# 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 predictive control to allow the 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. The provided skeleton code does have *suggestions* for *some* of these details.

This assignment can be done as a group. Only one member of the group needs to submit it to the Canvas dropbox. All group members' names should be on the document containing answers to the assignment's questions.

## 1 Getting Started

Here is some prerequisite information for this assignment:

1. Please git 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 a while depending on the map (you can view its progress by looking at the terminal where you launched teleop.launch), 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. Note that only the steering angle is being varied - the robot's forward velocity should be constant.

### 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 $\delta$ 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.

Test your system by playing back the provided bag file (lab1/bags/line_follower.bag) when using the 'small_basement' map. The bag file will automatically specify the initial and goal pose - i.e. you don't need to manually specify the initial pose and goal pose in rviz. Choose 3 different sets of PID parameters, where each set specifies a controller that causes the robot to successfully navigate from the start to the goal. Measure the performance of these controllers by recording the error at each iteration of the pose callback. Plot all three data series on a single figure where the y axis is the error and the x axis is the iteration index.

Hint: Try your script out on the simple map first so that it doesn't take a long time to compute a plan. Or, you could have the simulation generate a plan and then create a bag of the plan, which you could then replay whenever you want to test your code.

For more details, see the provided skeleton code.

## 3 Wandering Around with MPC

In this section, you will use the kinematic car model to simulate possible trajectories for your robot to follow. Based on received laser scans, your robot will execute a steering angle control that avoids obstacles while minimizing the steering angle control effort. Note that only the steering angle is being varied - the robot's forward velocity should be constant.

### 3.1 In Simulation

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

1. Rolls out n trajectories and caches them
2. Vizualizes the final pose of each trajectory as the car moves around in the world, such as in Fig. 1 below. This will require subscribing to the pose of the robot.
3. Subscribes to the robot's laser scans
4. Whenever a laser scan is received, spend a fixed amount of time computing the trajectory costs as illustrated in Fig. 2
   (a) Compute the cost of the first step of each trajectory.
   (b) If there is remaining computation time, compute the cost of the second step of each trajectory and add them to the cost of the first trajectory steps.
   (c) Continue this process until the allowed amount of computation time has expired.
5. Chooses the steering angle that corresponds to the least costly trajectory and executes it

Given a pose in a trajectory to compute the cost for, initialize the cost as the magnitude of the steering angle that corresponds to that trajectory. Then consider the line that emanates from the robot to the pose. Calculate the angle $\theta$ between this line and the x-axis of the robot, as shown in Fig. 3. Then find the laser ray that corresponds to $\theta$. If the laser ray measurement is closer than the pose, then a heavy penalty should be added. If the laser ray measurement is further away than the pose, then the pose does not incur additional cost.

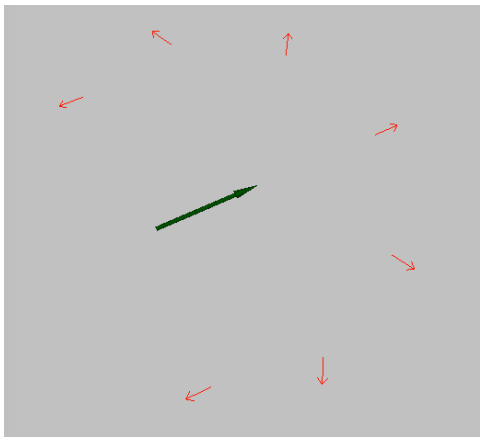

**Figure 1:** The robot in green and the final pose of each trajectory in red

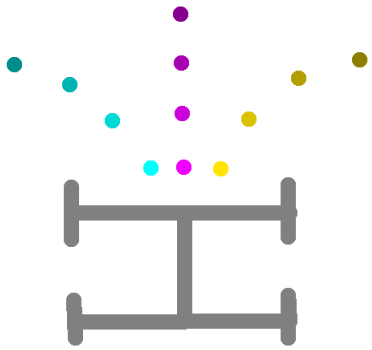

**Figure 2:** Three different trajectories. Earlier steps of the trajectories are drawn in lighter shades, while later steps of the trajectories are drawn in darker shades. Under the specified any-time algorithm scheme, the costs of the lighter shaded poses are first computed, and the cost of the darker shaded poses are only computed if there is computation time remaining.

Hint: To check that you are calculating the costs correctly, initialize your robot in different poses and print out the costs of each trajectory (but don't publish any controls so that your robot doesn't actually move). Check if these costs make sense.

Finally, implement the **laser_wanderer.launch** file to launch your node.

Test your system by playing back the provided bag file (lab1/bags/laser_wanderer.bag) when using the 'no-end-floor4_corridor' map. The bag file will automatically specify the initial pose - i.e. you don't need to manually specify the initial pose in rviz.

For more details, see the provided skeleton code.

### 3.2 On Robot

The robot is no longer a point in space - it now has width! Modify your algorithm to account for this.

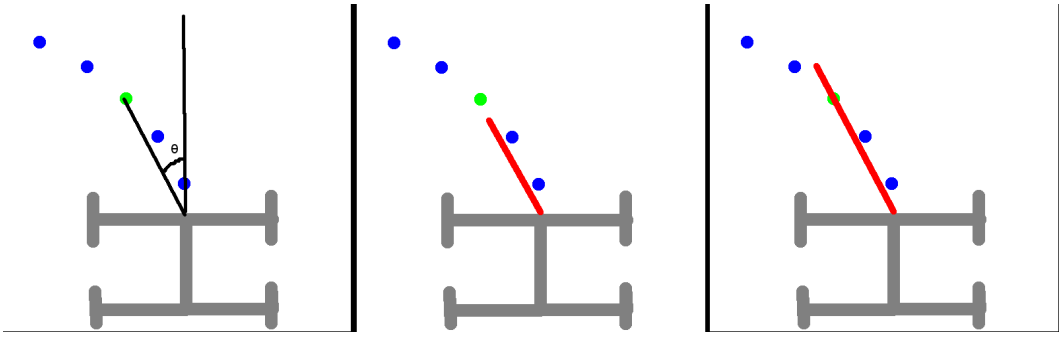


**Figure 3:** Left: The cost of the green pose is being computed. Middle: The corresponding laser ray in red is closer than the pose, so the pose receives a large penalty. Right: The corresponding laser ray is farther than the pose, so no additional penalty is incurred.

## 4 Assignment Submission

Submit the following:

### 4.1 Line Following with PID Control

1. Answer: What exactly was your error metric? How exactly did you choose which point the robot should navigate towards?
2. Answer: What are the 3 sets of PID parameters that you choose? Which set of PID parameters performed best? Why do you think this is the case?
3. Answer: What is missing that is stopping us from implementing this on the real robot?
4. The error plot described at the end of Section 2.1
5. A video of your best PID controller following the path generated by the provided bag file
6. Your line_follower.py script and line_follower.launch launch file

### 4.2 Wandering Around with MPC

1. Answer: Describe the parameter values that you chose (e.g. T, delta_incr, etc.). Describe any other design choices that you made.
2. A video of your robot continuously wandering around the no-end-floor4_corridor map for at least one minute when initialized by the provided bag. Make sure the robot's laser scan and the vizualization you implemented are visible.
3. Your laser_wanderer.py script and laser_wanderer.launch launch file

In addition, if you are working with the real robot, answer the following:
Describe the changes you made when transferring from the simulated robot to the real robot, if any.

You will also demo your robot wandering around the EE basement. The requirement will be that your robot should continuously wander around without crashing for at least 30 seconds.