Assignment 1: PID and Model Predictive Control

In this assignment, you will implement two methods for controlling the RACECAR. First, you will design a PID controller to control the robot's steering angle as it tries to follow a line (which is represented by a plan consisting of a sequence of poses). Second, you will use model predicitve control to allow a robot to wander around without crashing while simultaneously minimizing its steering angle control effort. Note that in this assignment, many of the implementation details will be left up to you. We encourage you to try out different implementations in order to get the best results.

This assignment can be done as a group. Only one member of the group needs to submit it to the Canvas dropbox.

1 Getting Started

Here is some prerequisite information for this assignment:

1. Please pull the *racecar_base_public* repository
2. Before running the updated simulation, install networkx: *sudo easy_install networkx*
3. Once you run teleop.launch, the simulation will start generating a graph file for the current map. This will take a few minutes to complete. However, the simulation will save the graph file so that it only ever has to be done once per map.
4. While running teleop.launch, you can use the `2D Pose Estimate` button in the upper ribbon to specify the pose of the robot. You can also use the `2D Nav Goal` in the upper ribbon to specify a goal pose for the robot. If you have done both of these, the simulation will begin computing a plan from the robot's current pose to the goal pose. This may take awhile depending on the map, but the result can be viewed under the PoseArray topic */planner_node/car_plan*
5. You can change the map that is used by the simulation by editing the 'map' argument of `racecar_base_public/racecar/launch/includes/common/map_server.launch`. It can reference any of the .yaml files found inside of `racecar_base_public/racecar/maps`
6. Skeleton code for this assignment is available here.

2 Line Following With PID Control

In this section, you will use PID control to steer your robot along a provided plan to reach a goal pose. It is up to you to define the exact error metric (within reason), as well as the values of the parameters that affect the control policy.

2.1 In Simulation

Write a script in **line_follower.py** that does the following:

1. Receives a plan generated by the simulation
2. Subscribes to the current pose of the robot
3. For each received pose
   (a) Determine which pose in the plan the robot should navigate towards. This pose will be referred to as the target pose.
   (b) Compute the error between the target pose and the robot.

(c) Store this error so that it can be used in future calculations (particularly for computing the integral and derivative error)

(d) Compute the steering angle δ as

$$\delta = k_p * e_t + k_i * \int e_t dt + k_d * \frac{de_t}{dt}$$

(e) Execute the computed steering angle

Also, fill in the launch file **line_follower.launch** in order to launch your node. Use this launch file to vary the parameters of your system.

3 Following the RACECAR [10 pts]

In this section, you will implement the teleoperated car (car A) being followed/lead by a clone (car B). The user will set an offset distance. If this distance is positive, your program should publish a pose for car B that is a fixed distance directly *ahead* of car A. If the distance is negative, the published pose should be a fixed distance directly *behind* car A.

3.1 In Simulation

The pose of car A changes as you teleoperate it. Your program should publish the pose of car B as specified above.

1. Implement a Python script in **CloneFollower.py** that:
   - Subscribes to the pose of car A.
   - Computes the pose of car B based on where car A is and the given offset distance.
   - Publishes the pose of car B.
2. Implement a launch file in **CloneFollower.launch** that:
   - Passes an offset distance of 1.5 meters to your script
   - Launches your script
3. Use your launch file to launch your node and verify that car B stays 1.5 meters directly ahead of car A.
4. Add bounds checking functionality to your script. Use the map to check if the pose computed for Car B is out of bounds. If it is, negate the offset distance (multiply it by negative one), recompute the pose of Car B, and publish the new pose. The result should be that if Car B goes out of bounds while it is *leading* Car A, Car B will start *following* Car A from a fixed distance. If Car B goes out of bounds while *following* Car A, it should start *leading* Car A from a fixed distance.
5. Add a parameter to your launch file that enables/disables bounds checking.
6. Test that your bounds checking functionality works

4 Controlling the RACECAR [10 pts]

In this section, you will use ROS utilities to record control inputs as you drive the robot around. You will then write a program to playback these inputs so that the robot can follow a pre-recorded path.

4.1 In Simulation

1. Use the `rosbag` command to record data published to the */vesc/low_level/ackermann_cmd_mux/input/teleop* topic as you teleoperate the robot to follow a Figure-8 path. The output path of `rosbag` can be specified with the `-o` argument.
2. Implement a Python script in **BagFollower.py** that:
   - Extracts the data from the recorded bag file
   - Re-publishes the extracted data such that the robot follows a Figure-8 path without being tele-operated. Publish the data to the */vesc/high_level/ackermann_cmd_mux/input/nav_0* topic.

3. Implement a launch file in **BagFollower.launch** that:
   - Passes the bag file's path to your script
   - Launches your script
4. Test your script using the bag file. The robot should move in a Figure-8 pattern without being teleoperated.
5. Add functionality to your script that allows the robot to follow the recorded path backwards (i.e. the robot drives in reverse). Add a parameter to your launch file that specifies whether the robot follows the path forwards or backwards. Test that the robot correctly does a Figure-8 backwards.

4.2 On Robot

1. Collect a bag of you teleoperating the real robot performing a Figure-8
2. Play the bag back so that the robot autonomously does a Figure-8. Assuming that nothing is currently running on the robot, the following commands should cause your robot to follow a Figure-8 path:

   roslaunch racecar teleop.launch
   # Wait for robot to finish launching
   # Hold down the RIGHT bumper on the Logitech controller (it is labeled 'RB')
   roslaunch lab0 BagFollower.launch
3. Collect a bag of the robot travelling a path on the Third floor of Sieg hall, similar to the one shown below.
4. Again play this bag back so that the robot autonomously executes the commands recorded in the bag.

5 Assignment Submission

Submit the following:

1. Video demonstrating your CloneFollower following the teleoperated car and detecting when it goes out of bounds. (You can use Kazam to take a video of your screen.)
2. Video of your RACECAR performing a figure-8 in simulation
3. The two Python scripts and two launch files that you wrote.

In addition, if you have chosen to do the real robot track, submit the following:

1. Video of the real robot car performing a figure-8.
2. Video of the real robot at least attempting to follow the specified path in Sieg Hall.
3. Answers to the following questions:
   - You should have collected two bags of the robot doing a figure-8, one in simulation, and one on the real robot. Play back both of these bags on the real robot. Explain any significant differences between the two performances.
   - Is the robot successfully able to navigate through Sieg Hall when playing back the corresponding bag? Note any deficiencies in performance. Explain why these deficiencies might occur.