Project activity

Planning and control methods

Objectives

- Develop a ROS 2 package to run planning and control algorithms on a mobile robot.
- Plan a path to the goal of the robot in the map and generate control commands to track the path.
- Obstacle avoidance and dynamic goal chasing with Dynamic Window Approach.
- Test the develop algorithms on the real robots, coupling two different robot behaviours in a unique navigation mission.

Requirements

- The Python scripts explained during lectures (DWA, A*, Pure Pursuit), present in the Portale della Didattica (materiale/lecture_notebooks/…)

Project Organization

Team groups will be divided to carry out two different Tracks, A and B. The division will be made according to Groups numbers, and such that two groups will perform a combined final experimentation but can work independently and validate their work independently.
Example: Group n and n+1 will carry out Track A and B, choosing the respective preferred track among them.

Team A Track: Path Planning and Tracking

Main objective: develop a navigation pipeline to efficiently plan a safe path from the pose of the robot to an assigned goal, considering the obstacles present in the map and generating commands to stay on the computed path.

Task 1 [3 points]

Starting from the Path Planning scripts that were presented during lectures, implement in a ROS 2 package an A* algorithm to generate a global path to the goal in a given map.

The planner should work accordingly to the following requirements:

- Load the map image and transform it into a gridmap representation with a reasonable scaling resolution factor.
- The planner can plan in 8 possible directions.
- Assign a cost value to each cell of the map based on the presence of obstacles. The cost values should be in the range [0,100]. The maximum cost 100 should be given to map cells with obstacles, while an inflation layer with scaled cost values should be implemented to obtain reasonable non-zero cost values in neighboring cells. An example is provided in the function generate_costmap()
- The path must be generated using the overall score function of the A* algorithm f(n) = g(n) + h(n), where g(n) is the cost so far of the path according to the costmap and h(n) is the heuristic to inform the planner about the goal position.
- The heuristic function must be chosen properly for an 8 direction movements planner.

ROS-related implementation details and suggestions:

- Use a ***`Path`*** msg to publish the global path computed on a new topic **`/global_path`**.
- The map will be given.
- Read the map using the **`get_gridmap()`** utils function provided to read map images of different formats. Usually in ROS map configuration files are .yaml files.
- If you want to visualize the costmap used for planning, use an Occupancy_Grid msg to publish the costmap on a **`/global_costmap`** topic (not mandatory).
- The planner node can be invoked once at the beginning or multiple times with a certain fixed frequency of 0.5 Hz to account for replanning necessity.

Task 2 [2 point]

Starting from the Pure Pursuit scripts that were presented during lectures, implement in a ROS 2 package a Pure Pursuit algorithm to track a given global path with kinematically reachable velocity commands for the robot.

The controller should work accordingly to the following requirements:

- **Define a proper lookahead distance** parameter ***L*** and a target velocity v respecting the maximum linear speed of the TurtleBot3 Burger robot.
- **Find the lookahead point** based on *L* on the given path.
- **Implement the angular velocity computation** using the geometry derived from the Pure Pursuit model to move the robot on the target circular trajectories.

- **Implement a proportional control law to compute the acceleration** to reach the target velocity according to the acceleration and velocity limits of the TurtleBot3 Burger robot.
- Try to implement an **Adaptive Pure Pursuit** strategy where the lookahead distance parameter is updated every time step according to the equation below. The parameter l_t is a numerical score that should be tuned accordingly to the situation (consider sharp edges and turns on the path)

$$L_t = v_t \cdot l_t$$

ROS-related implementation details and suggestions:

- Use a Twist msg to publish the generated velocity commands $u = (v, \omega)$ on the **`/cmd_vel`** topic.
- Read the global path from a **`/global_path`** topic where you are publishing the resulting path generated with A*.
- The controller should run at a fixed frequency of 15 Hz.

Task 3 [2 point]

Run the ROS 2 packages for Path Planning and Pure Pursuit on the real robot and record the performance of the navigation system obtained in terms of success rate, trajectory and velocity profiles. Test the navigation system on 3 different scenarios in the map (defined by 3 different initial poses of the robot and goal poses) and provide average results for the metrics obtained.

Team B Track: Obstacle Avoidance and Robot Following

Main objective: develop a navigation pipeline to follow another moving robot. The following algorithm should be based on a DWA-based algorithm for obstacle avoidance and dynamic goal chasing. The target robot position will be detected visually with a tag placed on 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 in the while loop of the function `go_to_pose()`.
Hint: An explicit while loop will block callbacks execution! 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.
  - A LaserSensor class prototype is provided in the utils module.
- **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 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 an exponentially decreasing scoring term in the `vel'` score 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. (Hint: try to stay at a certain distance from the target robot and minimize the relative heading angle.)

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

In Gazebo, the moving robot 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)
With real robots, AprilTag will be placed on the target moving 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.

Task 3 [2 point]

Run the ROS 2 package for the robot following on the real robot trying to follow another moving robot. The motion system of the target robot should be provided by Team A solution, or by tele-operating the robot with the keyboard. 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 robot among the obstacles in the scene) and provide average results for the metrics obtained.

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 design the costmap and the heuristic for A* planner, how do you find the lookahead point for PurePursuit, 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.**
    - Track A: Record the `/ground_truth`, `/global_path`, and /cmd_vel topics.
    - Track B: 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 three topics. (For Track A only: plot the path computed by A* and trajectory performed by the robot).
        - 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:
        - Track A:
            - **Success Rate**: How many times does the robot reach the goal? If failure happens, what type of failure (Collision, Timeout,...)?
            - Overall **distance [m]** and **time of travel [s]** to reach the goal.
            - **Root Mean Square Error (RMSE)** between the trajectory performed by the robot and the trajectory given by the global planner to evaluate the Pure Pursuit path tracking.
        - Track B:
            - **Success Rate**: How many times does the robot reach the goal? If failure happens, what type of failure (Collision, Timeout,...)?
            - Overall **average and minimum distance [m] from the obstacles**, using the */scan* lidar data.
            - **Time of tracking [%]**: how long have you been chasing the moving robot correctly? Measure the amount of time (percentage over the entire duration of the experiment) during which you did not lose the target robot.
            - **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.

- **Results [Task 3]**: **Run your navigation algorithm on the real robot.**
    - Track A: Record the `/odom`, `/global_path`, and `/cmd_vel` topics.
    - Track B: 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. (For Track A only: plot the path computed by A* and trajectory performed by the robot).
        - 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

The following operations shall be executed on your PC.

A simulation is provided to ease the development and testing of your code. (dynamic goal for Task B is available only for Gazebo).

Update your `turtlebot3_simulations` repository as described in the Lab04 text.

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

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

Real Robot

The following operation shall be performed on the robot.

Track A: Load the map

1. Add the map file to your package at the position …
2. Add the map yaml address to the params file of your package to pass the map to the planner

Track B: Install the camera

1. Mount the camera on the robot as you did in Lab04.
2. In the file **`turtlebot3_perception/config/apriltag.yaml`**, modify the parameter **`size`** to the actual size in meters of the AprilTag to follow. Then, rebuild the workspace.
3. 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
   ```
4. If AprilTags are visible, they will be published on topic `/camera/landmarks` at approximately 6 Hz.

Record data of your experiments

Based on your Track, record all the topics that you need to evaluate your experiments using

```
ros2 bag record /topic1 /topic2 ...
```