Assignment 2: Control

The goal of this assignment is to get you familiar with several fundamental controllers and their trade-offs within the context of *path tracking* using mobile robots. In the base assignment you will implement two controllers and analyze their differences. In the extra credit section, you can choose to 1) implement alternative error function for PID controller, 2) empower a MPC controller to avoid real obstacles using laser sensor or 3) implement a pure pursuit controller. Each controller is designed to track a predetermined trajectory, but they will face trade-offs in terms of capability, complexity and robustness. By the end of this assignment you will:

- become familiar with strengths and weaknesses of various feedback control strategies.
- become familiar with PID, MPC and (optionally) pure pursuit controller.
- be able to identify the best control strategy for different scenarios.
- have developed skills to analyze control strategies and iteratively improve them.
- gain an appreciation for the power of cost functions.

Please refer to the *Sec.5 Deliverables* before you start coding. Some part of this assignment requires scrupulous notes and careful data collection. A principled approach to observing and analyzing your system will go far in helping you develop robust controllers.

Four member teams should attempt at least one of the extra credit questions.

1 Overview

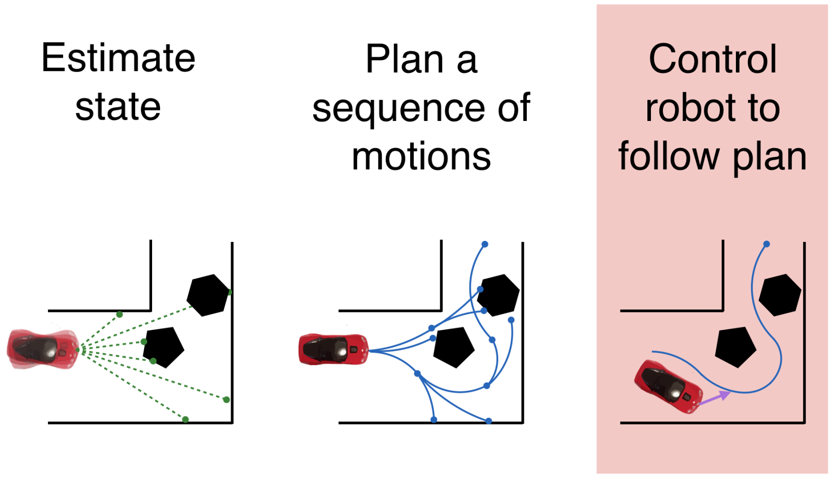

Figure 1: Path tracking on mobile robotics.

In this assignment we focus on one problem: path tracking on mobile robots, illustrated by Figure 1. To help our mobile robot navigate on land following a desired path, we break down the task to two components: motion planning and control. The motion planner generates a *reference trajectory* and the controller tries to follow the *reference trajectory*. The controllers you will implement have been used in DARPA Grand Challenge and DARPA Urban Challenge among many other scenarios.

A controller usually takes in the current pose of the robot and a reference trajectory. For this assignment, our controller will i) takes in the inferred pose returned by the localization module and

ii) takes in a reference trajectory provided by a motion planning module and iii) generate controls that will track the trajectory trajectory. Your implementation will adhere to the diagram in Figure 2.

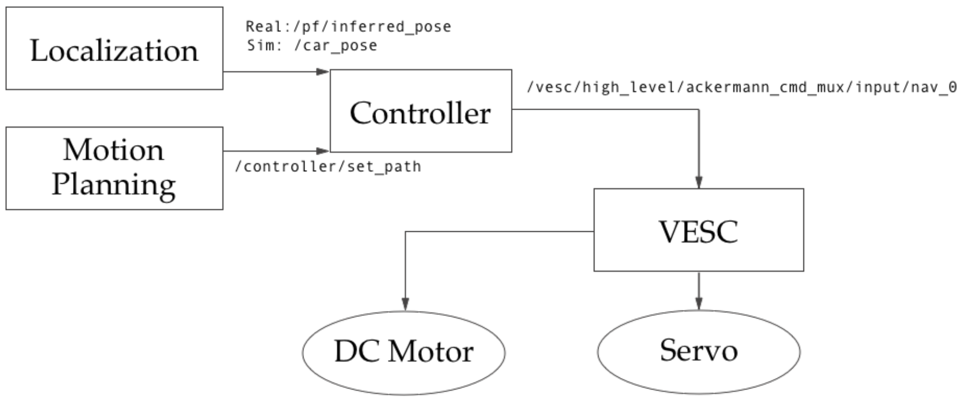

Figure 2: System overview of localization, planer, and controller nodes.

On a high level, a controller seeks to minimize some form of error between the robot's pose p and a reference pose p_{ref}. To implement the PID controller and the pure pursuit controller, we define the error using the Cross Track Error (CTE). Cross Track Error, denoted as y_e, equals to the y component of the error vector e_p between the pose of the car p and the reference point p_{ref}. In order to compute the CTE, you must *rotate the error vector into the car frame*, using the inverse rotation matrix R^T:

$$e_p = p - p_{\text{ref}}$$

$$e_p = \begin{bmatrix} x_e \\ y_e \end{bmatrix} = R^T(\theta_{\text{ref}}) \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_{\text{ref}} \\ y_{\text{ref}} \end{bmatrix} \right)$$

See Figure 13.20 in *Modern Robotics* for a useful visual explanation of the transformation.

2 PID Controller

The proportional-integral-derivative (PID) controller is a feedback control mechanism that is simple to implement and widely used. Here you will implement the PID controller in `pid.py`, tune it, and evaluate its performance on a series of simple control tasks.

A PID controller is usually defined as:

$$u(t) = K_{\text{p}}e(t) + K_{\text{i}} \int_0^t e(t')\,dt' + K_{\text{d}}\frac{de(t)}{dt}$$

where K_{p}, K_{i}, and K_{d}, are all non-negative coefficients for the proportional, integral, and derivative terms of the error $e(t)$ at time t. The weighted sum computes your control $u(t)$, which in the case of the racecar is going to be the steering angle $\delta(t)$. One slight modification we will make, however, is to drop the integral term entirely. Thus you will actually be implementing what is called a *PD Controller*.

Getting start to code by inspecting `controller.py` and `pid.py` and writing your own PID controller and a launch file `launch/pid_controller.launch`.

Once you are confident with the implementation of your controller, use the `runner_script.py` script to test your controller against a set of reference trajectories (right turn, left turn, and circle). You can vary the radian of the arcs of the turns and circle in the code to test the robustness of your controller, as well as start with different initial poses to see how your controller responds. A principled approach to finding robust K$_p$ and K_{d} values is described in this article.

One way of computing $\frac{de(t)}{dt}$ is to use $\sin(\psi_e)$ where ψ_e is the heading angle of the car with respect to the centerline. Another is to use the numerical differentiation. You will compare the two methods, select one for your final implementation and justify your selection in the write-up.

We provided an IPython notebook `bags/Controller Plotting.ipynb` that might be useful for generating plots that describe the performance of your controller. You can use this as a starting point for analyzing your controller, and document your tuning process. To use the notebook, you will need to install Jupyter on your machine (if it is not already there). See instruction from Jupyter website here (Note we are using Python 2).

Additional reading can be found in Fast line-following robots part 1: control, Andy Sloane, 2018. This is a brief but helpful blog post which describes a number of path tracking controllers, including PD control. You can use the sliders in the blog post to get an intuition for the influence of each gain term on the vehicle's control.

To get full credits on the gain-tuning section, we do not ask for *the perfectly tuned gain* that would have a small error on all path configurations you tried. However, we would like you to try out different gains on different paths. We will ask you in the write up to document your gain-tuning process and justify your final choice of gains.

3 Model Predictive Control

The model predictive controller is decidedly more complex. Give your team ample time for the implementation and testing of this controller.

Although the simple control laws above can be adequate in some driving conditions, additional factors (e.g. nearby obstacles or the slippage of the vehicle) might affect path-tracking. To take into consideration such details, we need to first incorporate such information into a *model* of the vehicle or the environment. The MPC algorithm will use this model to solve a limited time horizon open-loop motion planning problem. Once the action has been determined, the system will execute it and immediately begin updating the state estimation and computing the next action. Such "tight loop" control can allow the system to penalize undesirable paths such as paths with collisions, while rewarding paths that advance the pose of the car closer to the goal. Such "punishments" or "rewards" are decided by the cost function.

Implement your model predictive controller in `mpc.py` and write a launch file for it. In `mpc.py` you will:

- Pre-compute a set of K control trajectories.
- Roll out each control trajectory T steps forward using the kinematic car model from the current pose as specified by your localization module.
- Evaluate each rollout using a cost function that penalizes collisions and rewards states which are closer to the goal.

Writing an effective cost function will require principled design, iteration and testing. Document your process for determining your cost function using the plotting tool (`viz_paths_cmap` in `rosviz.py`). We have provided a number of test cases for you to evaluate your controller's ability to avoid obstacles:

1. `cse022_w_obstacles.yaml`: To be used with `scripts/script_runner.py` option `cse022 real path` both in sim and real. You can use it to avoid a "ghost" object that's in the map, but not in world. You should think how this will affect your particle filter.
2. `sandbox_circle.yaml`: To be used in sim to test obstacle avoidance.
3. `sandbox_2_circle.yaml`: To be used in sim to test obstacle avoidance.

To comprehensively test your system, consider formulating scenarios in which the car must avoid an obstacle. You can create new maps (by editing `sandbox.png`) which contain obstacles. What worked and what didn't? Are there any pathological cases which were difficult for the model predictive controller to solve? In the "Deliverables" section, there are additional questions you should address as you implement and evaluate the controller.

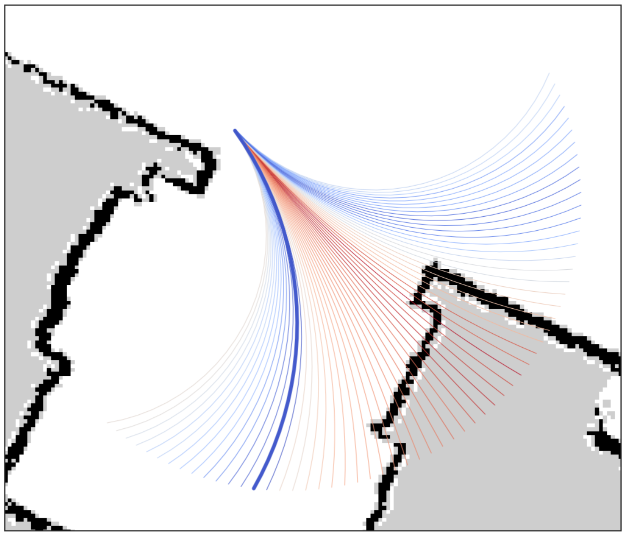

Figure 3: MPC rollouts, colored by cost. Red rollouts are of high cost (from collision) and blue are of low cost.

4 Extra Credit

4.1 Pure Pursuit Control

Pure Pursuit is a classic feedback control mechanism that is frequently used on active mobile robots. As described in *Implementation of the Pure Pursuit Path Tracking Algorithm*:

> Pure pursuit is a tracking algorithm that works by calculating the curvature that will move a vehicle from its current position to some goal position. The whole point of the algorithm is to choose a goal position that is some distance ahead of the vehicle on the path. The name pure pursuit comes from the analogy that we use to describe the method. We tend to think of the vehicle of chasing a point on the path some distance ahead of it - it is pursuing that moving point. That analogy is often used to compare this method to the way humans drive. We tend to look some distance in front of the car and head towards that spot. This lookahead distance changes as we drive to reflect the twist of the road and vision occlusions.

Getting start with implementation from the `purepursuit.py` file and write your own launch file for pid controller. We encourage you to reference this paper when implementing the controller.

You will notice that pure pursuit relies on fewer parameters than the PD controller. Experiment with different lookahead distances and document which distance yielded the most robust control across your test cases. We expect you to evaluate your controller across varying turn radians and speed regimes. Document your testing and provide a table which shows you exhaustively tested your controller across varying configurations.

How does pure pursuit compare against PD control? What tradeoffs would you consider when deciding between controllers?

Reading on the Pure Pursuit algorithm can be found here:

1. *A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles, Brian Paden, et. al., 2016*
   This paper surveys a number of popular control techniques for vehicles, including Pure Pursuit.

2. *Implementation of the Pure Pursuit Path Tracking Algorithm, R. Craig Coulter, 1992*
   This is the original Pure Pursuit paper which both describes the algorithm succinctly and its history.

4.2 Alternative Definition of Errors for PID

Try to come up with an alternative definition of error to be used with your PID controller.

1. Justify your definition. Why does it make sense? What pros and cons it may have?
2. Tune the gains for the PID controller on the alternative error you defined.
3. Document the running of your car on a set of reference trajectories.
4. Submit your code and a video of your controller running in sim.

4.3 Avoiding Real Obstacles with MPC

We will be avoiding real world obstacles with the MPC controller that we wrote!

1. Write a subscriber to the laser.
2. Create a cost function that evaluates if a trajectory is within a threshold distance of a hit from the laser scan. If so, set the cost to be very high.
3. Design a custom path to track in real world. Your path would be encoded as `XYHVPath.msgs` and transmitted to the controller. When testing, put a real obstacle (can be anything :) in the path.
4. Document your car tracking these paths in the lab (CSE 022).
5. Submit your code and a video of the car running in lab with the remainder of your submission.

5 Deliverables

After you put the associated work in the directories, `tar` your repository for hand in on canvas.

```
$ tar czf lab2.tar.gz lab2
```

1. `lab2/videos/`: Videos of ...
   - (a) `pid.py` driving the car on each of the 0-3 cases generated by `runner_script.py` in **sim** on `sandbox` map, as `sim_pid.mp4`
   - (b) `mpc.py` driving the car on each of the 0-3 cases generated by `runner_script.py` in **sim** on `sandbox_2_circle` map, as `sim_mpc.mp4`.
   - (c) Any videos of your car running in the **real** world, tracking pre-generated trajectories within the CSE 022 lab space as `real_yourcontroller.mp4` where `yourcontroller` is the controller you chose. Executing paths in the lab space may require some modifications to the `runner_script.py` script.
   - (d) (Extra Credit) `purepursuit.py` driving the car on each of the 0-3 cases generated by `runner_script.py` in **sim** on `sandbox`, as `sim_purepursuit.mp4`
   - (e) (Extra Credit) `pid.py` driving the car with PID controller but using alternative error definition, on each of the geometry cases generated by `runner_script.py` in **sim**, as `sim_pid_alt.mp4`
   - (f) (Extra Credit) `mpc.py` driving the car on your custom path with an obstacle in **real**, as `real_mpc_obstacle.mp4`
2. `lab2/bags/`: Bags:
   - (a) Bags corresponding to each of the runs recorded in the `*.mp4` videos. Name them correspondingly, e.g. `sim_pid.bag, sim_mpc.bag, sim_purepursuit.bag`
   - (b) In folder `lab2/bags/experiments` submit all bag files which correspond to plots you submitted in your writeup. They should correspond to various runs you collected, from initial tests to final experiments. You can name them with any convention you wish, but one suggestion is to have one sub-folder corresponding to each controller, and then to number each bag by trial run.
3. `lab2/src/`: Code:
   - (a) Your `control_node.py`

   (b) Your `pid.py`
   (c) Your `mpc.py`
   (d) Any extra credit source files, such as `purepursuit.py`.

4. `lab2/launch/`: Launch files:
   (a) Your `pid_controller.launch`
   (b) Your `mpc_controller.launch`
   (c) Any extra credit launch files, such as `purepursuit_controller.launch`.

5. `lab2/writeup/writeup.tex`: Your source LaTeX doc containing your project writeup. Put any figure in `lab2/writeup/figs/`.

6. `lab2/writeup/writeup.pdf`: PDF Document documenting your writeup. It should answer the following questions:
   (a) **PID Controller**
      i. Document the tuning process for your controller.
         A. What metrics did you use to evaluate the quality of the controller?
         B. What was its performance profile before tuning?
         C. How much did these metrics improve from tuning it?
         D. What tradeoffs did varying the gains K_p and K_i have on the controllers performance?
         E. What K_p and K_i values did you select for your controller?
         Substantiate your answers with plots and tables where necessary. We encourage you to use and modify `Controller Plotting.ipynb` as needed.
      ii. Compare the two ways of computing $\frac{de(t)}{dt}$. Which do you prefer and why?
      iii. (Extra Credit) Alternative definition of error.
         A. Justify your definition. What can be its pros and cons?
         B. Brief about your gain tuning process.
         C. How does your controller work on a set of different reference trajectories?
         D. How does your controller work compared with the one using Cross Track Error?
   (b) **Model Predictive Control**
      i. **Trajectory generation**
         A. Plot the trajectories generated by your Model Predictive Controller using the kinematic car model. Your result should resemble Figure 3. We provide a helper function `viz_paths_cmap` in `rosviz.py` that you can use.
         B. What are the number of rollouts (K) and number of time steps (T) you selected? Document your process for finding these parameters.
      ii. **Cost function**
         A. What is your cost function?
         B. How did you find your cost function? What worked and what didn't?
         C. Did you need to tune gains for different terms in your cost function? If so, document your process for tuning the gains.
      iii. **(Extra Credit) Avoiding obstacles**
         A. What is your new cost function?
         B. Plot your trajectories colored by cost when the car is near the obstacle, like Figure 3.
   (c) **(Extra Credit)Pure Pursuit Controller**
      i. Document the process of tuning your controller. Consider documenting the performance of your controller in a table accross varying turn radii and speed regimes.
         A. How did varying the lookahead distance affect the robustness of your controller? At which distances did performance suffer most, and in what way?
         B. In the `runner_script` run the `circle` case with various turn radii. How did varying the turn radii of the reference paths affect the robustness of your controller? Did testing on different turns help narrow down which lookahead would be most performant?
         C. In the `runner_script` try different values of the `desired_speed` parameter on each reference trajectory. How did varying the fixed speed affect the robustness of your controller?

D. You may find that the configured parameters of your controller work better in sim than they do in real. Did you have to tune your controller to have different parameters when operating in the real world than in sim? If so, document your process of tuning in real. Feel free to supplement with plots, bag files and videos.

ii. How does pure pursuit compare against PD control? What tradeoffs would you consider when deciding between controllers?

(d) **Overall findings:** Systematically compare each controller and summarize your findings. Think about the trade offs between each controller, such as:

i. In which settings is each controller most advantageous?

ii. Which controller is most robust?

iii. Which controller worked best in high speed regimes?

iv. How did sim-to-real transitions compare between controllers?

6 Demo

On demo day, you will be asked to demonstrate to the TAs:

1. One or more of your controllers on the generated paths from `runner_script.py` in **real**.
2. One or more of your controllers on a previously unknown path in **real**.
3. (Extra Credit) MPC controller avoiding obstacles on a previously unknown path in **real**.

7 Appendix

7.1 Writing a launch file

You are encouraged to search and use online resources to learn to create a ROS launch file. For each of your controller you are encouraged to create a separate launch file. To give you a general idea, e.g. you want to write a launch file `pid_controller.launch` that will 1) launch the PID controller class you implemented and 2) pass the required parameters for PID controller.

To achieve 1), you should get familiar with `main.py` and `control_node.py` and see how they create controllers. For 2), see how Line 73 at `control_node.py` depends on a param `/controller/type`? It is expecting that some launched node called `controller` and has defined a parameter named `type`.

e.g. Your launch file will launch `main.py` from `lab2` package as a ROS node named `controller`. You will also need to pass parameters in your launch file.

You can learn to write launch files from past assignments, e.g. `publisher.launch` from lab0 and `ParticleFilter_Sim.launch` from lab1. Also this website might help you.

7.2 Running the controllers

In RViz, visualize `/controller/path/poses` under PoseArray. This will visualize the reference path.

To run the controllers in the simulator:

- Launch `teleop.launch`
- Launch `lab2/xxx_controller.launch`
- Use `runner_script.py` to test your controller against different paths.

To run the controllers on the real robot:

- Launch `teleop.launch`.
- Launch `lab2/map_server.launch` with `cse022.yaml` as the map file.
- Launch the `xxx_controller.launch` file you wrote. Note that if you have a flag for running in real, you need to turn it on.

- Put the robot in CSE022, localize it by setting its initial pose via Rviz.
- Use `runner_script.py` to test your controller against different paths. You can run it with `cse022 real path` when asked for the plan. We recommend putting the car near the reference path and checking whether it closely follows the reference path.
- Consider tuning `desired_speed` in `runner_script.py` line 87 to a lower value (e.g. 0.5) during testing.

The `xxx_controller.launch` file should do the following:

- Spin off `main.py` and set ROS parameter `/controller/type` with the controller name (PID, MPC, PP). See `lab1/launch/ParticleFilter.launch` for spinning off a node with a python script and setting a ROS parameter.
- Specify `pose_topic` to be the pose topic published by the particle filter, i.e., `pf/viz/inferred_pose`.

7.3 Generating a new path

You can generate new paths making a list of waypoints for the robot to follow. One easy way may be to tune the code we provided in `runner_script.py`, i.e. tuning the parameters of `saw, left_turn, right_turn, circle` or concatenating the generated trajectories to generate various shapes. You can change parameters such as `turn_radius, straight_len` etc.

When using these paths, you should translate the path such that the initial waypoint starts from the map coordinate close to the robot. That is, if the robot is at (x, y, θ) in the map frame and the generated pose starts from $(0, 0, 0)$, you should translate the whole path with something close to (x, y, θ). Likewise, when concatenating two paths, the the second path should be translated to start at the end of the first path.

7.4 Misc

When working with angles, be aware of that $2\pi = 0$.