Lab 07-08-09

Control and obstacle avoidance with DWA

Objectives

- Develop a ROS 2 package to run control algorithms on a mobile robot.
- Obstacle avoidance and dynamic goal chasing with Dynamic Window Approach.
- Test the developed algorithms on the real robots following an AprilTag detected through RGB camera.

Requirements

- DWA lecture concepts (slides)
- Python scripts explained during lectures (DWA), present in the GitHub repository SESASR-Course/planning_control_methods (Controllers module)
- ROS 2 Humble
- The turtlebot3_simulation package (update from GitHub if using Ignition)
- Camera packages included in turtlebot3_perception

Exercise

You shall develop a navigation pipeline to follow a moving target. The following algorithm should be based on a DWA algorithm for obstacle avoidance and dynamic goal chasing. The goal will be detected visually moving an AprilTag handled by you in front of the robot.

Task 1 [3 points]

Starting from the **Dynamic Window Approach** scripts that were presented during lectures, implement inside a ROS 2 node a Dynamic Window Approach for obstacle avoidance using the original objective function of the algorithm:

$$J = \alpha \cdot heading + \beta \cdot vel + \gamma \cdot dist_{obst}$$

Where α, β, γ are design parameters of the algorithm that should be carefully tuned to optimize the performance of the algorithm. The `heading` term can be computed using the estimated X-Y coordinate of the detected robot, or by minimizing the sensed bearing. The controller should run at **15 Hz**.

Hint: use a **ROS 2 timer** to implement the main sequence of operations of the DWA, as done in the function `go_to_pose()` in the provided example. If the goal pose is not set or if sensor data is not available, the callback function should simply return. Otherwise, the callback function should perform the sequence of actions of the function `go_to_pose()`.
Hint: An explicit while loop will block callbacks execution in ROS! Instead, the callback function of the timer called at a constant rate acts exactly as a while loop!

Obstacles are not provided as a list of items in the map, you will use LiDAR scan ranges as obstacles perception source. Hence:

- **Implement a function to filter the scan ranges**:
    - Remove NaN, Inf or irregular values. Assign the minimum value of the LiDAR measurements to NaNs and the maximum to Inf.
    - Only consider distance measures up to 3.5 meters
    - 270 obstacles measurements are computationally expensive! Filter the total amount of ranges to `num_ranges` values in the range `[12 - 30]` (up to your choice) Only consider the minimum distance perceived for each angular sector.
    - Determine the obstacle position in [x-y] coordinate given the robot pose and the laser scan ranges obtained.

- **Implement a safety mechanism** to stop the robot and avoid collisions. Considering the robot shape as circular and a proper radius for the TurtleBot3, immediately stop the robot if a certain collision tolerance (15-25 cm) is not respected by the DWA algorithm. Use laser ranges to implement this functionality.

- **Implement a goal manager functionality** to wait for a goal to be received and change the target of the navigation in whatever instant (also during another goal task).

- **Return the resulting event** of the navigation task, choosing among:
    - *Goal*: task succeeded.
    - *Collision*: if the collision condition is met.
    - *Timeout*: if the navigation task duration overcomes the maximum number of control steps allowed.

- **Provide an intermediate feedback** of the navigation task, publishing on a dedicated topic and displaying the current distance to the goal, every N=50 control steps.

Task 2 [2 points]

Implement the following variants of the DWA modifying the objective function:

1. Add a decreasing scoring term **`vel'`** to take in account that the robot should slow down when getting close to the goal (define a distance threshold to start adding this term to the cost)

2. Add a new term in the cost function for the dynamic robot following task. The success of the following task is tied to the capability of the robot to keep the target in good sight, hence, design a new objective function term to keep the target robot at a certain distance and fully visible.

$$J = \alpha \cdot heading + \beta \cdot vel' + \gamma \cdot dist_{obst} + \delta \cdot dist_{target}$$

Task 3 [2 point]

Run the ROS 2 package for the following algorithm on the real robot with a moving target. Record the performance of the navigation system obtained in terms of success rate, trajectory, minimum and average distance from obstacles, and velocity profiles. Test the navigation system on 3 different experiments (defined by 3 different trajectories performed by the target moving among the obstacles in the scene)

Report requirements

- **Provide a concise description and comments** on your **algorithmic design and implementation**, Explain the main implemented functionality, key parameters and conceptual aspects of your program.
    - Example: explain how you defined the cost term and LiDAR scan processing for DWA, etc.

- Briefly comment the main **structure of your ROS 2 program** (do not copy your code in the report, just highlights the main elements and the workflow: nodes, publishers/subscribers to relevant topics and parameters)

- **Results [Task 1 - 2]**: **Run your navigation algorithm in the simulation environment.**

    - **Record** the `/ground_truth`, `/scan`, `/camera/landmarks`, and `/cmd_vel` topics, then

    - **Make plots** comparing the position and orientation reported in the three topics
        - Plot the (x, y) trajectory on the 2D plane using the robot's pose data from the topics.
        - Plot the command signal profiles (v,w) obtained with the controllers.
        - Comment on the results.

    - **Compute the following metrics** using the data in the ground truth:
        - **Success Rate**: How many times does the robot reach the goal? If failure happens, what type of failure (Collision, Timeout,...)?
        - **Time of tracking [%]**: how long have you been chasing the moving target correctly? Measure the amount of time (percentage over the entire duration of the experiment) during which you did not lose the target robot. Losing the tracking means you are NOT following the target anymore.
        - **Root Mean Square Error (RMSE**) of the target **distance** and **bearing** from the target moving robot, considering as optimal value the distance you define in the objective of the DWA and 0 for the bearing angle.
        - Overall **average and minimum distance [m] from the obstacles**, using the */scan* lidar data.

- **Results [Task 3]**: **Run your navigation algorithm on the real robot.**
    - Record the `/odom`, `/camera/landmarks`, `/scan`, and `/cmd_vel` topics.
    - Make plots comparing the position and orientation reported in the two topics. For `/odom` you can subtract the first pose to all the other poses to align to the origin. Then:
        - Plot the (x, y) trajectory on the 2D plane using the robot's pose data from the three topics.

    - Plot the command signal profiles (v,w) obtained with the controllers.
    - Comment on the results.
  - Compute the same metrics of Task 1-2 whenever it is possible to use /odom instead of the ground truth position of the robot.

How to test your algorithms

Simulation Environment

A simulation is provided to ease the development and testing of your code.

If you are using **Ignition** simulation, please download the newest version from GitHub (https://github.com/SESASR-Course/turtlebot3_simulations.git).

To launch the simulation run the following command in a workspace with `turtlebot3_simulations` package:

```
ros2 launch turtlebot3_gazebo project.launch.py
```

In Gazebo, the moving target will be replaced by a moving object. You can find the position of the moving target in the topic **`/dynamic_goal_pose`** with a message of type **`nav_msgs/Odometry`**. (the moving object is a red cube, sometimes it may seem stuck but if you check the pose on the topic it is actually moving)

Real Robot

With real robots, AprilTag will be handled by you in front of the robot. You will detect the tag using the RGB-D camera mounted on the robot. From the images, range and bearing data are already computed and published on a topic, you should:

- Take the detected tag range and bearing from the topic **`/camera/landmarks`** with message type **`landmark_msgs/LandmarkArray`**.
- Transform the range and bearing measurements into X-Y coordinates expressed in the frame odom to have the correct goal_pose

Run AprilTag detector

1. In two different terminals start the camera driver and the landmark detector using the following commands

   ```
   # Terminal 1
   ros2 launch turtlebot3_perception camera.launch.py

   # Terminal 2
   ros2 launch turtlebot3_perception apriltag.launch.py
   ```

2. If AprilTags are visible, they will be published on topic `/camera/landmarks` at approximately 6 Hz.