Lab 4: Self-Balancing Robot

Lab TAs: Zachary Dawson, Sarah Tan, Andrew Schroeder

Learning Objectives

- Implement an example of the inverted pendulum problem.
- Apply ideas of control theory and PID control.

Part 1: Control Theory

The Inverted Pendulum Problem

Balancing robots represent the classic inverted pendulum problem, in which a large mass is placed at the end of a pole. The pole is free to rotate around the base, and the base is free to move in the plane perpendicular to the vertical. The goal of the problem is to keep the pole vertical by moving the base in response to changes in the angle.

PID Control

PID control, which stands for Proportional-Integral-Derivative control, is an incredibly powerful an ubiquitous control algorithm. An output signal u can be generated by summing the three components:

$$u(t) = \underbrace{K_p e(t)}_{\text{Proportional}} + \underbrace{K_i \int_0^t e(\tau)d\tau}_{\text{Integral}} + \underbrace{K_d \frac{d}{dt} e(t)}_{\text{Derivative}}$$

The constants K_p, K_i, and K_d are referred to as the proportional, integral, and derivative constants, or sometimes the PID tuning constants. $e(t)$ is the error from the desired output at time t. While these exist in continuous space, more practical implementations can be considered below:

The error function e can be implemented as an array of errors at time `t`.

```
e[t] = curr_postition - target_position;
t = t + 1 % ERR_ARRAY_SIZE;
```

The derivative component can be implemented as the difference between the current and the previous errors, divided by the time between the two. However, in most practical applications the time sampling rate is constant, so we do not divide by a constant and instead modulate K_d.

```
de_dt = e[t] - e[t-1];
```

The integral component is somewhat straightforward, as we can simply sum all the elements in our array (this works especially well if uninitialized values are set to 0). However, for various reasons, we may not actually want an integral term in our system. Depending on the implementation of the integral error, it could dominate the controller over time, or add undesired effects near the beginning of the curve (before the error has had sufficient time to settle).

```
sum_errs = 0;
for (int j = 0; j < ERR_ARRAY_SIZE; ++j) { sum_errs += e[j]; }
```

PID Constants

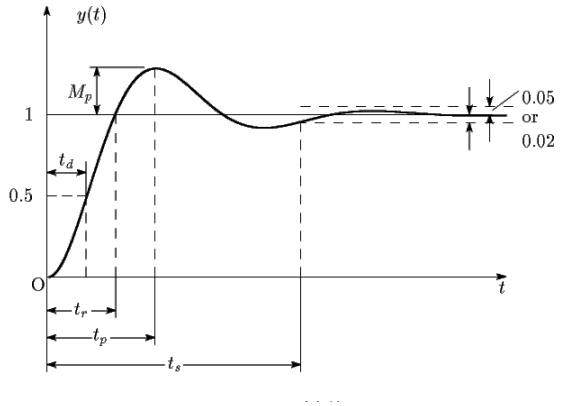

Figure 1: A Parametrized PID Response Curve

By manipulating the PID constants, we can tweak the behaviour of our system to perform as we want. First, let us define some terms that we might be interesting in controlling:

- Rise time: The amount of time it takes the system to go from 10% to 90% of the target value (t_r in Figure 1).
- Settling time: The amount of time it takes the system to settle within a certain percentage of the target value (t_s in Figure 1).
- Overshoot: The maximum value the system takes above the target value (M_p in Figure 1).
- Steady-state error: Error that does not go away over time, i.e. the controller consistently sits at 5% above or below the target value.

Each constant affects these values in different ways (note, these are general rules of thumb, as when the constants are too large they can affect the controller erratically).

- Increasing K_p improves rise time, while worsening settling time.
- Increasing K_d improves rise time, overshoot, and settling time (to a point).
- Increasing K_i improves steady-state error, but can have undesirable effects on your controller.

Tuning a PID Controller

There are many algorithms for tuning PID controllers, but we present a popular and reliable method for your consideration:

- Set $K_p = K_i = K_d = 0$.
- Adjust K_p until the system remains in balance, but rapidly oscillates around equillibrium.
- Adjust K_d until the sytem reaches steady state.
- If there is a steady state error, tune K_i. However, be warned that if the system is not actually reaching equillibrium, the integral term can dominate the controller and otherwise have negative effects on your controller.

Part 2: Building the Robot *(75 points)*

In this lab, you'll have to design, create, and program a balancing robot. This lab is an exercise in allowing you to understand the fundamentals of PID control and control theory.

Robot Constraints

- All weight of the robot should lie over a single axis (more formally, the support polygon must be a straight line).
- "Falling over" is defined as any point at which the support polygon is no longer a straight line (that is, something on the robot touches the ground that does not lie along the single axis of support).
- The robot should fall over if powered off (it cannot be naturally balanced). This must happen within five seconds of being set down and released.
- The robot should only use the lego parts provided in the kits to perform its task. This includes an additional light sensor, which will be provided to you for this lab.
- The robot may have at most 3 motors and 4 sensors.
- In order to gain any point for travelling, the robot must remain standing for at least 5 seconds.

Demo Logistics

- You will be provided with a flat, vertical board as a reference point if you so choose. Note, you should be able to determine the robot's angle with just the two light sensors pointed at the floor.
- The robots will be run on whiteboards scattered throughout the room.
- Part of your demo deliverable is a short code inspection, where the TAs will ensure you have correctly implemented an appropriate PID controller (or some variation of P, I, and D in a control loop).
- You have 3 tries to demo your robot. We will take the highest score achieved.
- The demo is during the lab time on Tuesday, February 9th.
- You must also submit your source code by midnight, Tuesday, February 9th. (Submission instructions will be posted on Piazza).
- Don't forget your grading sheet! It's on the website.

Hints

- Ensure that any light sensors you use are an equal distance from the axis of rotation (the axle).
- Make sure you calibrate your light sensors, because different sensors can give different readings under the same conditions.
- Build your robot tall above the ground (a 'taller' robot will have a larger moment of inertia and will be less susceptible to small fluctuations).
- Build a rigid robot (mechanical flex causes unpredictable oscillations).
- Make sure you test on the whiteboard. The robot will act differently on different surfaces.
- Charge the robot frequently, but don't tune your parameters at the highest power. Power in NXTs drop significantly from fully charged, but they achieve a steady state of power at about 7.4-7.5V and remain that way for the longest period of time. You will not be given time to significantly recalibrate your robot on demo day.

- Watch for the battery power of your robot. It will have a significant impact on your performance.
- You do not have to implement a full PID controller. You can implement any subset thereof (P, PI, PD) that performs as required.
- Tune your constant one at a time. This is critical to being able to lock down the correct constants.
- Don't thrash your motors! This may cause them to lock up at a critical moment.
- Use the sensors in raw mode (accessed by `SensorRaw[<sensor_name>]`) for a higher resolution (but potentially noisier) stream of data.
- The command `nMotorPIDSpeedCtrl[<motor_name>] = mtrSpeedReg` can help motors respond more smoothly.
- Use floating point numbers, then convert them to integers right before assigning them to your motor powers.
- If you're having trouble maintaining position:
  - Use dead reckoning to determine how far from the start point you are.
  - Use an ultrasonic sensor to determine how far you are from the wall.
  - Be sure to transform inputs in terms of angle error (i.e. change your target angle to compensate for moving away from a direction instead of throwing a strange term into your controller).
- Your robot will never get 0 error. Don't frustrate yourself trying to make that happen! (If you do, let us know. We will show you where to collect your PhD.)