Control Systems

- Given a desired value, increase/decrease our output until we see matching feedback value
- Simple control algorithm:
    - As the command changes, we change
        - Software reacts to it as an event and move up/down
    - Otherwise, maintain desired command value
    - Note:
        - Speed of change is limited by our sample rate
            - We don't want to change our output faster or we won't see our changes
            - Can't sample faster or we see erroneous data
    - In many cases, this is sufficient
        - Especially will slow feedback and slow change requirement
            - Img 72
        - When is this not enough?
            - 1. Large jumps
                - Img 73
            - We could simply take larger steps with our ramping
                - This leads to overshoot
                    - Img 74
            - 2. Maintaining a value can be hard
                - Img 75
                - We can't get an exact value so we continuously bounce
                    - Due to:
                        - Lack of accuracy
                        - Changing conditions
        - We need to adjust for these 2 error that are going to occur and minimize them

PID (Proportional, Integral, Derivative)

- Standard control algorithm used for decades comes from mechanical engineering
- Can us "any" combination depending on our needs: P, PI, PD, PID (P is neccessary)
- Our input is either the measured output or an error signal
    - Difference between command value & curr input
- Can be an open loop or closed loop system
    - IMG 76
        - Open: aren't measuring the true impact of controlling
        - Closed: feedback is what we're controlling
    - E.g. controlling ground speed
        - Open: measure RPMs at wheel
            - Dependent on surface

        - Closed: LIDAR giving us actual speed
    - Open loop are based on approximation (...)
    - Closed loop feedback are more accurate (...)
    - Goal: always have a closed loop system

Implementing a PID:

- General code structure
    - **State = readInput();**
    - **Output = updatePID(cmd-state(error), state); (...)**
    - **genOutput(output);**

Proportional:

- The error is multiplied by a constant (Gain) to determine where to go next
    - **pTerm = pGain * error;**
- Determines how much we're out and how much to push
    - Gain: how fast
- This is just rate adjusted simple control
- If we have too much response delay, we can't stabilize the output
    - Need to drive output fast enough to get something useful
    - But, too quickly(too much/high gain) causes oscillation
- A DC motor:
    - IMG77
    - If we do a P, updatePID() returns the pTerm
    - When to use P:
        - Slow change, not much noise

Integral:

- Adds longer-term precision
- Track state as the sum of all preceding input(error values)
- Integrator "remembers" all prior states
    - Lets us cancel out long term output errors
- Can't just use an I(integral)
    - Memory causes instability
    - Respond too late:
        - A new command is seen as a large new error
        - To catch up, we would accumulated error and adjust
- Use P to add in a level of the current command value
- Code:
    - **iState += error;**
    - **iTerm = iGain * iState;**
    - **…**
    - **return pTerm + iTerm;**
- Issues:
    - Sample rate defines how fast we accumulate error

- We can suffer from wind-up
- E.g. make a step change
    - I(integral) accumulates error
    - Our state grows until we have a very large value
        - Output keeps pushing for higher values until we get lots of negative error
    - Results in too much oscillation & never settles
    - Solution:
        - Define minimum & maximum values and trim state
            - Based on output min/max
- A motor:
    - IMG78
    - PI is used when we have slow change & noise
        - Long-term stability with few command changes
- Differential:
    - If we can't stabilize an operation with a P, we can't with a PI
    - We need a way to predict the future
    - What is derivative?
        - Rate of change over time
            - Velocity
            - $\frac{\Delta state}{time} = \frac{diff\ btw\ last\ \&\ curr\ val}{time}$
            - We track where our state will be
    - Code:

      dTerm = dGain * (dState-state); | dState : last , state : curr
      dState = state;
      …
      Return pTerm + iTerm + dTerm;
    - Problem: Differential control(itself) algorithm exaggerate noise
        - PD is good for rapid changes & low noise
    - A motor:
        - IMG79
        - Whole point of differential is get rid of overshoot
    - A full PID will account for where we are, how far away we are, and how quickly we'll get there
        - Adjust with each sample to compensate for new error, rate of change, disturbances, and commands

      Note: PID gives us an <u>offset</u> from our current value

PID Tuning

- How do we pick our gain values? (2 ways)
    a. Model the system
    b. Experimentation (method we will use)
- Algorithm:
    - Set all gains to 0

- Start with a low pGain (<= 1)
- Test
- Set dGain to ≈ 100 * pGain
- Increase dGain until you see oscillations r excessive overshoot
    - Backoff by a factor of 2 to 4
- Set pGain btw 1 - 100
    - We will see slow or oscillating operation
        - If oscillating
            - Drop by a factor of 8 - 10 until it stops
        - If slow
            - Increase pGain by 8 - 10x(factor) until it start to oscillates
                - Reduce by 2 - 4x
- Fine tune by x2, until happy
- Set iGain btw .0001 and .01
    - Increase
    - Want our change to stay fast, little overshoot, & not too close to oscillating

  Notes:
    1. All testing is under real conditions
    2. Sampling rate should be 1/10th to 1/100th of our desired settling time

       Usually, rate determines our settling time

Fuzzy Control

- In some systems we have multiple inputs to determine 1 output
- How do we combine them?
    - Weight is applied to each input
    - Transform into 1 input
    - Feed this 1 values into our PID
- Weighting depends on:
    a. Priorities
    b. Quality
- This relates to state transition via hysteresis
    - Multiple inputs feeding one hysteresis process
    - Give input various increment/decrement values
        - E.g.
            - input 1 : +2, -3
            - input 2 : +1, 0
            - input 3 : -1, +2 (weighting)
    - We can also have hysteresis processes with more than 2 states
        - E.g.
            - threshold 1 = on
            - threshold 2 = quick
            - threshold 3 = emergency