# Assignment 2: Control

The goal of this assignment is to make you acquainted with fundamental instances of controllers and their trade-offs within the context of *path tracking* with mobile robots. In this assignment you will implement three distinct controllers and analyze them with respect to each other. Each controller will be designed to track a predetermined trajectory, but as you will see 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, pure pursuit, and model predictive control (MPC).
- be able to identify for which scenario each control strategy works best.
- have developed skills to analyze control strategies and iteratively improve them.
- gain an appreciation for the power of cost functions.

*Please reference the "Deliverables" section early on in the assignment, as some parts of this lab will require you to maintain scrupulous notes and data collection practices. A principled approach to observation and analysis of your system will go far in helping you develop robust controllers.*

*There are TWO extra credit questions. Four member teams should attempt at least one of them.*

## 1 Path Tracking

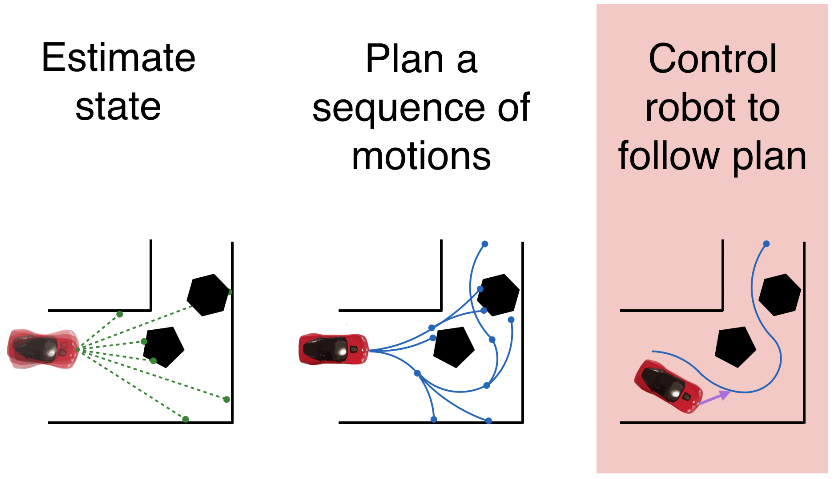

**Figure 1:** Path tracking within the context of mobile robotics.

### 1.1 Overview

Path tracking algorithms have a rich history in mobile robotics. The controllers you will implement have been used in DARPA Grand Challenge and DARPA Urban Challenge among many other contexts. In many land-based navigation scenarios, the robot's navigation task will be divided into two separate tasks: motion planning and control. The purpose of the motion planner is to generate a reference trajectory which the controller can follow. In our configuration, our controller will use the inferred pose generated by the localization module to generate controls which track a trajectory provided by a motion planning module. As such, your implementation will adhere to the diagram in Figure 2.

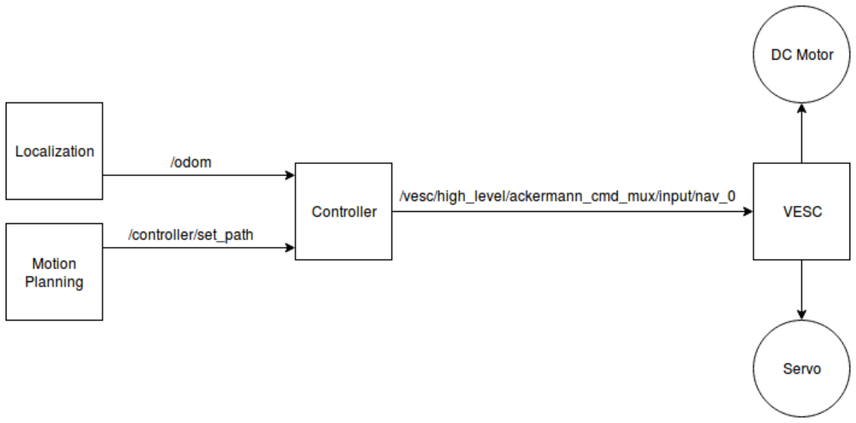

**Figure 2:** System overview of localization, planer, and controller nodes.

Each reference trajectory $\xi$ will be specified as a series of reference waypoints $\xi(\tau) = (x_{ref}, y_{ref}, \theta_{ref}, v_{ref})$ in the vehicle's world frame. Note that this does include reference velocities as well. For our purposes, the reference trajectory will be encoded as `XYHVPath.msg` ROS message.

Familiarity with the following terms will help you with the design and implementation of your controller:

### 1.2 Reference Index ($\tau$)

Every reference trajectory $\xi$ will consist of a series of reference waypoints your controller can track. The "reference index" $\tau$ specifies which reference point is selected from the trajectory, and it is usually a function of the vehicle's pose $p$.

### 1.3 Cross Track Error (CTE, $y_e$)

Each controller will seek to minimize some form of error between the car's pose $p$ and the reference pose $p_{ref}$. One way to compute this error is known as Cross Track Error (CTE). Cross Track Error $y_e$ is the $y$ component of the error vector $e_p$ between the pose of the car $p$ and the reference point $p_{ref} = (x_{ref}, y_{ref})$. In order to compute the CTE, you must rotate the error vector into the frame of the car, using the inverse rotation matrix $R^T$:

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

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

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

## 2 PID Control

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

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

where $K_\mathrm{p}$, $K_\mathrm{i}$, and $K_\mathrm{d}$, are all non-negative coefficients for the proportional, integral, and derivative terms of the error $e(t)$ (in our case this will be CTE). 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 **PD Control**.

Reference the `pid.py` file for getting started.

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 radii 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 $\mathrm{K}_p$ and $K_\mathrm{d}$ values is described in this article.

You may find the included ipython notebook `bags/Controller Plotting.ipynb` useful for generating plots that describe the performance of your controller. Use this as a starting point for analyzing your controller, and document your tuning process. 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.

## 3 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.

We encourage you to reference this paper when implementing the controller.

Reference the `purepursuit.py` file for getting started.

You will notice that pure pursuit relies on fewer parameters than the PD controller for its implementation. 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 radii 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.*

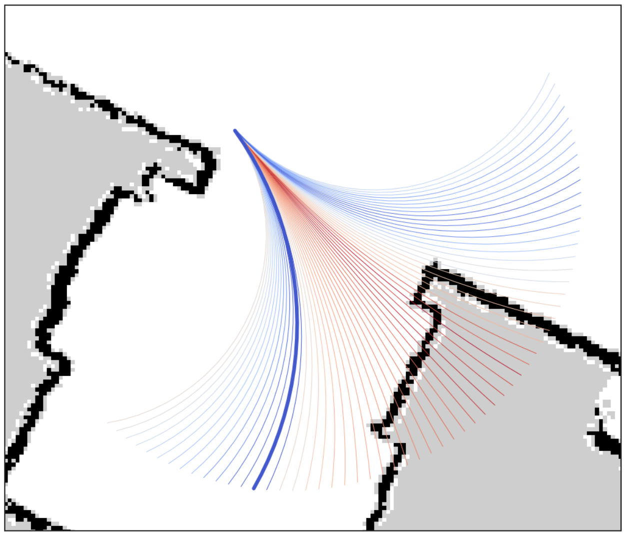

**Figure 3:** MPC rollouts colored by cost function, where red visualizes high cost (from collision) and blue visualizes low cost.

## 4 Trajectory Library (Model Predictive Control)

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

Although the simple control laws above are adequate in many driving conditions, taking additional factors into account (such as the slippage of the vehicle or nearby obstacles) is useful when tracking a path. Such details can be incorporated into a model of the vehicle or the environment, which is then used 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 computing the next action given the most recent state estimate. 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 encoded by the cost function.

Your model predictive controller will be implemented in `mpc.py`. In this file 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.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.

## 5 (Extra Credit) 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 for the robot to track. Put a custom obstacle 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 with the remainder of your submission.

## 6 (Extra Credit) Non-linear Control

As we discussed in class, a powerful and popular way of designing provably stable controllers for non-linear systems is using Lyapunov technique.

1. Write down the dynamics of the system in terms of cross track error and heading error.
2. Write down a Lyapunov function and prove that its a valid Lyapunov function.
3. Derive a control law. Implement this control law. Analyze what the law does in each of the following 4 cases:
   - heading error = 0, small cross track error
   - heading error = 0, large cross track error
   - heading error = $\frac{\pi}{2}$, small cross track error
   - heading error = $\frac{\pi}{2}$, large cross track error
4. Test the controller on the use cases provided for the PID and Pure pursuit controllers.
5. Is this controller always stable? What assumption does it make for the proof to hold? Try to get it to diverge in practice.

For a more sohpisticated Lyapunov controller, refer to: Bank-to-Turn Control for a Small UAV using Backstepping and Parameter Adaptation, Dongwon Jung and Panagiotis Tsiotras, 2008

## 7 Deliverables

Put the associated work in the directories and `tar` your repository.

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

1. `lab2/videos/`: Videos of ...
   (a) `pid.py` driving the car on each of the three cases generated by `runner_script.py` in **sim** as `sim_pid.m4p`
   (b) `purepursuit.py` driving the car on each of the three cases generated by `runner_script.py` in **sim** as `sim_purepursuit.m4p`
   (c) `mpc.py` driving the car on each of the three cases generated by `runner_script.py` in **sim** as `sim_mpc.m4p`
   (d) (Optional) Any videos of your car running in the real world, tracking pre-generated trajectories within the CSE 022 lab space. Executing paths in the lab space may require some modifications to the `runner_script.py` script.
2. `lab2/bags/`: Bags:
   (a) Three bags corresponding to each of the runs recorded in the `sim_*.m4p` videos. Name them `sim_pid.bag, sim_purepursuit.bag, sim_mpc.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 `controlnode.py`
   (b) Your `libcontrol/pid.py`
   (c) Your `libcontrol/purepursuit.py`
   (d) Your `libcontrol/mpc.py`
   (e) Any extra credit source files, such as `libcontrol/nonlinear.py`.
4. `lab2/launch/`: Launch files:
   (a) Your `pid_controller.launch`
   (b) Your `pp_controller.launch`
   (c) Your `mpc_controller.launch`
   (d) Any extra credit launch files, such as `nl_controller.launch`.
5. `lab2/writeup.tex`: Your source LaTeX doc containing your project writeup.
6. `lab2/writeup.pdf`: PDF Document documenting your writeup. It should answer the following questions:
   (a) **PID Controller**
      i. Document the process of tuning your controller.
         A. What was its performance profile before tuning?
         B. What metrics did you use to evaluate the quality of the controller?
         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?

         Substantiate answers to each of the previous questions with plots and tables where necessary. We encourage you to use and modify `Controller Plotting.ipynb` as is useful.
      ii. What $K_p$ and $K_i$ values did you select for your controller?
      iii. One way of computing $\frac{de(t)}{dt}$ is with $\sin(\psi_e)$ where $\psi_e$ is the heading angle of the car with respect to the centerline. How does this implementation affect the controller's performance as compared to numerical differentiation? Which did you select for the final implementation of the controller? Why is your choice more performant?
   (b) **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. Updated 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. Updated 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?
   (c) **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.
         B. What is the number of rollouts ($K$) and number of time steps ($T$) which you selected? Document your process for finding these parameters.

    ii. **Cost function**
        A. What is the cost function your team selected?
        B. What was your process for finding the 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.
        D. (Extra Credit:) Create a heatmap of your cost function for a selected goal point. Try plotting the heatmap for the Allen Center basement and also for CSE 022.
    iii. Did your team find it useful to use a similar lookahead principle as in the Pure Pursuit controller?
(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?

## 8 Demo (Updated)

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. Your MPC controller avoiding obstacles on a previously unknown path in **real**.

## 9 Appendix (Updated)

### 9.1 Running the controllers on the real robot

To run the controllers on the real robot, you need to follow steps you've taken in the first two assignments to load the particle filter:

- Launch `teleop.launch`.
- Launch `lab2/map_server.launch` with `cse022.yaml` as the map file.

Then, to start the controller,

- Launch the `xxx_controller.launch` file you wrote.

The 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`.

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

Finally, put the robot in CSE022, localize it by setting its initial pose via rviz, and run `scripts/runner_script.py` to start generating a path. You can run it with `cse022 real path` when asked for the plan. We recommend setting it 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.

### 9.2 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 use 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 $(x, y, \theta)$ in the map frame and the generated pose starts from $(0, 0, 0)$, you should translate the whole path with something close to $(x, y, \theta)$. Likewise, when concatenating two paths, the the second path should be translated to start at the end of the first path.