COMP 417 – Assignment 3

PID and LQR Control

Out: Nov 21st, 2019
Due: Dec 3rd, 2019 @ 6pm

Getting Started:

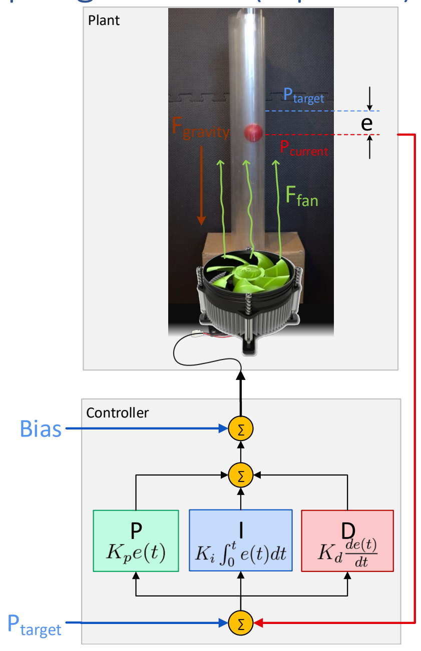
Obtain the code by doing **$ git pull** in the COMP417_Fall2019 repo (https://github.com/dmeger/COMP417_Fall2019.git). Or, if you have any disk space issues such as at Trottier, you can get only the two needed folders:

```
$ svn export https://github.com/dmeger/COMP417_Fall2019/trunk/pid_question
```

and

```
$ svn export https://github.com/dmeger/COMP417_Fall2019/trunk/lqr_question
```

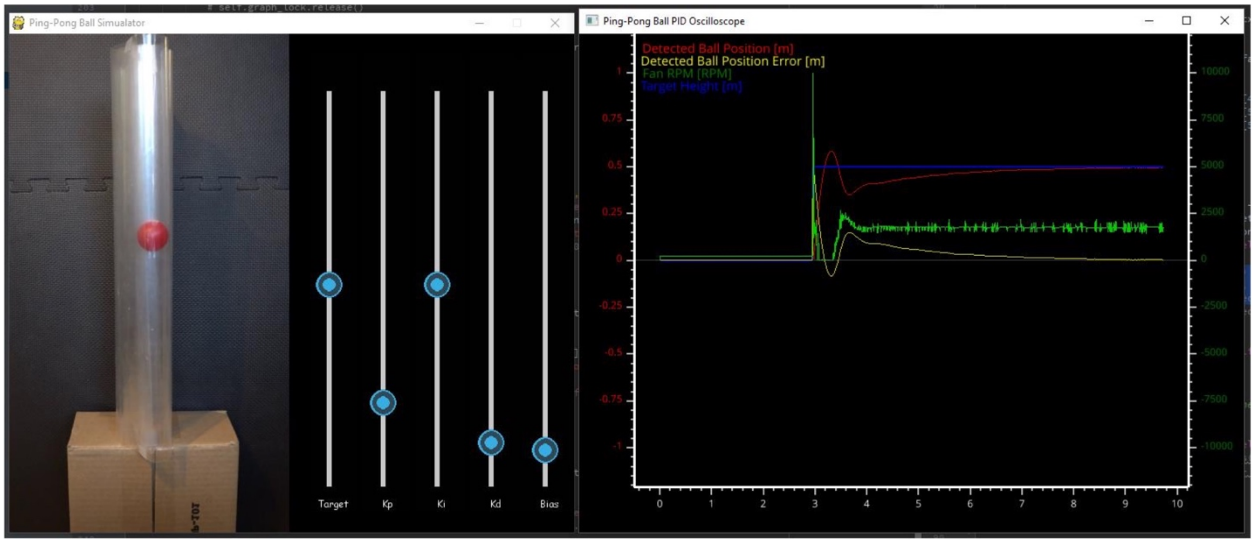
Question 1: PID Control of a Ball Controlled by Spring and Fan (5 points)



The goal of this question is to understand the PID controller in the context of balancing a ping-pong ball using a fan as shown to the right. The main tasks will be to code the PID control logic, and then perform tuning of your controller for several cases. Inspect the Python code in the **pid_question** folder. This code has few requirements and can be run on your own system if you desire. The main dependencies are pygame and vispy, both of which are (hopefully) easy to pip install. As we publish this assignment, the code needs one more package to be installed to run at Trottier, and that should be done by the end of the day on Nov 21.

The simulator is run using the command: **$ python main.py**. Afterwards, you can type 'e' to run the simulator in experimental mode (normal mode of operation), or 'v' [target height] (ie 'v 0.5') for evaluation mode where the program will run a step response for several seconds and then show a plot with the results. The validation mode will be used to mark your assignment.

Two windows should appear as shown below (only the left window appears on Mac). The left window is the simulator. The four sliders adjust the target height, Kp ,Ki, Kd and bias parameters of the pid controller. You can manually control the output of the fan by pressing the left mouse button inside the image area and slide the cursor up and down to adjust the fan rpm.


Ping-Pong Ball Simulator
Target
Kp
Ki
Kd
Bias
Ping-Pong Ball PID Oscilloscope
Detected Ball Position [m]
Detected Ball Position Error [m]
Fan RPM [RPM]
Target Height [m]

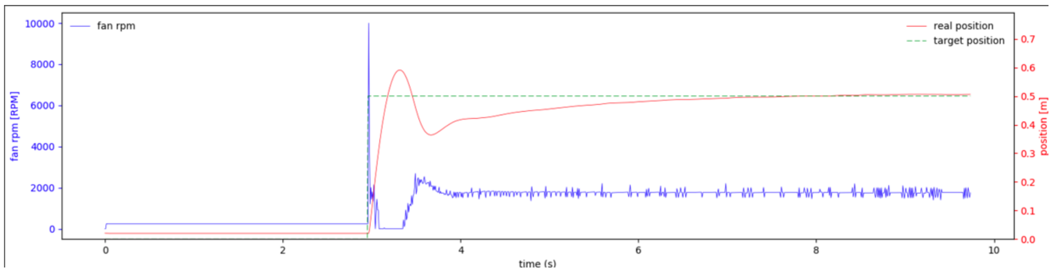
Some additional commands are:
•’r’ - reset the simulation, the target height is set to zero and the ball is reinitialized to the starting position.
•’g’ - show a plot of the target height, fan output, detected ball position, and real output.
•numbers ’0’ through ’9’ - sets the target height of the ball. This is useful to simulate the step response of the PID controller.

The right window is a real-time graph of the state of the system (Note that currently, this live graph does not appear on macOS due to a threading limitation by the OS, however it is not necessary for the assignment. The ‘g’ function on the simulator still works to pull up a snapshot graph for marking, even on Mac). You can manipulate the graph using the mouse buttons and scroll wheel and the keyboard commands are:
•’r’ - reset the graph to show whole plot.
•’s’ - auto-scroll

Step 1) Code the PID controller

In the file **pid_question/pid.py** you will find the implementation of a PID control class. Note that the “self.” variables in that class for the bias and PID gains are set by our code automatically as you interact with the sliders on the GUI. The function **get_fan_rpm** is supposed to implement the controller, but in the starter code, it only outputs the setting of the “bias” slider – as in, it doesn’t really do any control. You must fill in the right logic to get the ball moving automatically to the target. Implement the proportional, integral and derivative contributions as discussed in class. You can add new global or member variables or change that file in any way that you like, except do not change the existing member variables of the class, as that would break the GUI interface. When you’re finished, run the code and move the sliders to see that things work as you expect. You will hand in your **pid.py** file.

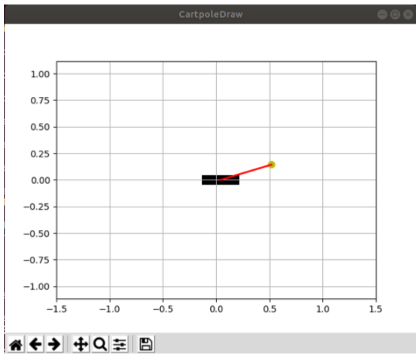
Step 2) Tune the PID gains and report performance. It is possible to find PID gains that make the ball respond quickly, but without overshoot or oscillation to target changes such as 0->0.5. Using methods in class, or those you discover yourself, experiment with gains until you have a responsive and smooth control performance without excessive oscillation or overshoot. Can you get a better control curve than this? (remember to see this type of graph by pressing 'g' on the simulator)



Save a screenshot of your graphs resulting from the best gains you can find and save as "**PID_response_Q1.png**" as well as a movie of the ball's motion as "**PID_response_Q1.<ext>**" where <ext> is ogg, mp4, avi or some other common movie format. Hand in both files.

2 LQR Control of a Cart Pole Robot (5 marks)

For this question, you will derive the matrices necessary to call an LQR library and derive the optimal controller that can balance a cart-pole robot at the goal, using the simulator provided in the **lqr_question** folder. Running **$ python cartpole_learn.py** will give the visualization below. Note this can also be run on your own computer or at Trottier. The dependencies are numpy, scipy and matplotlib in this case.



We have not been able to make it run directly on macOS due to a similar GUI problem to Q1, but this window is very lightweight, so it seems to work well to ssh to Trottier and forward the graphical output (such as installing XQuartz and then typing **"$ xhost +**" and then "**ssh -XC user@mimi.cs.mcgill.ca**").

The state vector of this cart pole is:

$$x = [x \quad \dot{x} \quad \dot{\theta} \quad \theta]^T$$

Where x is the position of the cart along the x axis, in metres, and theta is the angle of the pole (from the zero point: "downwards") in radians. Both velocity components (with a dot above) are in units per second.

The control for this system is simply a force applied to the cart (black rectangle), and otherwise the robot's dynamics are determined by a gravity force that pulls the pole downwards, a drag that opposes cart velocity and the mechanical coupling at the single joint.

We can write these dynamics equations on two lines after integrating the dynamics Lagrangian. You don't need to understand this derivation (we haven't taught you in 417), but feel free to investigate it if interested. For the assignment, you can just trust us that these 2 equations capture the dynamics that the simulator is using to model the cart pole robot:

$$\ddot{x} = \frac{2ml\dot{\theta}^2 \sin\theta + 3mg\sin\theta\cos\theta + 4(u - b\dot{x})}{4(M+m) - 3m\cos^2\theta}$$

$$\ddot{\theta} = \frac{-3(ml\dot{\theta}^2 \sin\theta\cos\theta + 2((M+m)g\sin\theta + (u - b\dot{x})\cos\theta))}{l(4(M+m) - 3m\cos^2\theta)}$$

The relevant constants for our simulation are:

- Pole length, l=0.5
- Pole mass, m=0.5
- Cart mass, M=0.5
- Friction constant, b=0.1
- Force due to gravity, g=9.82

Step 0) Run the code with **$ python cartpole_learn.py**. At first, you should see the pendulum fall and swing back and forth in the GUI window, because no control is implemented there yet. Add some code to the "**policyfn**(..)" that returns u's to explore their effect. For example, you can try applying a constant control or PID. This is just for the purpose of you getting familiar with the system and is not for marks.

Step 1) Using pen and paper, attempt to express this system as a linear approximation, $\dot{x} = f(x,u) = Ax + Bu$ around the goal (the upwards balanced point). In the robot's state space, this goal is $x = [0 \quad 0 \quad 0 \quad \pi]^T$. Intuitively, this describes that the pole is upright ($\theta = \pi$) and the cart is at the centre, x=0, with zero for both velocities.

The dynamics equations above can be tricky to differentiate entirely, so we can first use small angle approximations to simplify. Think about how to replace the $\sin\theta$ and $\cos\theta$ terms with simple functions of θ (or constants) near the goal. You should arrive at something trivial to differentiate with respect to each state variable, (no more than linear or quadratic terms remaining in each state variable). Remember A and B are both Jacobians, computed like this:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Also, write down Q and R matrices for the quadratic cost function that will encourage the system to stay balanced (that is, the minimum of the cost should be at the balanced state).

Step 2) Code your LQR controller by replacing the place-holder code in **cartpole_learn.py** with the correct solution. Using the A, B, Q, and R matrices you solved for, call a math library that will solve for the control gains K to apply to the system. Use **scipy.linalg.solve_continuous_are(),** who's documentation is here:
https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.linalg.solve_continuous_are.html
Apply the K that you receive from the LQR solver to the state of the robot within **policyfn**, to compute the control as: $u = K(x - g)$. Run the code to see the cartpole balancing. Record a video and name it "**LQR_reponse_Q2.<ext>**" where <ext> can be ogg, mp4, avi or any other standard movie format. You will hand in your cartpole_learn.py as well as this video result.

3 How to submit

Submit all your work in a file called A3.zip. It should contain 2 Python files, 1 per question, and, as well as 1 image and 2 videos. Submissions will be done on MyCourses.