulation, 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

While running the project, the movement of the robot can be visualized in real time in an *Rviz* window. However, more detailed results can be obtained by first recording all the data while the robot moves and then executing a custom Python script.

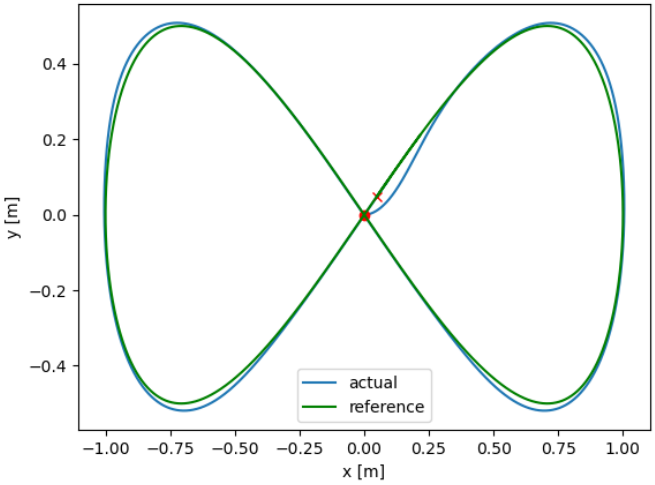
The script is able to plot the followings:

- the full eight-shaped trajectory of the robot, together with the reference;
- the linear and angular velocities of the robot;
- the angular velocities of the two wheels;
- the pose of the robot, so its coordinates and its heading;
- the position error with respect to the reference trajectory.

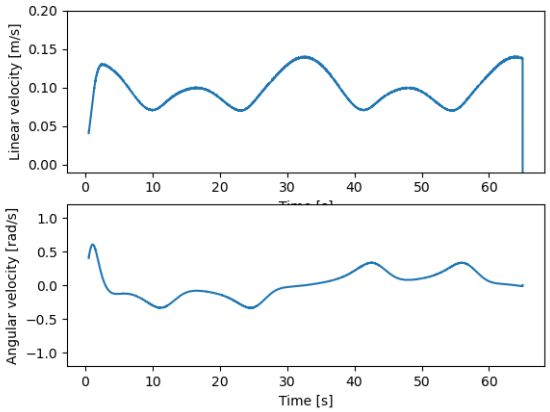
In addition to that, there is a second script that compares the data from two different recordings by displayig the two results in the same plots.

8.1. Tuned trajectory tracking controller

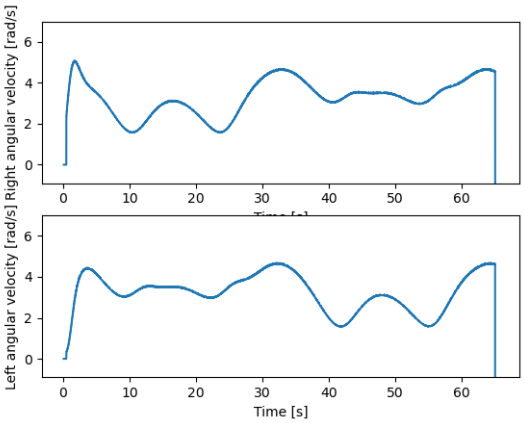
After tuning all the parameters to their optimal values, this is the final result obtained with the trajectory tracking controller:



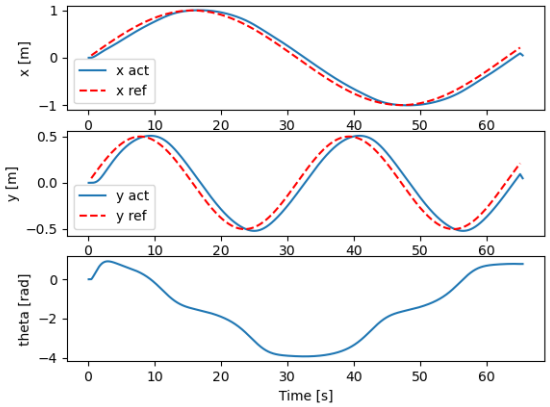
(a) Trajectory.



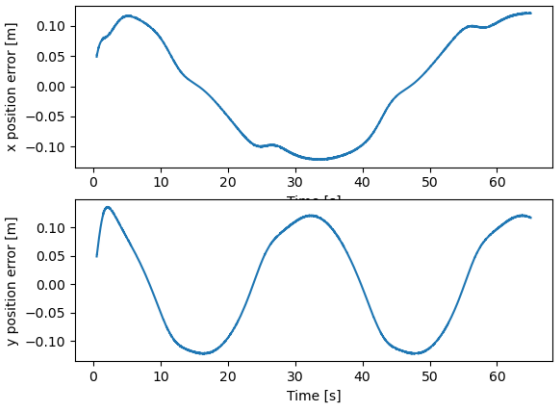
(b) Linear and angular velocities.



(c) Wheels velocities.



(d) Pose.

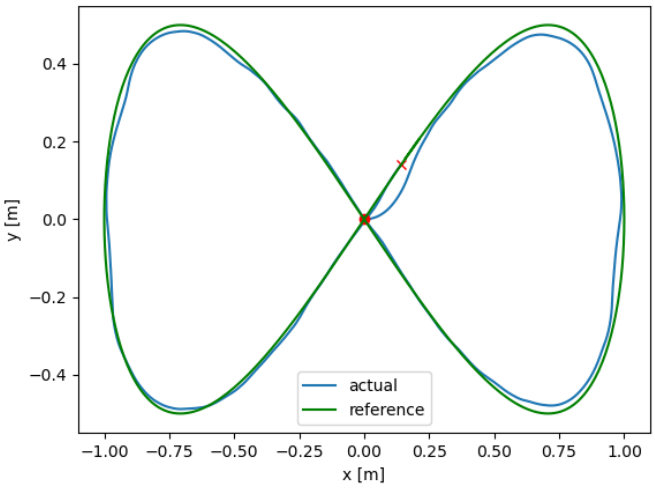


(e) Position Error.

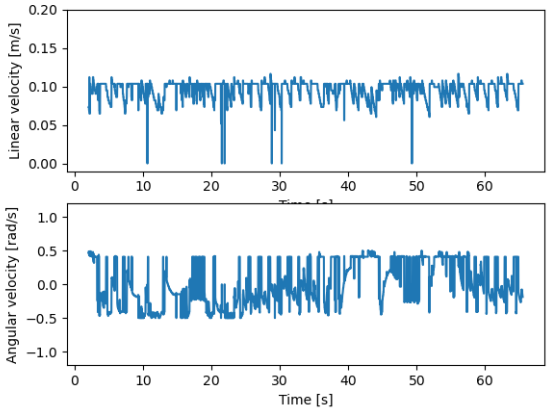
Figure 13: Behavior with the tuned custom controller.

8.2. Tuned DWA

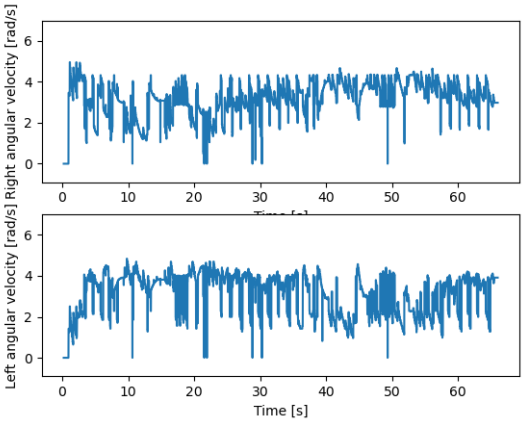
After tuning all the parameters to their optimal values, this is the final result obtained with DWA:



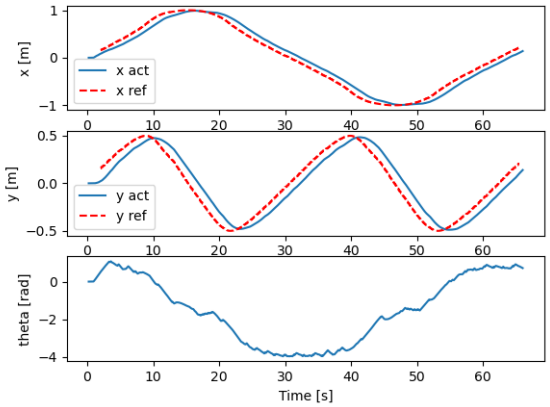
(a) Trajectory.



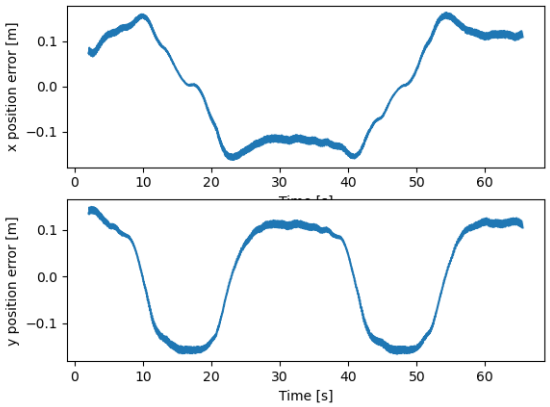
(b) Linear and angular velocities.



(c) Wheels velocities.



(d) Pose.



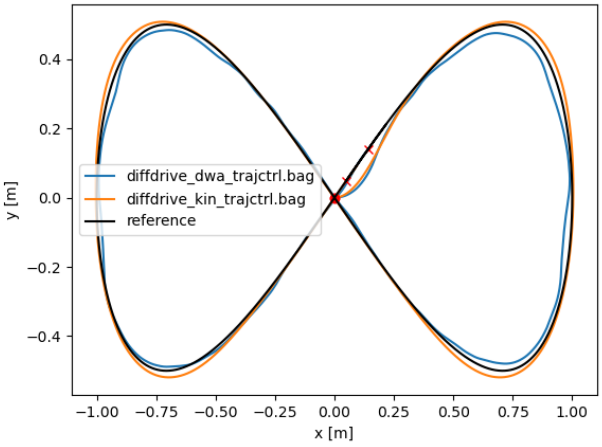
(e) Position Error.

Figure 14: Behavior with tuned DWA.

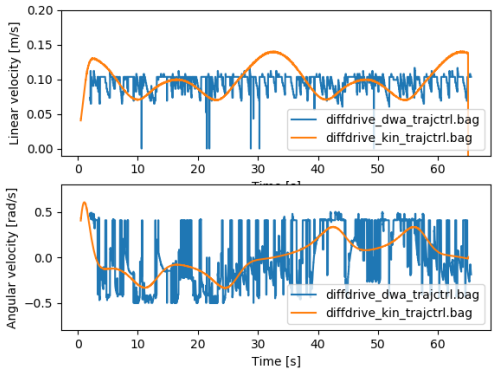
8.3. Comparison

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 controllers 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 is followed in a less precise way.

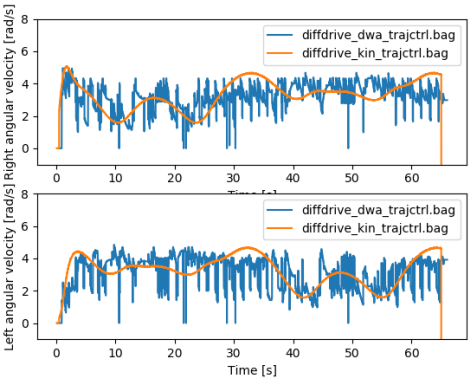
Below there is a direct comparison between the behavior of the trajectory tracking controller and DWA.



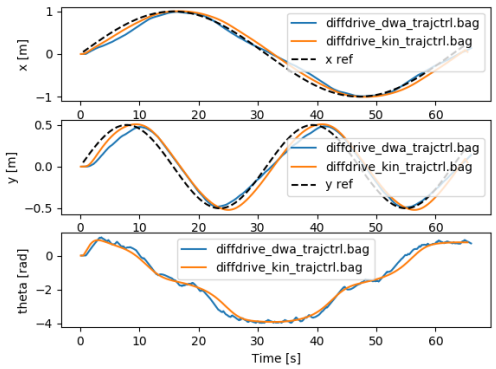
(a) Trajectory: comparison.



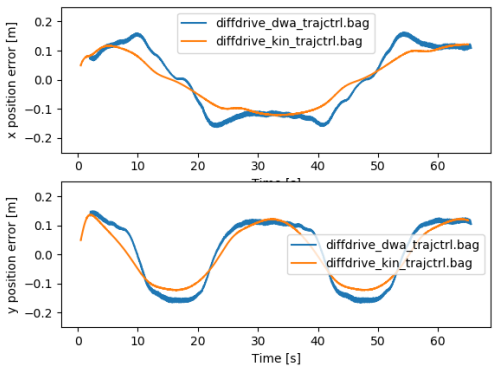
(b) Linear and angular velocities: comparison.



(c) Wheels velocities: comparison.



(d) Pose: comparison.



(e) Position Error: comparison.

Figure 15: Comparison between the two approaches.

9. Faced issues

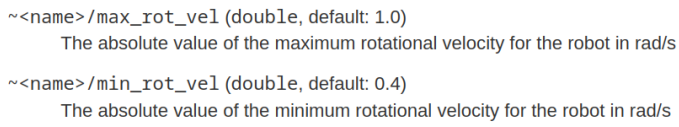
9.1. Deprecated parameters

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 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 the root of the problem when tuning parameters and seeing no difference in the robot's 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`, which is now deprecated and has been renamed into `min_vel_theta`;
- `max_rot_vel`, which is now deprecated and has been renamed into `max_vel_theta`.

Looking at the ROS Wiki, the parameters still have the old (wrong) name, while the source code of the library contains some hints about the (unadvertised - outside of `nav_core`) deprecation:



(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 16: 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 is 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 not working anymore due to the use of deprecated libraries (i.e. `tf` is the one mentioned in the example, while `tf2` is the actual supported one).

For instance, the following snippet is the one in the ROS Wiki (not working):

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <dwa_local_planner/dwa_planner_ros.h>

...

tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS costmap("my_costmap", tf);

dwa_local_planner::DWAPlannerROS dp;
dp.initialize("my_dwa_planner", &tf, &costmap);
```

[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

Instead, the code above must be modified as follow to correctly work:

```
#include <tf2_ros/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <dwa_local_planner/dwa_planner_ros.h>

...

tf2_ros::Buffer tfBuffer(ros::Duration(10));
tf2_ros::TransformListener tfListener(tfBuffer);
costmap_2d::Costmap2DROS my_global_costmap("my_global_costmap", tfBuffer);
my_global_costmap.start();

dwa_local_planner::DWAPlannerROS dp;
dp.initialize("my_dwa_planner", &tfBuffer, &my_global_costmap);
```

9.3. Multiple goals

In a standard scenario DWA needs only a single final goal and computes the full trajectory on its own.
In this project the trajectory is predefined and must be explicitly forced. This is done by continuously changing the goal, otherwise the robot would go straight to the final point of the overall eight-shaped trajectory skipping all the intermediate points.

The issue here is that a too dense vector of goal points would lead to an unsteady profile in the velocities, while a too sparse vector would lead to a large deviation from the reference trajectory.
Therefore a suitable value for the density of those points must have been searched to achieve a correct behavior.

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*.
For the following commands you are expected to already have Ubuntu 18.04 installed.

First, install ROS and all the required packages:

```
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 using bash, add the following line to the end of your ~/.bashrc file:

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

If using zsh instead, add this other line to the end of your ~/.zshrc file:

```
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 eight-shaped trajectory is set.
You will need two open terminals: one for launching the project, the other to record and plot the data.

[terminal 1] Enter the `script/` folder and start recording data:

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

[terminal 2] Start the simulation:

```
roslaunch diffdrive_kin_ctrl diffdrive_kin_trajctrl.launch
```

Wait until the end of the simulation (when the robot stops moving).

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

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

[terminal 1] Visualize the recorded data:

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

10.3.2 DWA

This simulation shows the behavior of DWA when an eight-shaped trajectory is set.
You will need two open terminals: one for launching the project, the other to record and plot the data.

[terminal 1] Enter the script/ folder and start recording data:

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

[terminal 2] Start the simulation:

```
roslaunch diffdrive_dwa_ctrl diffdrive_dwa_trajctrl.launch
```

Wait until the end of the simulation (when the robot stops moving).

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

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

[terminal 1] Visualize the recorded data:

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

10.3.3 Compare two simulations

If you wish, you can also compare the results of the two different simulations.

Given the two bag files already recorded as written above, visualize the comparison (specify the DWA bag first):

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

11. Conclusion

The project compares two approaches to make a robot follow a predefined trajectory: a custom controller, which uses a PID and a linearisation law, and DWA in its ROS implementation.

It has been shown how the two techniques behave in the particular case of following an eight-shaped trajectory in an ideal flat map with no obstacles. The presented results are the output of an optimal parameters configuration. Vastly different (worse) results can be obtained by changing, even slightly, those parameters.

Regarding the trajectory tracking aspect only, the two methods behave in a similar way even though the custom controller performs moderately better than DWA. However, the most noticeable difference is in the smoothness

of the plots of linear and angular velocities: the custom controller produces velocities that are much smoother and realistic than those produced by DWA.

In conclusion, this project allows the user to simulate two different approaches in a quick and easy way, while also exposing a lot of tunable parameters. This way the user is able to try out different behaviors at the change of those parameters, which have been extensively documented above.