Drive System Control

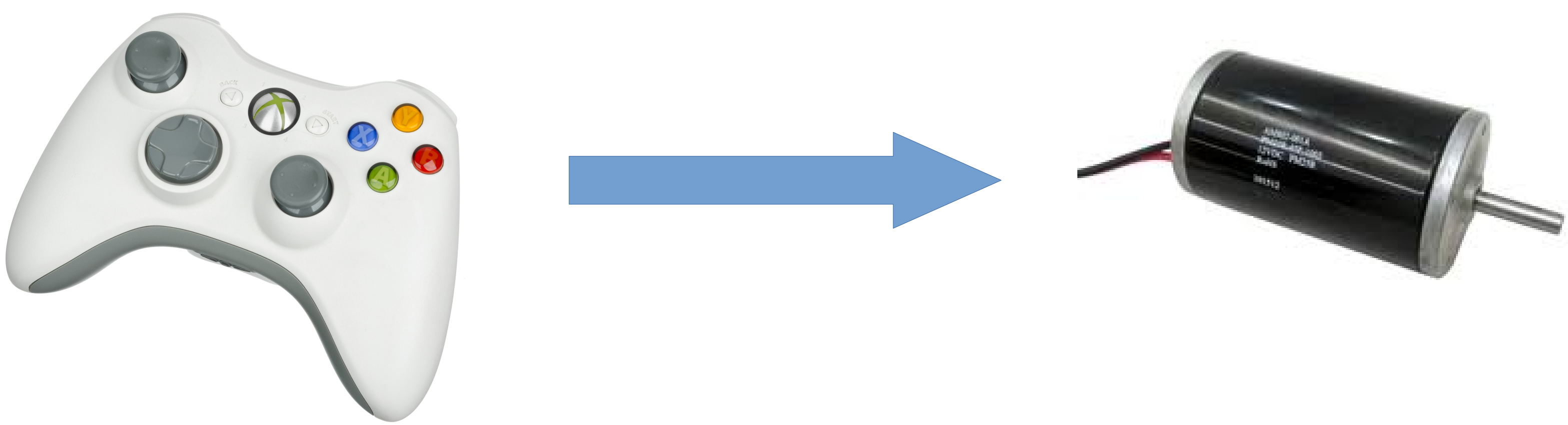
Course Overview

Course Overview

- Simple direct motor drive control
- Joystick Characterization
- Add speed / distance measurement
- Lag, 1st order Butterworth speed filtering
- Simple Motor Feedforward Algorithm – Open Loop
- Speed Setpoint Rate Limiting
- PID with feedforward – Closed Loop
- Simulation
- State Space Control

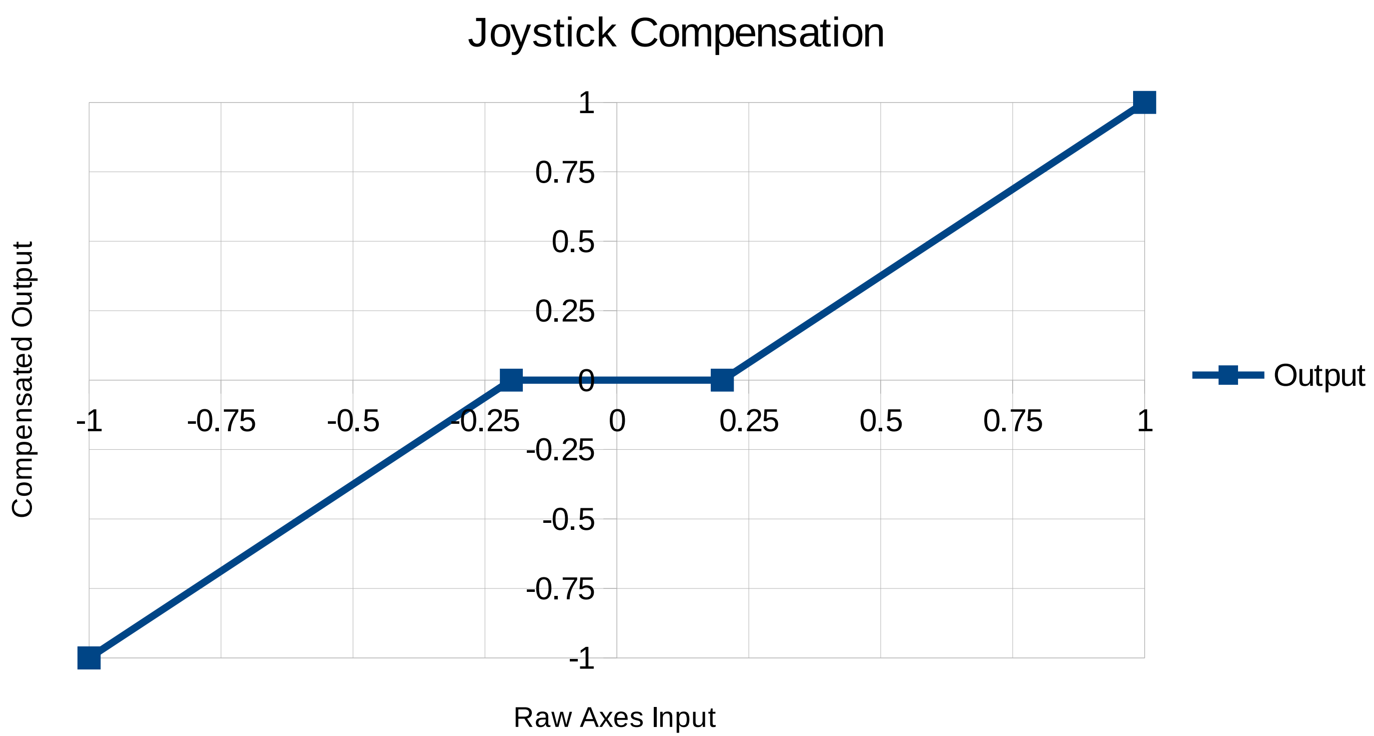
Direct Drive – Open Loop

- **Read joystick – write directly to motor controller.**
  - This works. Very limited ability to do any autonomous.



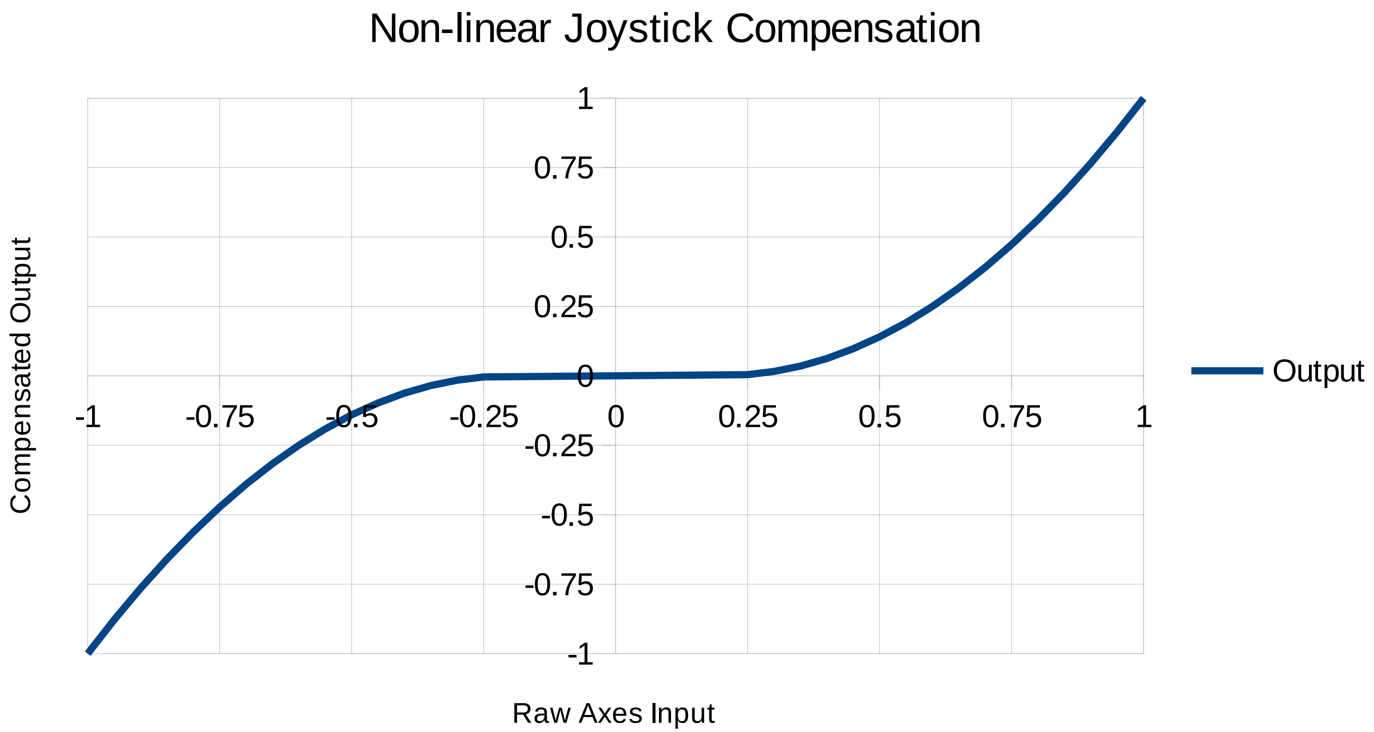
Joystick Characterization - Deadband

- **Next step – Joysticks don't read zero when they should**
  - Add some dead-band to avoid unwanted robot movement.



Joystick Characterization – Non-Linear

- **Next step – Some non-joystick characterization can be added.**
  - This potentially allows easier control at low speeds, with less control at high speeds.



Speed and Distance Measurement

- **Add encoders to read distance and speed.**
  - Encoders measure distance by counting as the wheel turns.
    - Need to specify distance/count, given encoder counts/rev.
  - Speed is calculated
    - FPGA calculates speed, but this can be jittery. Suggest calculating it yourself.
  - Scale these to useful units Feet, or Meters, Feet/Sec or Meters/Sec
  - TEST – put tape on wheels. Manually turn wheel 1 revolution. Resulting distance should be very accurate.
  - Write the values to Network Tables.

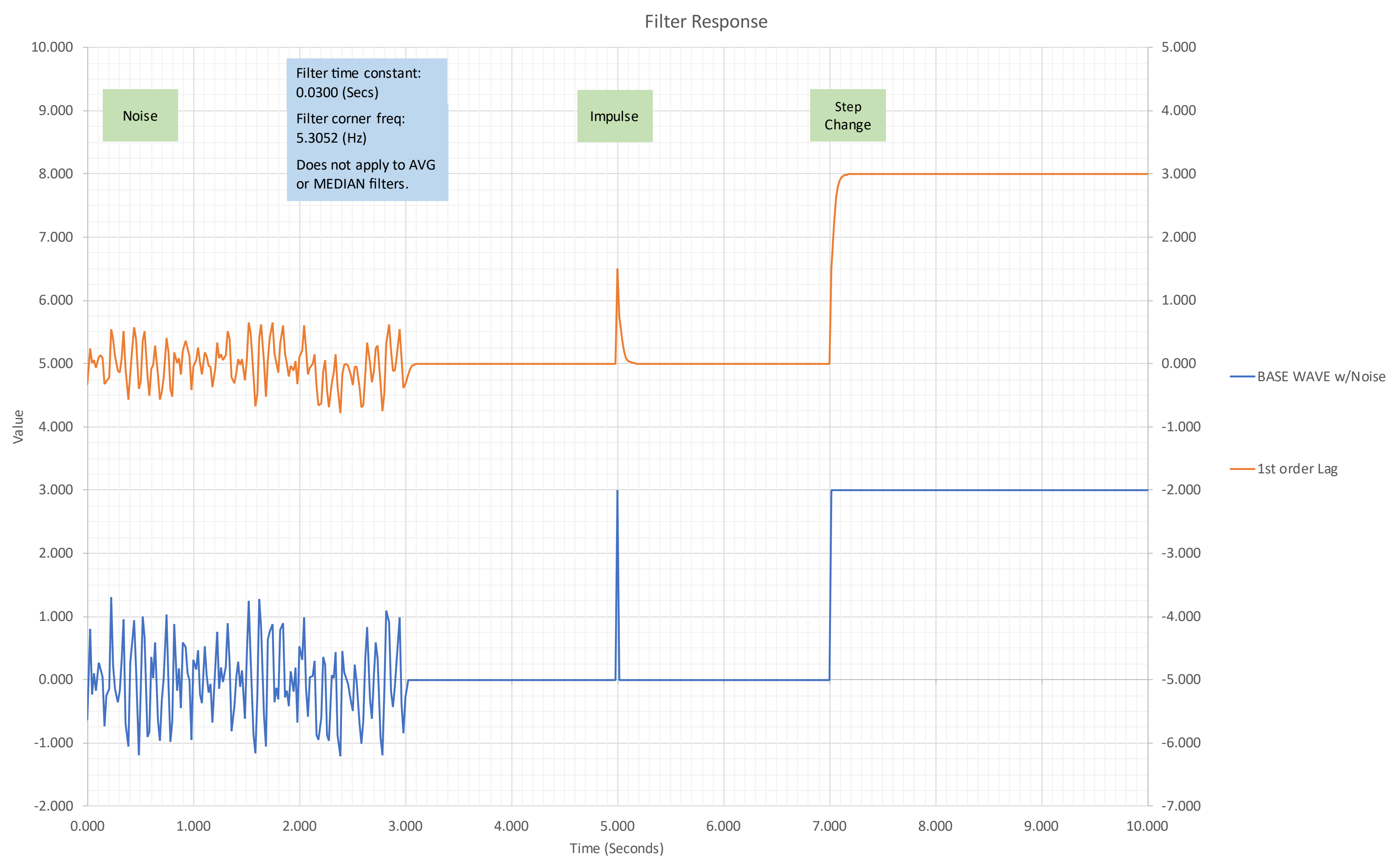
Calc Speed From Distance

- **This is numerical differentiation.**
- **Speed = d Distance / dt**
  - Numerically: Speed = (D2 – D1) / (T2 – T1)
  - Also, acceleration = d (Speed) / dt
  - Differentiation amplifies the noise in a signal and can introduce additional noise from time inaccuracy. If the valus is noisy consider using a filter.
  - We have had better success by calculating the rate instead of using the FPGA calculated rate returned by the encoder routines.

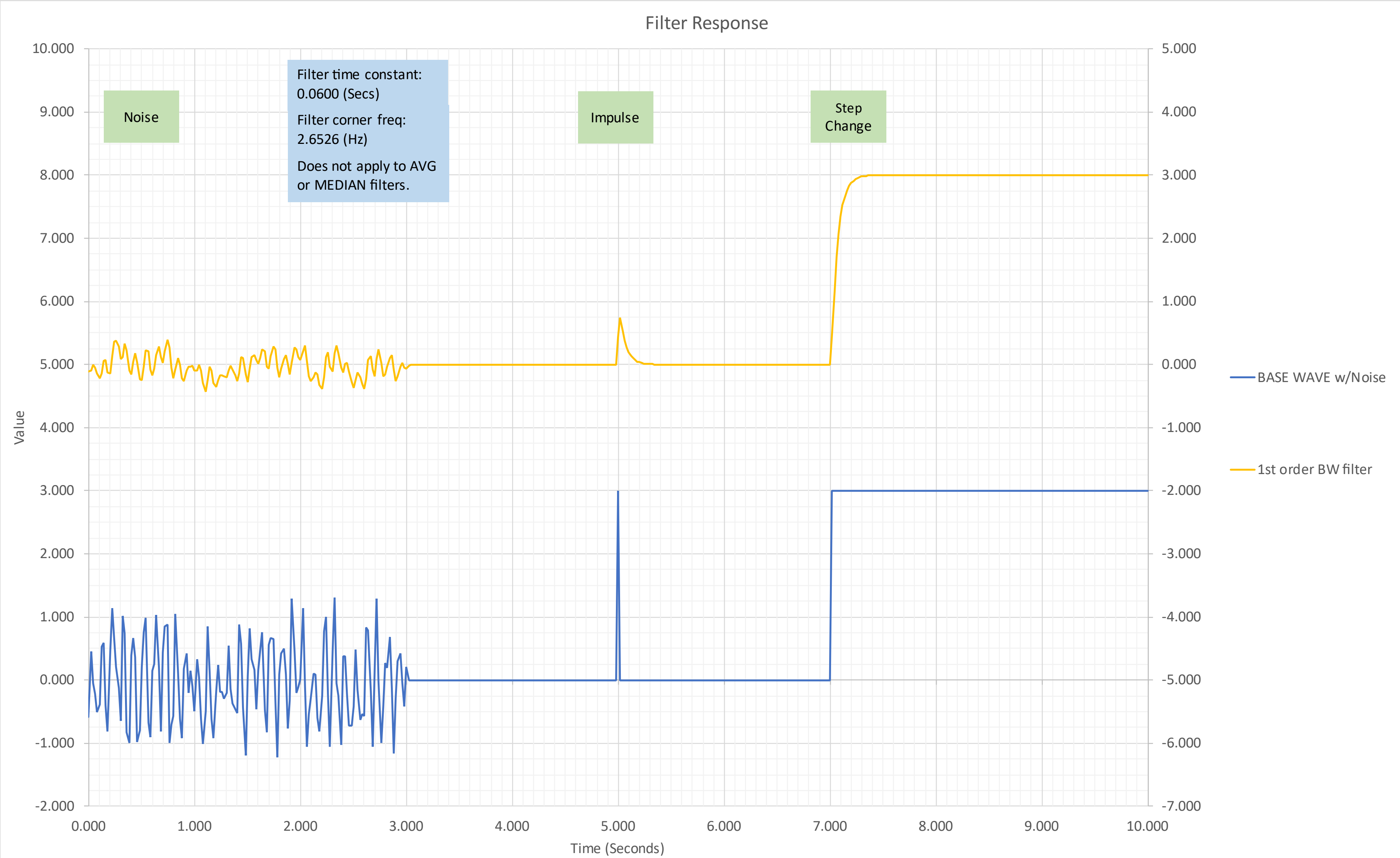
Filter Speed Measurement

- **If speed is still noisy consider adding a filter.**
  - Low pass filters reduce noise by reducing higher frequency portions of a signal at the cost of adding a some time delay to the signal.
  - Several different filters are available:
    - Average of last ‘x’ scans.
    - Finite differences filter – such as 1st order Butterworth filter (or first order lag filter) Specify time constant. Value = 63% of input after 1 time constant and 99% after 5 time constants.
    - Or create your own lag filter. Output = input * K + last_output * (K-1.0)   [1.0 >= K > 0.0]

1st Order Lag Filter Response – Tc = 0.03



Butterworth 1st Order Response – Tc = 0.06

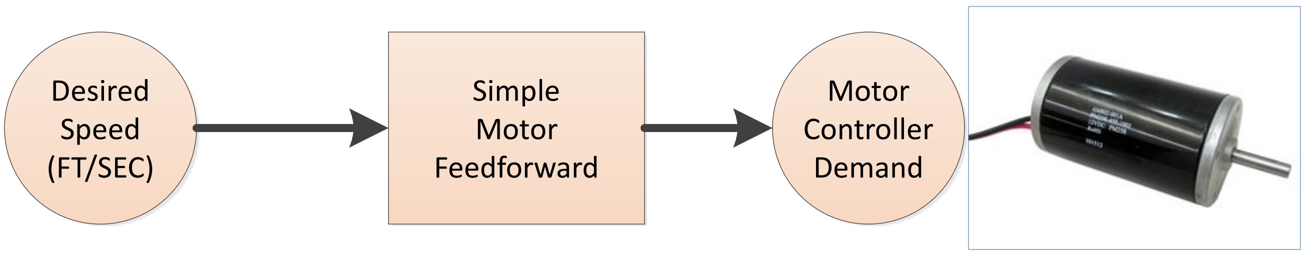


1st Order Butterworth Filter

- **Response is very similar to 1st order lag filter**
- **Equation (numerical implementation):**
  - Tc (Secs) = Time Constant, specified by user
  - Ts (Secs) = Sampling time, specified or calculated
  - Wd (Hz) = Cutoff Frequency = 1 / (2 * PI * Tc)
  - C = Cotangent(Ts / (Tc * 2))
  - D0 = (1 + C)
  - A0 = 1 / D0
  - A1 = 1 / D0
  - B1 = (1 – C) / D0
  - $Y_n = A0 * X_n + A1 * X_{n-1} - B1 * Y_{n-1}$

Simple Motor Feedforward – Open Loop

- **Estimates motor output needed for a particular speed based on testing**
  - Output = Sign(Speed Setpoint) * Ks + Speed Setpoint * Kv



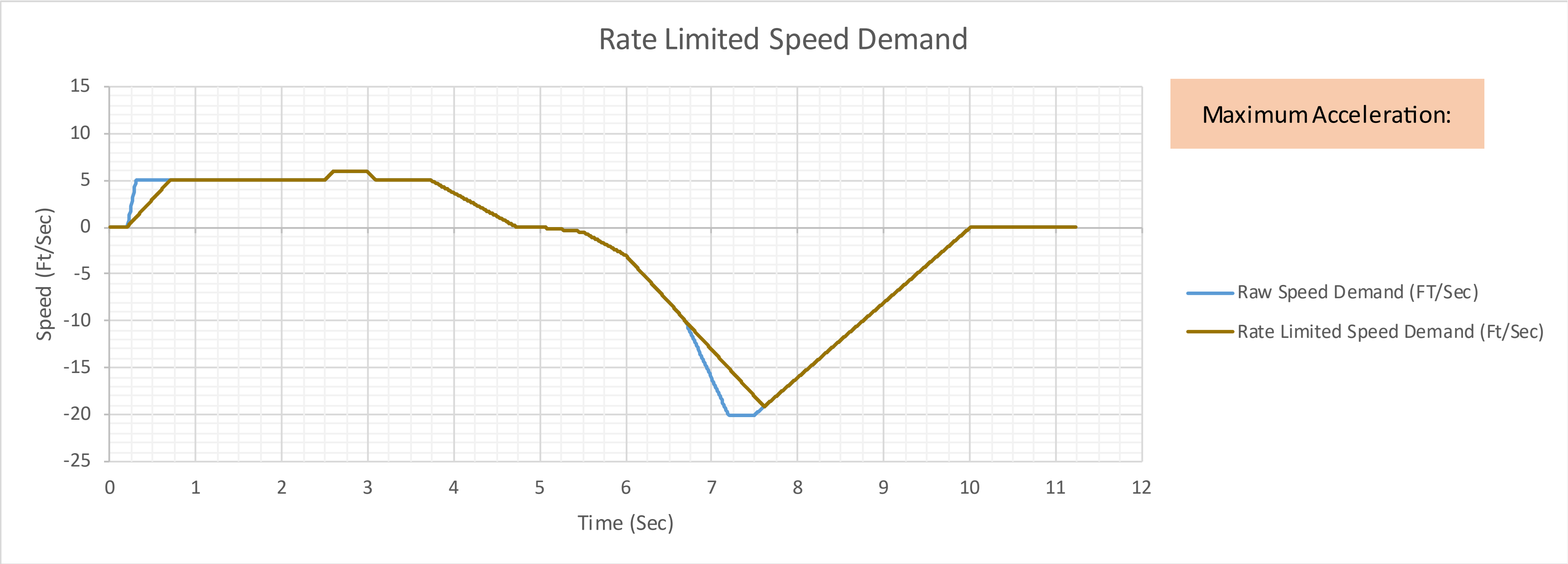
Simple Motor Feedforward – Open Loop

Determining constants

- Ks – static constant.
- Kv – slope constant.
- Drive robot at constant low motor output, say 0.15. Record average speed.
- Drive motor at constant high motor output, say 0.95. Record average speed.
- Optional – Drive motor at medium output, say 0.50. Record average speed.
- Calculate straight line coefficients. Y = mx + b. Ks = b, Kv = m.
- Ka – setpoint accleration constant. This is known as a “kicker” circuit. This can be left at 0.0.
- TEST

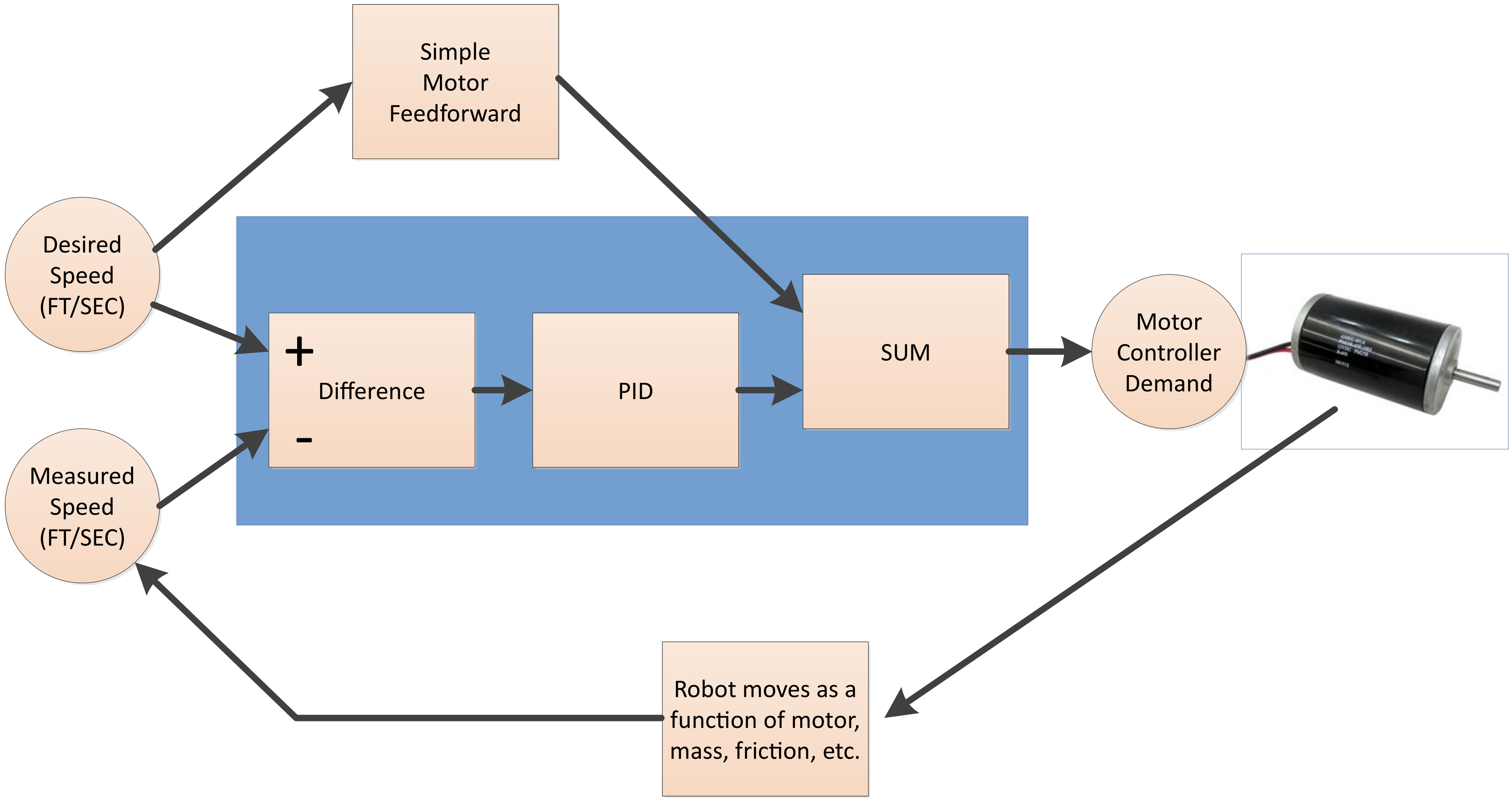
Speed Setpoint Rate Limiting

- **Why rate limit the setpoint**
  - Prevent over anxious drivers from toppling robot !
  - Reduces high motor current drain caused by fast changes in output.
  - Keeps robot "controllable"



Add PID with FeedForward – Closed Loop

- **PID calculates output as a function of speed error.**



PID

- **Mathematically adjusts output to correct for error**
- **ERR = (SP – PV)**
  - SP = Setpoint (speed demand – FT/SEC)
  - PV = Process variable (actual speed – FT/SEC)
  - In some cases the equation will be PV - SP
- **$ERR_{sc} = K_s * ERR$**
  - Scale (normalize) the error to be in the same units as the output. K_s = Max Output / Max PV

Mathematically:

$$Out = K_p \; x \; ERR_{sc} + K_i \; x \int ERR_{sc} + K_d \; x \; d\,\frac{ERR_{sc}}{dt}$$

PID – Notes

- **Suggest not using Kd**
  - Derivatives amplify noise.
- **Suggest using some Ki (and limiting)**
  - Things change through out robot's life. This can help to correct for that.
- **Ensure the output is within the allowed limits**
  - The individual terms could sum to a value larger than the allowed output!
- **This is a "non-interacting" implementation of a PID.**
  - Some tuning methods expect a classical PID

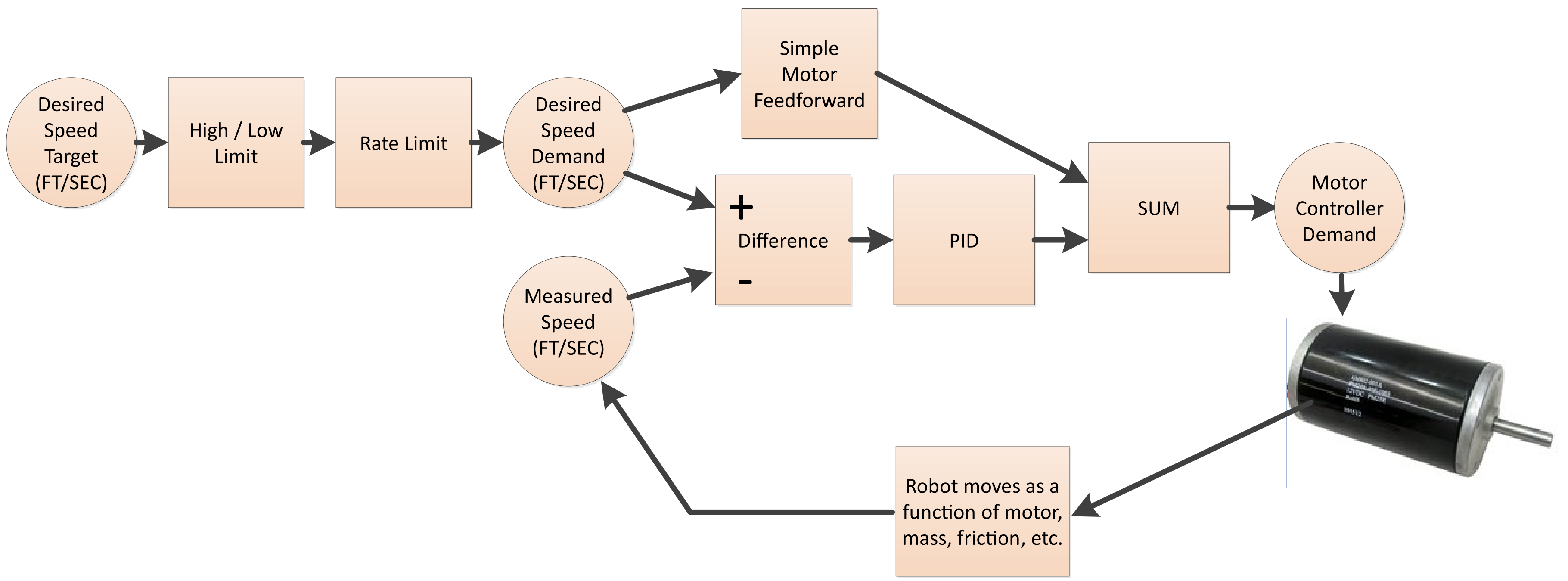
PID - Tuning

- **Goals**
  - Quickly match setpoint and process variable
  - Don't overshoot too much or oscillate
- **Trendi setpoint and process variable**
- **Repeatedly make step changes (near instantaneous movements) of setpoint and adjust Kp, Ki to obtain desired results.**
- **Leave Kd as zero. (Unless speed is not noisy.)**
- **Start with Kp between 0.5 and 2.0. Adjust Ki and Kp to get desired results.**
- **Many other more involved methods to tune PID.**
- **LabVIEW library has autotune.**

PID – Additional Things to Consider

- **Consider a manual mode that bypasses the PID**
  - Instrumentation breaks.
  - Different field conditions or robot conditions may invalidate the PID tuning
  - Stuff happens…
  - In manual, the PID output is set to zero. Only the feedforward is used. The PID internally "tracks" the output to all "bumpless" return to "auto" operation.
- **Note:**
  - The C++ and JAVA PID implementations don't appear to have a manual or auto mode, or manual mode tracking. The standard Labview "advanced" PID does have these built in.
- **Consider writing your own PID…**

Complete Speed Control Block Diagram



Alternatives

- **Much of this can be done in the motor controller**
  - Same concepts.
- **Alternative control – State space control**
  - Requires robot model

Simulation Demo

- **Demonstration of the “JSON” protocol simulation**
- **Differential robot is simulated using Differential Drive Train simulator.**
  - Works with real robot code
  - Should work with all supported languages
  - Accurate representation of motors
  - Does not include friction

Drive System Control

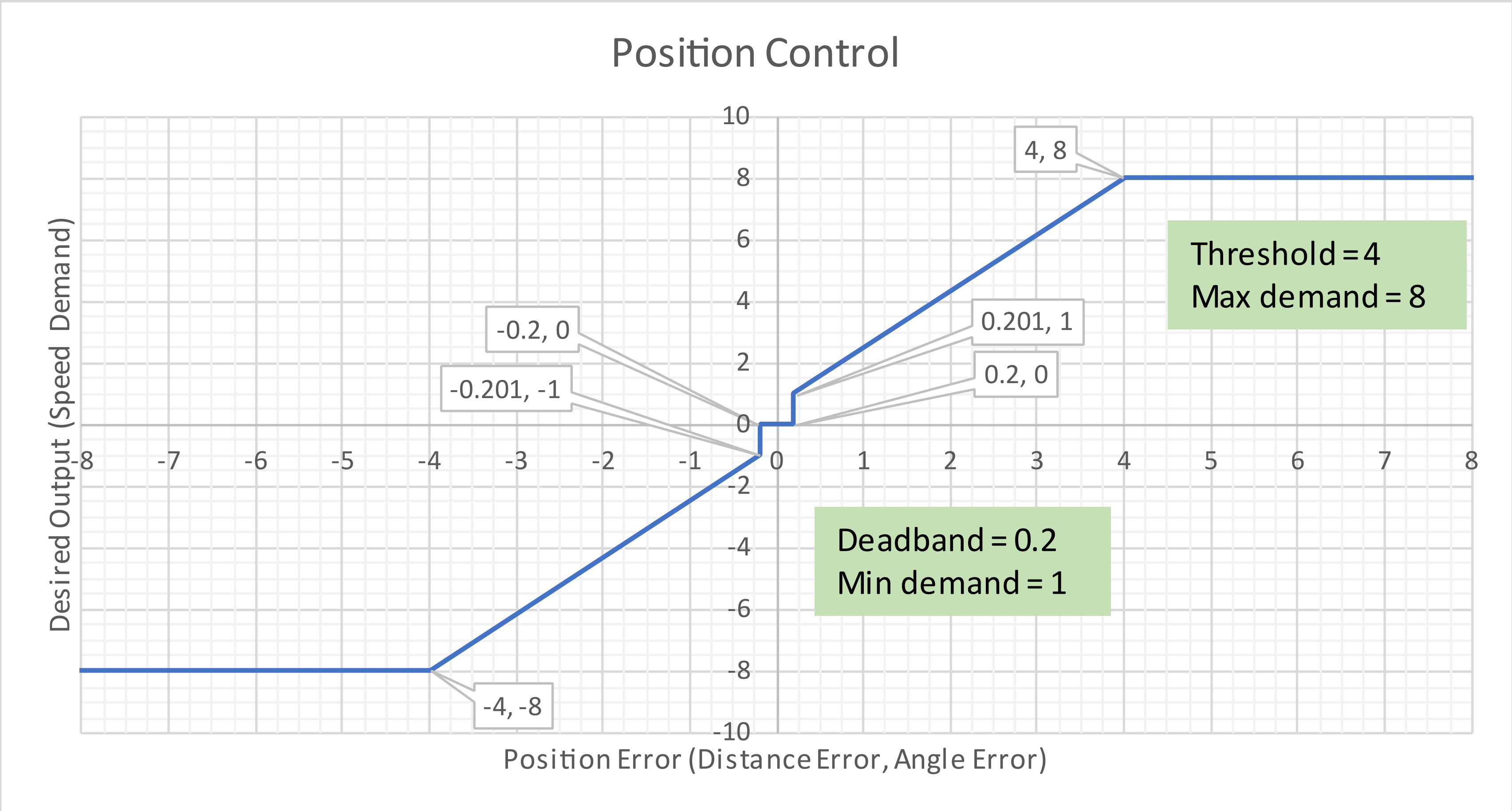
Extra – Position Control

Position Control Examples

- **Spin robot X degrees**
  - Target angle = starting angle + amount to turn.
  - Turn until Current Angle = Starting Angle
  - Angle measured using Gyro
- **Move robot forward X feet**
  - Target position = starting position + amount to move.
  - Move forward until Current Position = Target Position
  - Position measured by drive system encoders
- **These are the SAME problem. Consider:**
  - Start at rest
  - End at rest
  - Slow down when getting closer to target
- **YES, there are more sophisticated solutions…**

Position Control Algorithm

- **Input – Error (angle error, distance error)**
- **Output – Wheel Speed Demand**



Position Control Algorithm

- **Input variables**
  - Target position
  - Current position
    - Error is calculated from these
- **Input parameters**
  - Largest output
  - Smallest output
  - Error threshold
    - Where largest output starts to ramp to smallest output
  - Error deadband
    - How close is considered “on target”
    - Where output drops from smallest output to zero
  - Consecutive times within deadband to be considered done
    - Accounts for sliding through target

Position Control - Tuning

- **Max Speed**
  - Don't make it faster than your robot can handle
  - If this is too slow, it will take a long time to get to the target
- **Min Speed**
  - If this is too fast, the robot will overshoot the target and oscillate
  - If this is too slow, the robot may stop, never reaching the target, if it can't control at the slow speed.

Position Control - Tuning

- **Threshold**
  - If this is too far away, the action will take longer than needed
  - If this is too close, the robot's momentum will cause the speed to be greater than desired
- **Deadband**
  - Set this to get the accuracy you need
  - If this is too large, accuracy will suffer
  - If this is too small, robot may oscillate around target.

Position Control – Setting the Parameters

- **Set initial parameters based on how your robot performs**
- **Test and tune parameters to balance speed and accuracy.**
  - This will require time and a working robot!
- **The same tuning should work for all similar movements**
  - One set of tuning for moving forward / backwards
  - One set of tuning for spinning movements
- **For any autonomous action also include a timeout!!!**
  - If you measured the distance wrong and hit the wall, the robot needs to continue…