Instruction for Assignment 1 for Term Project

PID Algorithm

Introduction

In this assignment, a race car is required to follow or track a given path. So you have to implement a controller which can perform tracking a given path well. To do this, you will use "PID controller", which is widely used control feedback mechanism. The "PID controller" finds a control which makes a robot move toward goal smoothly. We will provide skeleton code to you and you have to fill in the TODO part.

PID

Main Theory [1]

A PID controller (proportional–integral–derivative controller) is a control loop feedback mechanism widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an error value e(t) as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively) which give their name to the controller.

The PID control scheme is named after its three correcting terms, whose sum constitutes the manipulated variable (MV). The proportional, integral, and derivative terms are summed to calculate the output of the PID controller. Defining u(t) as the controller output, the final form of the PID algorithm is

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,d\tau + K_\mathrm{d} \frac{de(t)}{dt}$$

where

Kp : the proportional gain

Ki : the integral gain

Kd : the derivative gain

e(t) : the error between the setpoint and the current process variable

The proportional term produces an output value that is proportional to the current error value. The proportional term mainly tries to adjust the error term e(t) to zero. The proportional response

can be adjusted by multiplying the error by a constant **Kp**, called the proportional gain constant. A high proportional gain results in a large change in the output for a given change in the error, but the system can become unstable. In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller.

The contribution from the integral term is proportional to both the magnitude of the error and the duration of the error. The integral in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. The accumulated error is then multiplied by the integral gain **Ki** and added to the controller output. The integral term accelerates the movement of the process towards setpoint and eliminates the residual steady-state error that occurs with a pure proportional controller. However, since the integral term responds to accumulated errors from the past, it can cause the present value to overshoot the setpoint value.

The derivative of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain **Kd**. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gain, **Kd**. Derivative action predicts system behavior and thus improves settling time and stability of the system.

Discretization

In the PID controller, the integral term can be approximately discretized, with a sampling period Δt, as

$$\int_0^{t_k} e(\tau)\,d\tau = \sum_{i=1}^{k} e(t_i)\Delta t$$

and the derivative term can be discretized as

$$\frac{de(t_k)}{dt} = \frac{e(t_k) - e(t_{k-1})}{\Delta t}$$

Therefore, the control value u(t) can be approximated as

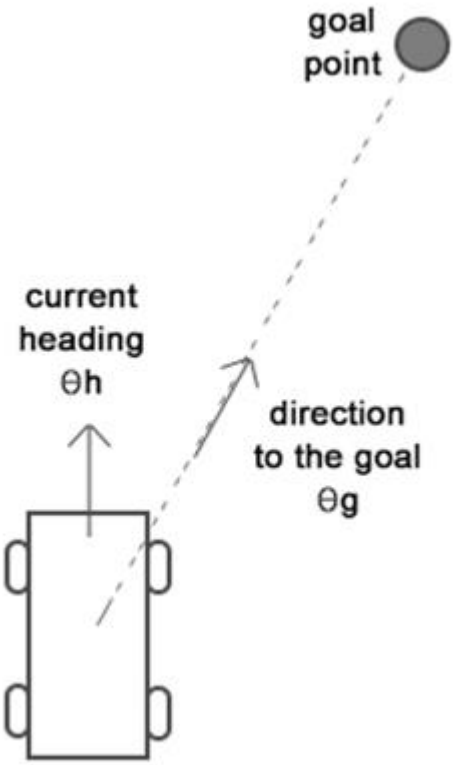
$$\mathbf{u[t]} = \boldsymbol{K_p e[t]} + \boldsymbol{K_i \Delta t} \sum_{\boldsymbol{k=1}}^{\boldsymbol{t-1}} \boldsymbol{e[k]} + \frac{\boldsymbol{K_d}}{\boldsymbol{\Delta t}} (\mathbf{e[t] - e[t-1]})$$

or, in recursion relation form,

$$\mathbf{u[t]} = \mathbf{u[t-1]} + (\boldsymbol{K_p} + \frac{\boldsymbol{K_d}}{\boldsymbol{\Delta t}})\boldsymbol{e[t]} + \left(-\boldsymbol{K_p} + \boldsymbol{K_i \Delta t} - 2\frac{\boldsymbol{K_d}}{\boldsymbol{\Delta t}}\right)\boldsymbol{e[t-1]} + \frac{\boldsymbol{K_d}}{\boldsymbol{\Delta t}}\mathbf{e[t-2]}$$

Application

You should control the heading direction of the car to make it follow given path. So, in this case, the variable to process is the heading direction of the car. To make the car reach the goal point, the car heading should face the goal point. You can calculate the direction to the goal point for your car in polar coordinate. This value becomes the setpoint which the controller should track.



To summarize, the process value which to control is θ_h, and the set point can be θ_g, then $e(t) = \theta_g(t) - \theta_h(t)$.

Skeleton Code

- Point.h
- pid.h & pid.cpp
- pidmain.cpp

You can download skeleton code from the course webpage. And extract it into "catkin_ws/src/" as previous assignment. This project files are composed of three header files and two cpp files. Two header files just indicate data type. One cpp file is for pid class. And main cpp is controller for race car.

Point.h

Point.h contains structure "point" or "point" data type. It contains three variables x, y and th. These variables represent car's position and heading respectively. (x, y) is the position from the origin according to global frame. Variable th is angle of heading from x axis in radian.

```
struct point{
        double x;
        double y;
        double th;
};
```

Pid.h & Pid.cpp

PID class has private member variables for the coefficients of the PID controller, and public member function "get_control(point car_pose, point goal_pose)". The function "get_control(point car_pose, point goal_pose)" returns control "ctrl" as a result of PID controller. You should calculate θ_g from the two arguments, "car_pose" and "goal_pose". Also, you can get θ_h and θ_d from "car_pose.th" and "goal_pose.th" respectively.

You must fill in the TODO part in the pid.cpp file. When you are to use this library in the main loop (pidmain.cpp), use a pid class instance and call get_control member function to get control. You can add your own private member variables to the PID class if you need them. The control value will be used to fix the steering angle of the car. Finally, publish them to GAZEBO simulator. Usage of this class is explained in next section.

Pidmain.cpp

It contains publishers visualizing path and get the racecar position from GAZEBO topics. So you just need to implement publisher of racecar's speed and steering angle. This publisher is exactly the same as in the previous assignment. But this time, you need to add ROS spin function and sleep process for some duration. Whole structure you have to implement in TODO part in pidmain.cpp is as follows.

- Make control input using pid class. Use predefined pid class instance "pid_ctrl"
- Publish control to racecar. Use predefined publisher, "car_ctrl_pub" and use predefined variable, "drive_msg_stamped".
- Check whether the car reach a currently following way point or not. Calculate distance between current position of racecar and currently following way point (look-ahead point). If

the distance is less than 0.2m (certain threshold, you can change this value), pursue next way point (look-ahead point).

- Check whether the car reach final way point (end of the path). If so, terminate controller.

All instances for publishing control to pioneer in GAZEBO is already defined and you can use it by calling it to control the racecar. "car_ctrl_pub" is the publisher to publish velocity to GAZEBO. "drive_msg_stamped" is a variable which contains control value for the speed and the steering angle of the car.

You can get the current position of the car from predefined variable "car_pose". "car_pose.x" and "car_pose.x" represent the position of the car in Cartesian coordinate, while "car_pose.th" represents the direction of the car heading in polar coordinate. The variable "car_pose" is updated automatically in other parts of the Pidmain.cpp code, so you don't need to get the position value of the car directly from ROS system.

Path will be given as vector type variable "path". All the way points are stored in this variable "path". Also, pre-defined directions θ_d are stored in the "path[i].th". Template type of the vector is "point" (std::vector<point>). The racecar has to sequentially track the given path and the controller should be terminated when the car arrives the final point of the path.

Assignment

You have to implement a code that a racecar smoothly tracks a given path. You can check your code by simulating the controller using following commands.

roslaunch project1 project1.launch

rosrun project1 project1

If you type "roslaunch project1 project1.launch", you will see GAZEBO simulator with a car. And if you run your rosnode, the blue balls appear in the air above each way point. Your racecar will follow the blue balls.

The pid controller will be used in the next assignment. Therefore, it will be helpful to design a good and neat pid controller.

Submission Format

Compress your project folder which includes all your project files and upload it on eTL. The name of compressed file should be "**IS_Project_01_[TeamName].tar.gz**".

Reference

[1] https://en.wikipedia.org/wiki/PID_controller#Alternative_nomenclature_and_PID_forms