Lab 3: Feedback Control and PID

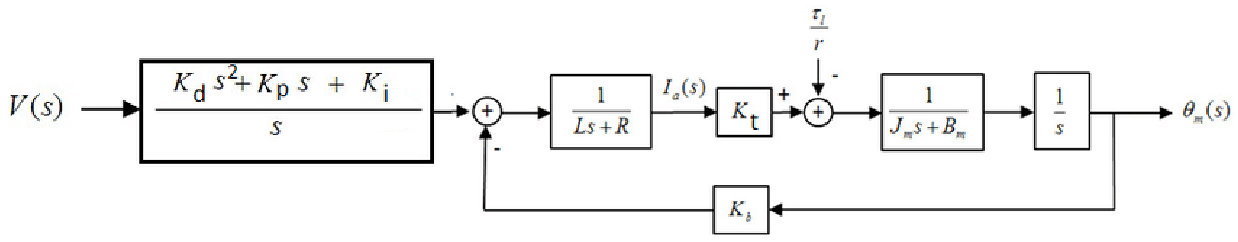
Learning Objectives:
This lab is intended to acquaint you with closed-loop feedback control using microcontrollers and single-board computers. You will use a sensor to provide feedback data from a physical system, and an actuator to change the state of the system and a controller to match the output to your desired state. The controller will measure time increments and calculate the error and the integral and derivative of the error function.

Minimum Parts Required:
A Raspberry Pi and/or an Arduino, an actuator such as a servo or DC motor, and a sensor such as an encoder, rotary potentiometer or ultrasonic or infrared range-finder.

Lab Overview

The goal of this lab is to create a closed-loop feedback control system for a physical system. Examples could include (but are not limited to):

- Hovering ping-pong ball. Use a fan and a column of air to float a ball at a designated, and variable height.
- Self-balancing ball track – using a range-finder to determine the position of a ball on a track and a servo to adjust the angle of the track.
- Inverted pendulum. This is the classic PID demonstration. If you really want to explore the tuning of a PID system, this is your platform.



Other similar systems are possible. Please discuss with Prof. Reamon. Make the physical system as cheap and easy as possible. For instance, cardboard, foamcore, PVC, Legos would all be appropriate.

PID libraries and sample code are available and referenced below, but they may require a lot of adjustment to work with your sensor and actuator. Writing your own code for a PID system is fairly simple in practice, and the lab parts below will lead you through the process. If you find code that works for you, however, you are free to use it.

If you are using an encoder, you may want to start with some code or a code library. You may also need to use interrupt pins. Here are some examples and possible sources:

PID Control with Python/ RPi:
https://projects.raspberrypi.org/en/projects/robotPID/0
https://onion.io/2bt-pid-control-python/
https://github.com/m-lundberg/simple-pid

PID control with Arduino:
https://www.teachmemicro.com/arduino-pid-control-tutorial/
https://microcontrollerslab.com/pid-controller-implementation-using-arduino/

Encoders with Python/ RPi:
https://pypi.org/project/Encoder/
https://github.com/nstansby/rpi-rotary-encoder-python
https://softsolder.com/2020/10/14/raspberry-pi-interrupts-vs-rotary-encoder/
http://abyz.me.uk/rpi/pigpio/examples.html#Python_rotary_encoder_py

Encoders with Arduino:
https://create.arduino.cc/projecthub/curiores/how-to-control-a-dc-motor-with-an-encoder-d1734c
https://www.electroniclinic.com/arduino-dc-motor-speed-control-with-encoder-arduino-dc-motor-encoder/

Interrupts with Python/ RPi:
https://roboticsbackend.com/raspberry-pi-gpio-interrupts-tutorial/
https://www.abelectronics.co.uk/kb/article/1088/io-pi-tutorial-4---more-interrupts

Interrupts with Arduino:
https://learn.sparkfun.com/tutorials/processor-interrupts-with-arduino/all
https://www.allaboutcircuits.com/technical-articles/using-interrupts-on-arduino/
https://www.tutorialspoint.com/arduino/arduino_interrupts.htm

Timing with Python/ RPi:
http://abyz.me.uk/rpi/pigpio/index.html
https://www.geeksforgeeks.org/python-time-clock_gettime-method/

Timing with Arduino:
https://www.instructables.com/Arduino-Timing-Methods-With-Millis/
https://www.best-microcontroller-projects.com/arduino-millis.html
https://www.robotshop.com/community/forum/t/arduino-101-timers-and-interrupts/13072

Part 1 – Sensor Data

Start off by getting sensor data into the computing device of your choice. Once you can print nearly real-time data to the serial monitor, you can move on. Whatever your sensor is measuring is your ***control variable***.

Part 2 – Feedback System

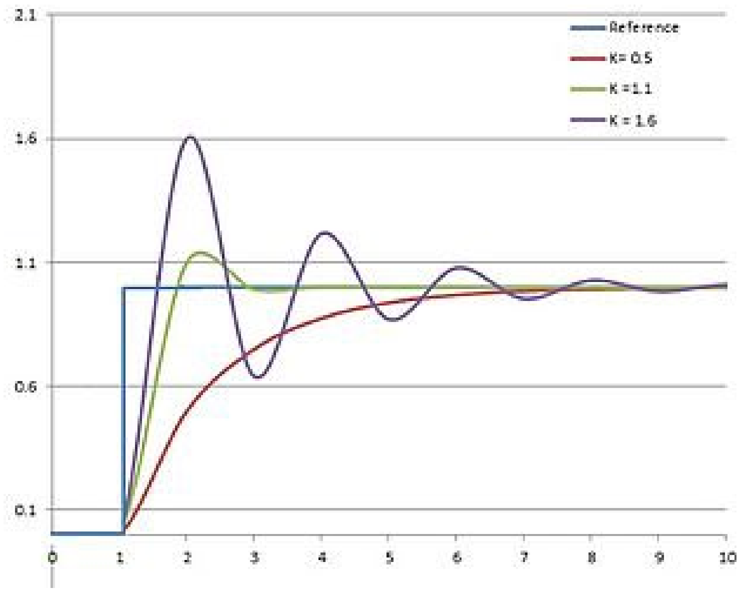
The most important term to calculate is the ***error***. The error provides the feedback in the control loop, and is the difference from the control variable target and the current sensor reading of that variable’s value. A new error term should be calculated each time through your code loop. You will also want to calculate the error sum, so add the newest value to a running total of the error. The error can generally be positive or negative, indicating which side of the target value your system is currently operating on. When the error becomes zero, you have reached the target value.

Part 3 – Time Increments

Next, get your computing device to measure the time increment between sensor measurements. In most cases this will entail saving the current time when the sensor is read, relative to a system clock. During the next loop iteration, you can save a new time, allowing you calculate the time difference (delta-t) between readings. If you find that your time increment is very consistent, you may not use the actual delta-t value, but just adjust your K values accordingly.

Part 4 – PID control and tuning

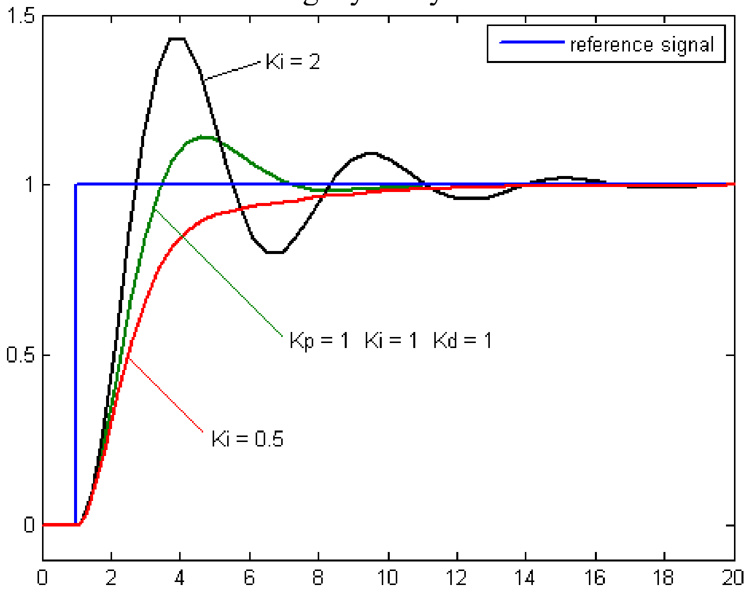
Once you have your error, error sum and delta-t, you are ready to create your PID system, but let’s start with just a proportional (P) control system. For this, take the error value, multiply it by a K_P constant and feed that into your actuator. The K_P value will need to take into account what your actuator is. If you are sending a PWM value to a motor driver, for example, (on Arduino) the maximum value is 255. If your sensor is using an analog read, the maximum value is 1024, but you will want to move your device around to see what its full range is. Assume the sensor can vary from 100 to 900 in operation, and that the target is 500. So the maximum error would be +/- 400. So we would want our error * K_P = 255. So K_P = 255/400 to start (assuming a direction pin is controlling the motor direction based on the sign of the error). At the end of our loop we should send the new pwm value (actuator command) to the motor as K_P * error. Now we have proportional feedback control, which typically works, but includes an inherent steady-state error. Adding the integral term will eliminate the steady-state error.



Response of control variable to step change of setpoint vs time, for three values of K_p (K_i and K_d held constant)

https://en.wikipedia.org/wiki/PID_controller

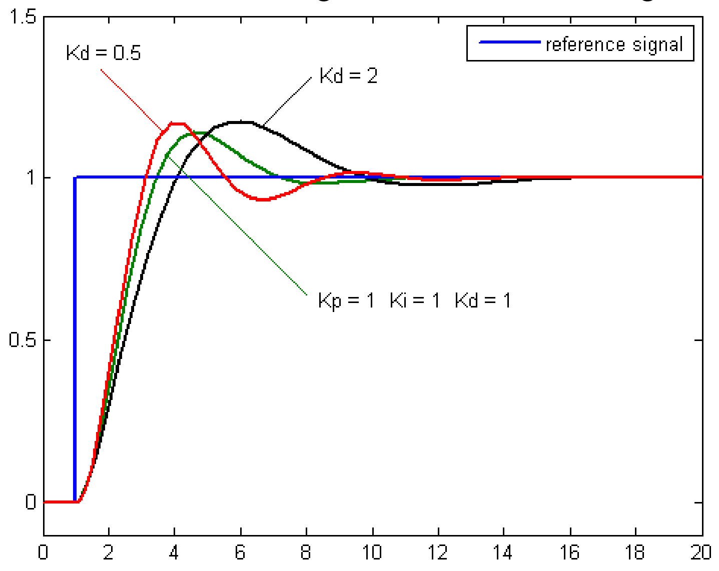
Your error sum term is your integral. To be mathematically correct, you can multiply your error by the delta-t when saving the sum, but this is usually not necessary if your loop time is consistent. Either way, you multiply your error sum by K_I to create the integral term, and add it to your K_P * error term to feed into the actuator command. Since the error term can be positive or negative, the error sum can as well, but we may need to put bounds on it, so that it does not become too large and dominate the P term. You will usually want the K_I * sum term to be smaller than the K_P * error term, so you can scale the K_I value accordingly, but it will definitely require some iteration to get K_I large enough to eliminate steady-state error, but still allow the system to come to equilibrium. You should be able to get your system to overshoot and oscillate with a higher K_I.



Response of control variable to step change of setpoint vs time, for three values of K_i (K_p and K_d held constant)

https://en.wikipedia.org/wiki/PID_controller

Finally, you can bring in the derivative term. You can find the derivative by subtracting the previous error term from the current error term, and dividing the difference by delta-t. Again, if your loop time is consistent, you can leave out the delta-t and adjust K_D accordingly. If your sensor is noisy, you will want your K_D value to be very small, and may not even be able to use the derivative term in practice. If your sensor data is fairly clean, the derivative term will help anticipate large changes and damp the response. The derivative term will generally flatten the response curve, slowing the time to target, but can be critical in reducing overshoot from the integral term.



Response of control variable to step change of setpoint vs time, for three values of K_d (K_p and K_i held constant)

https://en.wikipedia.org/wiki/PID_controller

Once you get everything working (or at least PI control) ***tune it for two responses***. One should be a response with a large K_I such that the system overshoots and oscillates before coming to rest at the target. The other should be a ‘properly’ tuned response that comes to the target quickly, with little or no overshoot, and no steady-state error. This should be close to the critically damped response.

4)

Signoff

I have witnessed ___________________________________’s PID feedback control system – properly tuned, and tuned for overcompensating oscillation.

Witness __________________________ Date ________________