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

This project aims at helping a student attending the Control of Mobile Robots course by providing a simple case study which compares the behavior of a robot controlled with DWA and of one controlled with a trajectory tracking control law.

The software simulates the robot, using its kinematic model, and implements the two controllers. Everything runs on ROS Melodic on Ubuntu 18.04 LTS.

2. DWA overview

This section is basically a brief overview of the Dynamic Window Approach (DWA) as presented in the original paper[1].

DWA is an approach to perform collision avoidance in mobile robots, while dealing with the constraints imposed by limited velocities and accelerations.

2.1. Search Space

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.

The search space of the possible velocities is further reduced in other two steps.
First, we consider only all the admissible velocities, which correspond to the velocities allowing the robot to stop before it reaches the closest obstacle on the corresponding curvature.
Then, we leave out all the velocities that can't be reached within a short time interval given the limited accelerations of the robot.

Starting from a sequence of n future time intervals, DWA performs a forward simulation of different values for the velocities keeping them constant for those time intervals.
As a result, the simulated trajectories are all circular arcs, and the most suitable one (we will go into details when talking about the cost function) is selected.

2.2. Optimization

The remaining velocities are fed into the following objective function to maximize it:

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

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).

3. DWA in ROS

3.1. From ROS wiki

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

[1] D. Fox, W. Burgard, S. Thrun (1997) *The Dynamic Window Approach to Collision Avoidance*
[2] https://wiki.ros.org/dwa_local_planner

As stated in the ROS Wiki:

> "The dwa_local_planner 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] within the navigation stack.
In the Wiki there is the following cost function to score each trajectory:

> 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].

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 in our `diffdrive_traj_ctrl.cpp`, line 78;
2. `DWAPlannerROS::dwaComputeVelocityCommands(...)`[6] called from the function at step 1, line 302;
3. `DWAPlanner::findBestPath(...)`[7] called from the function at step 2, line 209;
4. `SimpleScoredSamplingPlanner::findBestTrajectory(...)`[8] called from the function at step 3, line 317;
5. `SimpleScoredSamplingPlanner::scoreTrajectory(...)`[9] called from the function at step 4, 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:

- `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.

[3]https://wiki.ros.org/move_base
[4]https://docs.ros.org/en/melodic/api/dwa_local_planner/html/dwa__planner_8cpp_source.html (line 184)
[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.htm

4. Setup of the experiment

4.1. 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.

4.2. 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.

4.3. 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.

5. Implementation

5.1. 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.

5.2. Service

5.2.1 `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.

5.3. Nodes

5.3.1 `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.

5.3.2 `diffdrive_kin_trajctrl_node.cpp`

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.

5.3.3 `diffdrive_dwa_trajctrl_node.cpp`

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_r) velocities are computed and published on the `/robot_input` and `/controller_state` topics.
The implementaion of DWA and all the functions that have been used are provided by the following two files in the ROS library:
(add navigation stack image)

5.3.4 `odom_to_baselink_tf_node.cpp`

...

5.4. Topics

...

5.5. Parameters (YAML)

...

- screen rqt_graph di quando usiamo il nostro controllore e di quando usiamo dwa (messe di fianco) - subsection con service che discretizza la traiettoria ad 8 - fare una lista di cosa accade dal rosrun al movimento del robot (menzionando il contenuto dei launch file)

6. Parameters tuning

... (nuber of points when discretizing the trajectory: t+=0.001... MAKE IT PARAMETER!!)

7. Experimental Results

... (plots of the bags + plots of the comparison with the custom script)

8. Encountered problems

... (deprecated parameters name in the official doc: put screen of the doc + screen of the comments in the code of the library)

9. Usage of the code

...

10. Conclusions

...