The Code Behind High Hopes

Design and Programming of the 2011 Control System

Kerry Loux
Team 3167

Overview

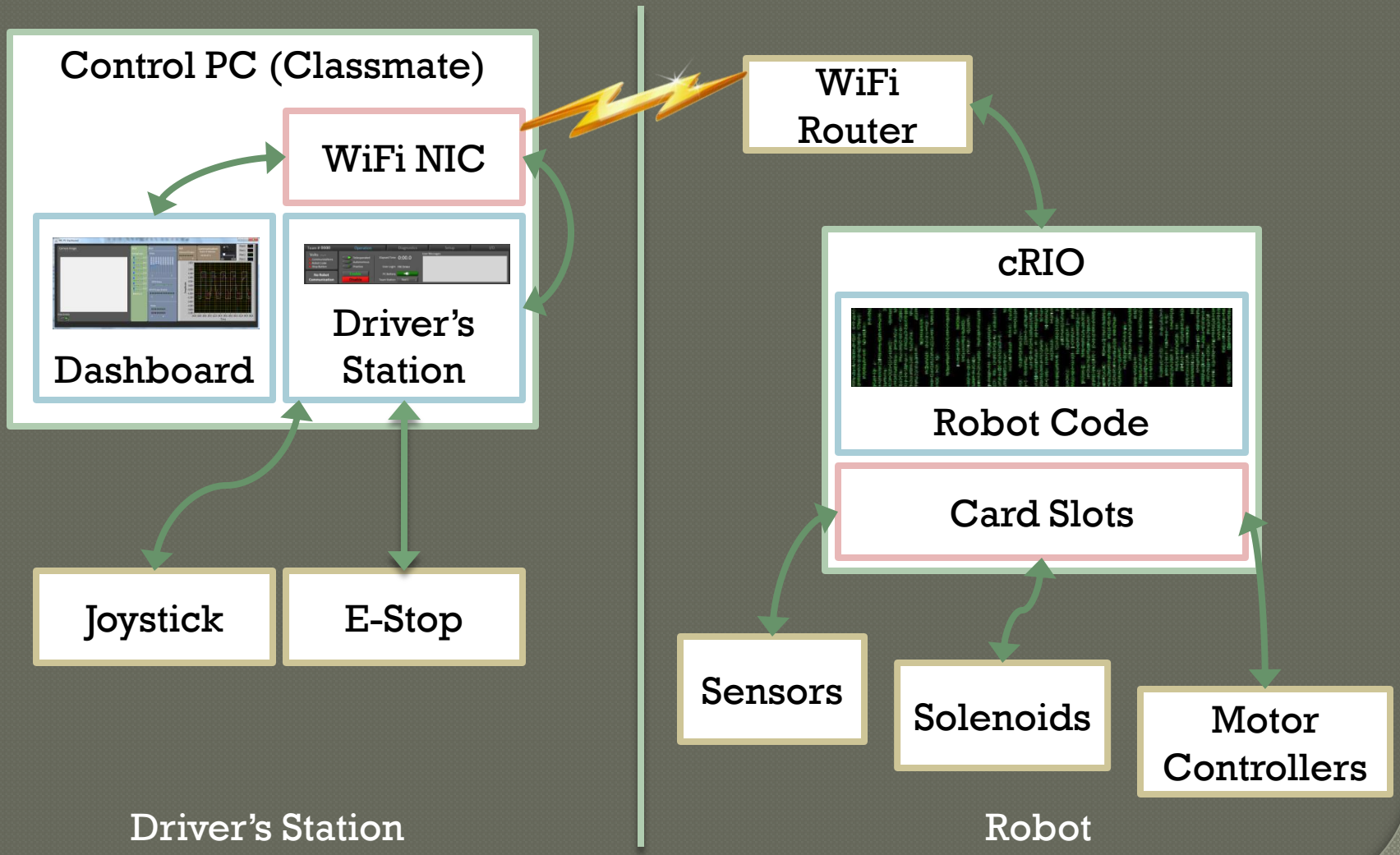
- The Basics
- Software Design
- Program Architecture
- Utility Classes
- Sensors
- Code Maintenance and Testing

The Basics

- Java
- Object-oriented
  - Classes → Objects
  - Fields → Variables/Parameters
  - Methods → Functions/Subroutines
- Netbeans

  Compiler

Robot System Architecture



Software Design

- What does the robot do?
  - Define your game strategy
  - How can software help?
    - Respond to operator inputs
    - Respond to sensor inputs
- What elements are likely to be reusable in future games?

State Machines

- Manage "states" instead of conditions
  - Easier and cleaner to implement
  - Less prone to error
  - Controlled state transitions instead of system conditions
- Three main methods
  - Enter state
  - Perform appropriate actions
  - Exit state
- Don't always need all three methods
- Lots of information online
  - Google "state machines" or "finite state machines"

Condition-Based

```
if (startedFunction1 && !readyForFunction1 && !doneFunction1)
{
    startedFunction1 = true;
    // Do stuff
}
else if (readyForFunction1 && !doneFunction1)
{
    // Do stuff
    if (FinishedFunction)
    {
        doneFunction1 = true;
        if (ConditionsToGoToFunction2)
            readyForFunction2 = true;
        else
            readyForFunction3 = true;
    }
}
else if (readyForFunction2 && !startedFunction2 && !doneFunction2)
{
```

Many different conditions define appropriate

Managing multiple flags is a common source of errors

State-Based

```
switch (state)
{
case State.Function1:
    // Do stuff
    if (isDone)
    {
        if (ConditionsForFunction2)
            nextState = State.Function2;
        else
            nextState = State.Function3;
    }
    break;

case State.Function2:
    // Do stuff
    if (isDone)
        // On to the next state
    break;
}
```

Finite states make code much easier to read and debug

No guessing what the next state will be

Handling State Transitions

What would condition-based equivalent Entry() look like?

```
if (nextState != state)
{
    ExitMethod(state);
    state = nextState;
    EntryMethod(state);
}
```

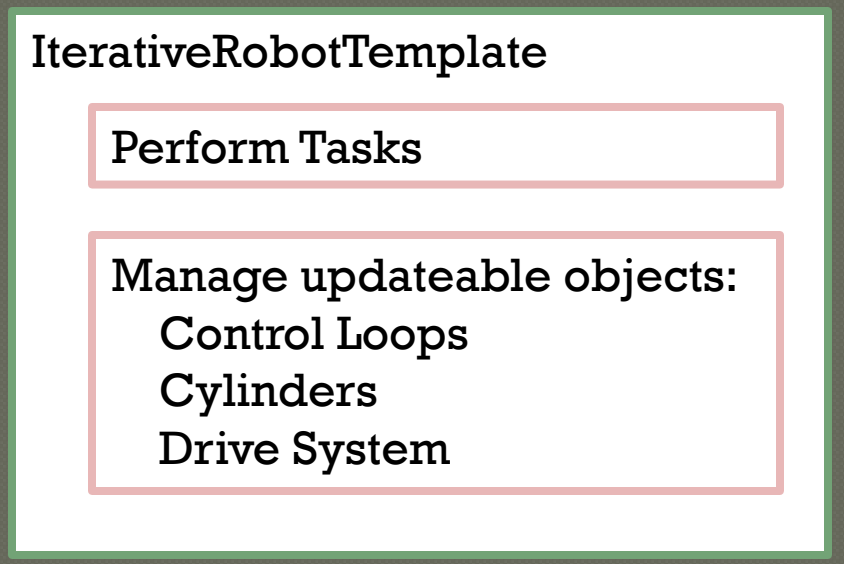
What would condition-based equivalent Exit() look like?

```
void ExitMethod(oldState)
{
    switch case (oldState)
    {
    case State.Function1:
        // Do stuff
        break;

    case State.Function2:
        // Do stuff
    }
}
```

Program Architecture

- IterativeRobotTemplate class
- TaskQueue and TaskBase classes
- Updateable objects



IterativeRobotTemplate Class

- Provided with libWPI (from FIRST)
- State machine
  - Disabled
  - Autonomous
  - Teleoperated
- Entry methods
- Periodic methods (50 Hz)
- Continuous methods (as fast as possible)

Core Loops

```
public void teleopPeriodic()
{
    if (taskManager.OkToDrive())
        drive.Drive(stick);

    taskManager.DoCurrentTask();
    elevator.Update();
    miniBotArm.Update();

    UpdateDSDataCluster();
}

public void teleopContinuous()
{
    if (button2.HasJustBeenPressed())
    {
        taskManager.ClearAllTasks();
        drive.Reset();
    }
```

Respond to joystick inputs

Perform tasks

Update cylinders and elevator height

Respond to button presses

Same every time

Commands change here

TaskQueue Class

- Several tasks to be performed in specific order

  Example:
  - Follow line
  - Hang tube
  - Return to "normal" configuration

- Main methods
  - Add to list (takes TaskBase derivatives)
  - Do current task
  - Clear list

Queues

- First-in, first-out (waiting in line)
  - Push (add to end of line)
  - Pop (get from front of line)

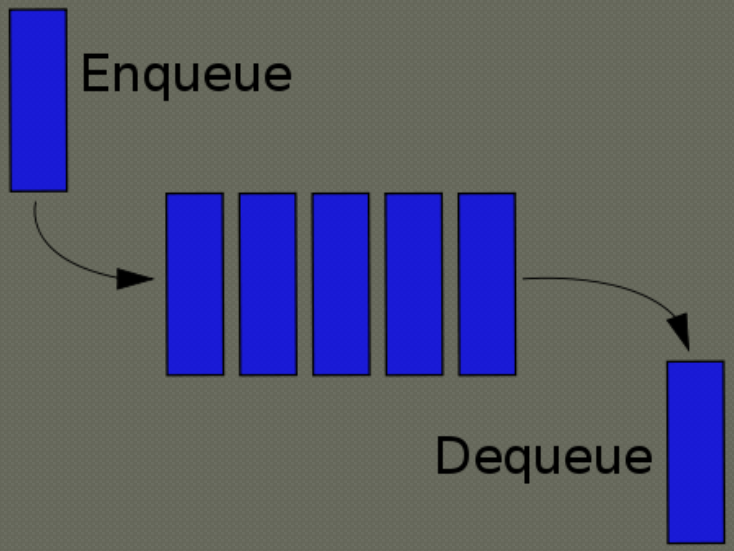


Image Source: Wikipedia
http://en.wikipedia.org/wiki/Queue_%28data_structure%29

TaskBase Class

- Abstract class
- Derivatives inherit
  - Perform action method
  - Exit method
  - OK to drive? method
  - Am I done? method

TaskBase ←
FollowLine ←
HangTube ←
DeployMinibot ←

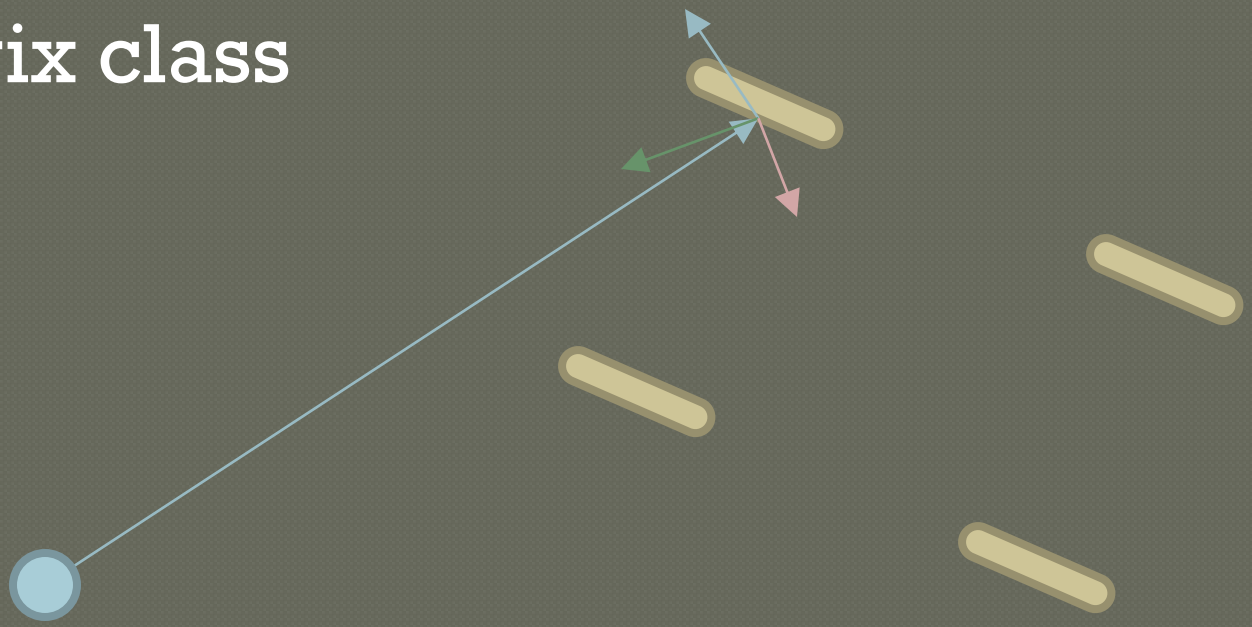
Animal	Eat	Speak
Cat	Catnip	Meow
Horse	Hay	Neigh
Lion	Zebra	Roar

Updateable Objects

- Updated once per cycle
  - Periodic loop
- Closing control loops
- Examples
  - Ladder (and fork)
  - Holonomic drive
  - Holonomic positioner
  - Navigation

HolonomicDrive Class

- Maps v_x, v_y and ω into wheel speeds
- At least three wheels, no upper limit
- Separate document explaining math
- Matrix class



HolonomicPositioner and Navigation Classes

- Navigation uses various sensors to estimate position and speed
- Provides "fusion" of data from many sensors to get best position estimate possible
- Feedback to close position loops
- Output from position loops is input into velocity loops

Utility Classes

- Joystick button handlers
- Filters
- Bang-bang controllers
- PID Controllers
- Limiters
- Pneumatic cylinders

Joystick Button Handlers

cRIO Is faster than your finger

```
Joystick stick;
boolean pressHandled;
```

```
if (stick.getButton(1) &&
    ! pressHandled)
{
    pressHandled = true;

    // Do stuff
}
else if (!stick.getButton(1))
    pressHandled = false;
```

The solution:

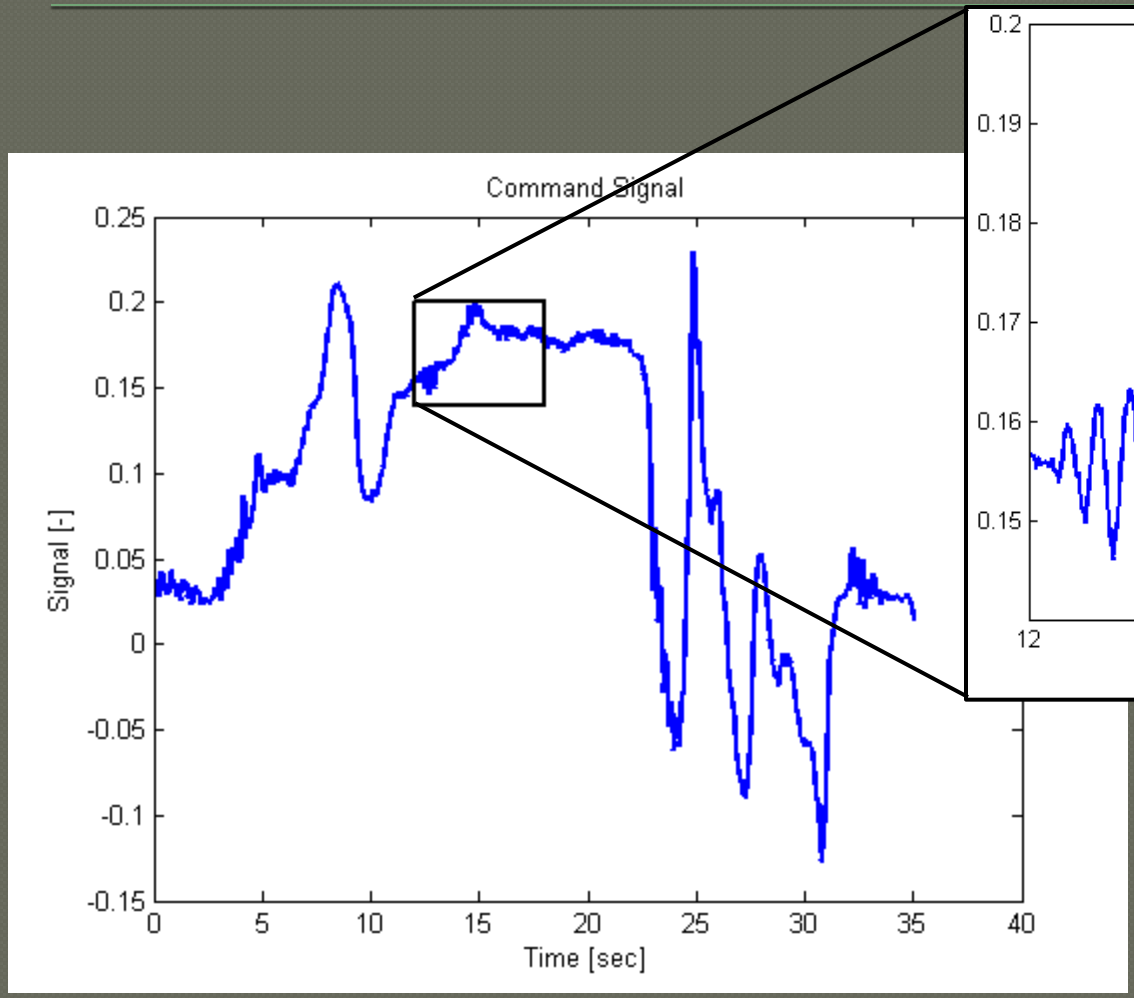
```
Joystic stick;
trigger = new JoystickButton(
    stick, 1);
```

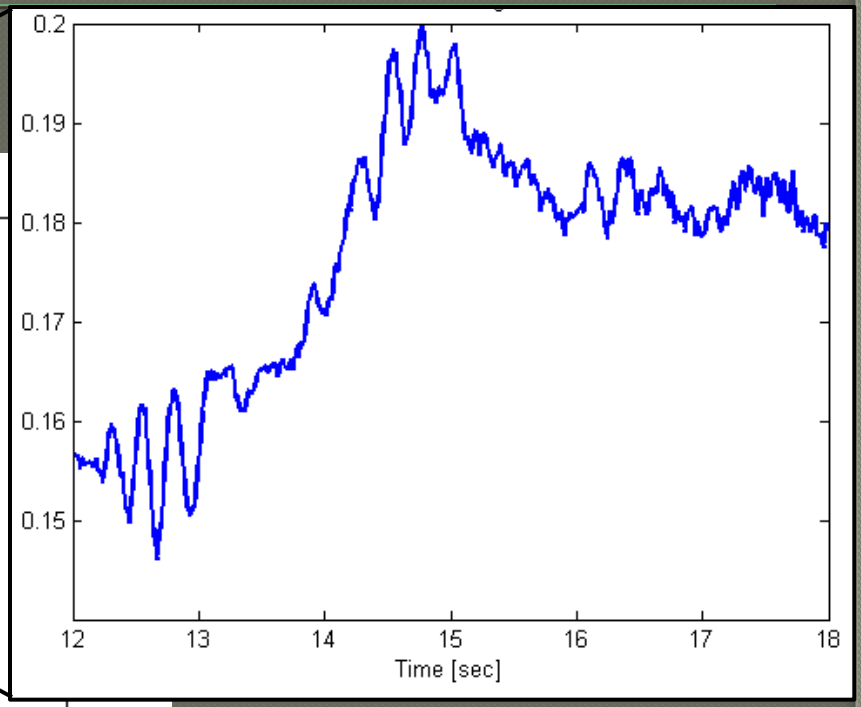
```
if (trigger.
    HasJustBeenPressed())
{
    // Do stuff
}
```

Filters

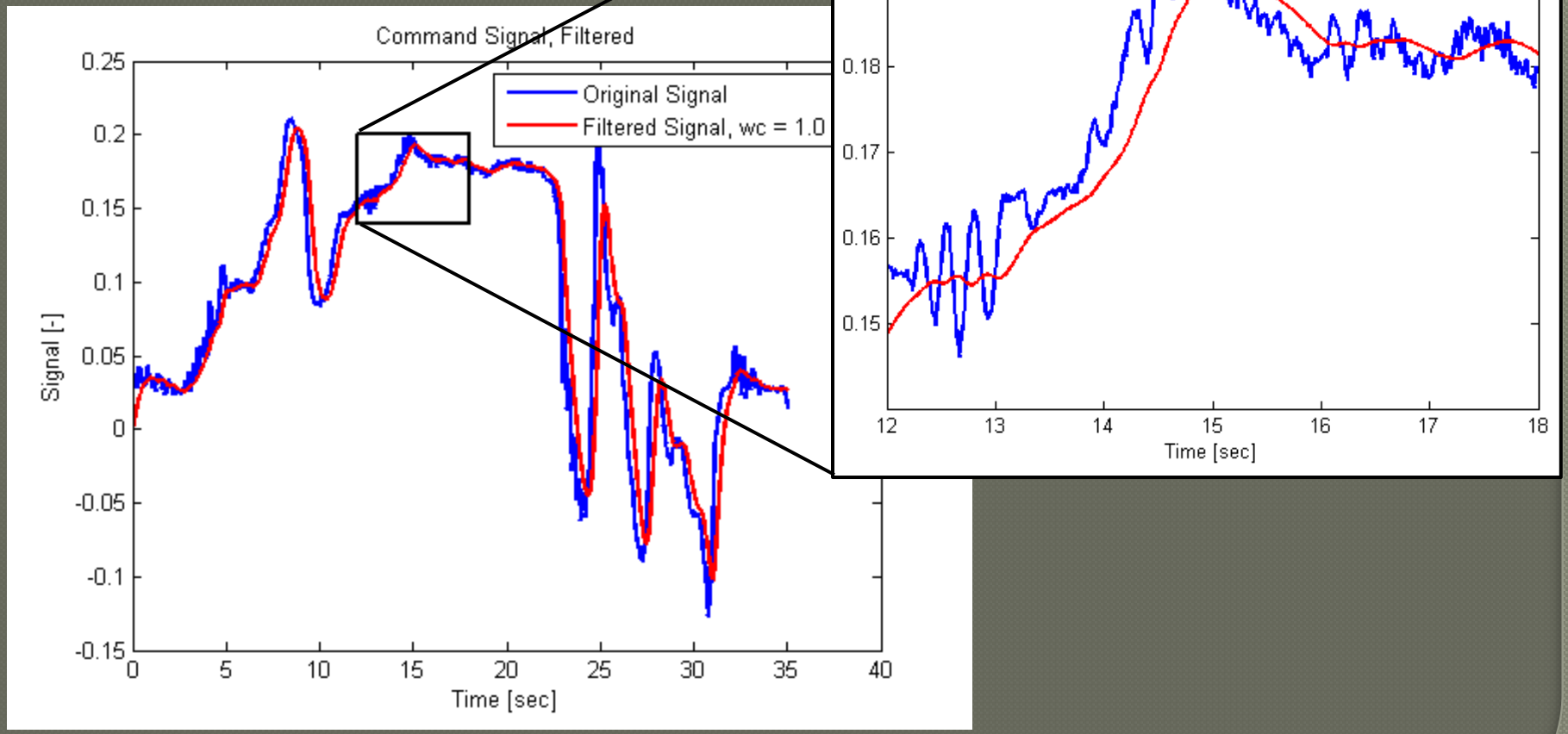
- Real-world signals are noisy
  - System response
  - Sensor accuracy
  - Electromagnetic interference
  - Derivatives are noisy
  - Derivatives of derivative – forget about it
- Smoothes the signal (PID+)
- Removes "high frequency" content
- Google "low pass filters"

Real-World Signal Noise

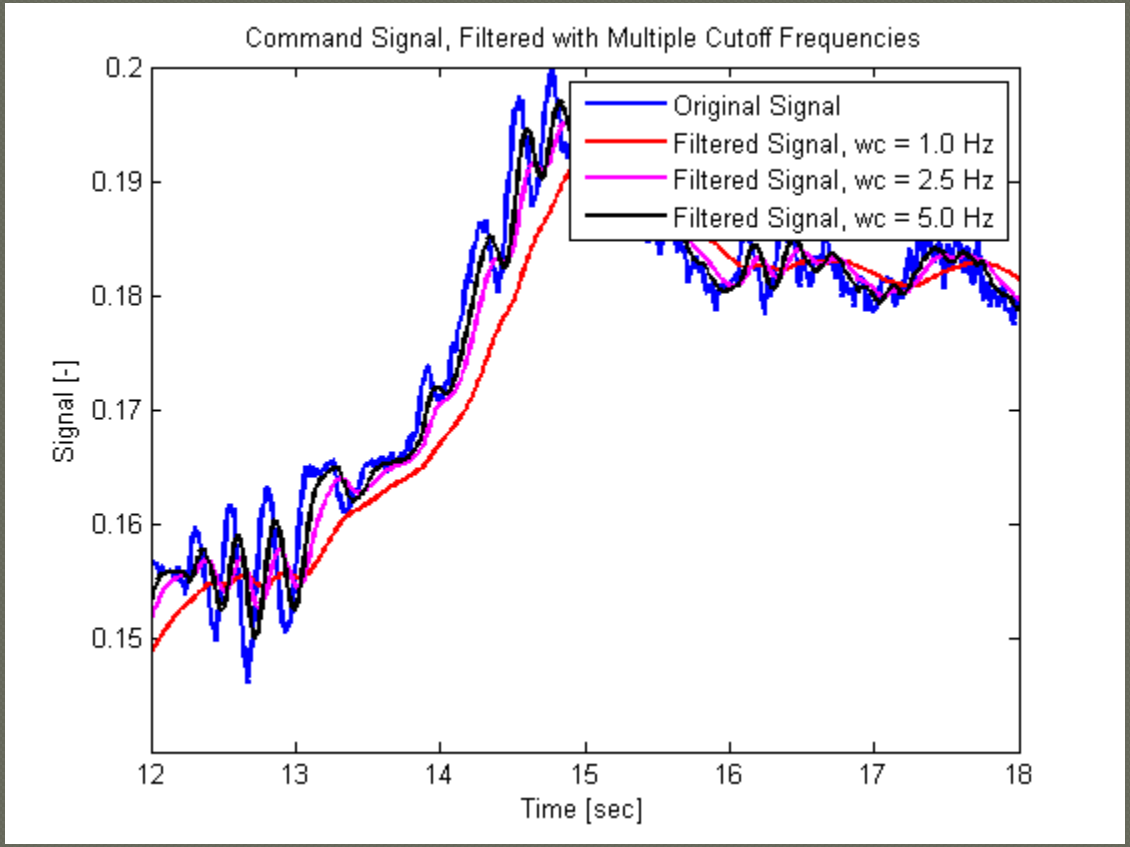




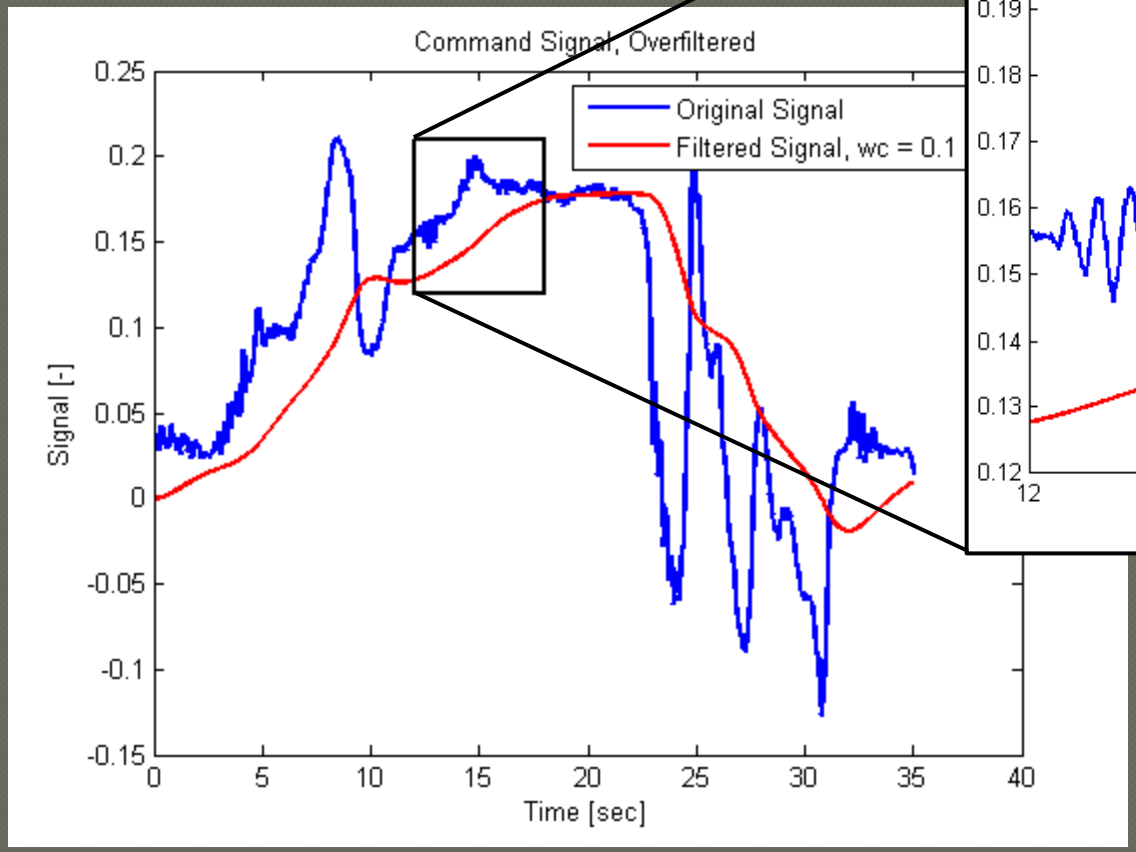
Using Filters

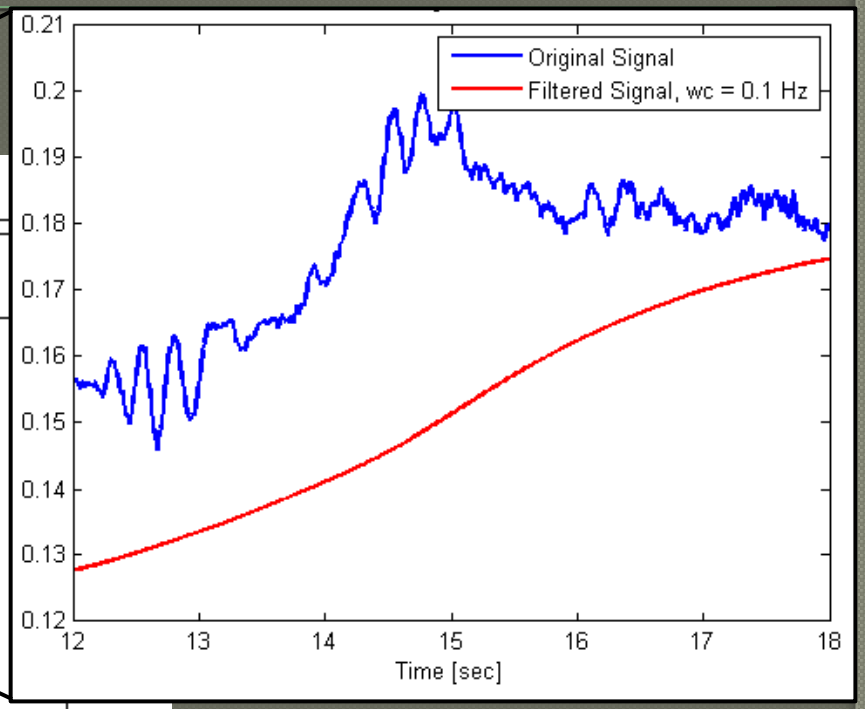


Choosing Cutoff Frequencies



Overfiltering

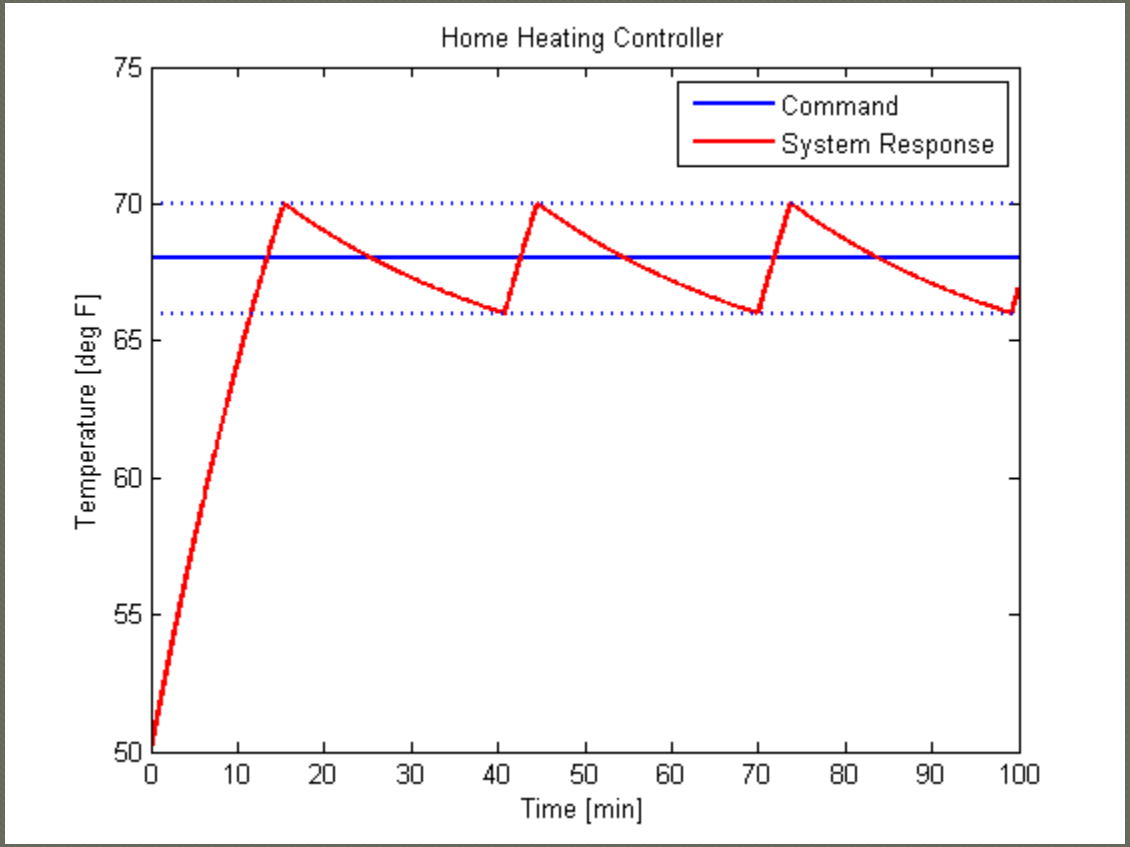




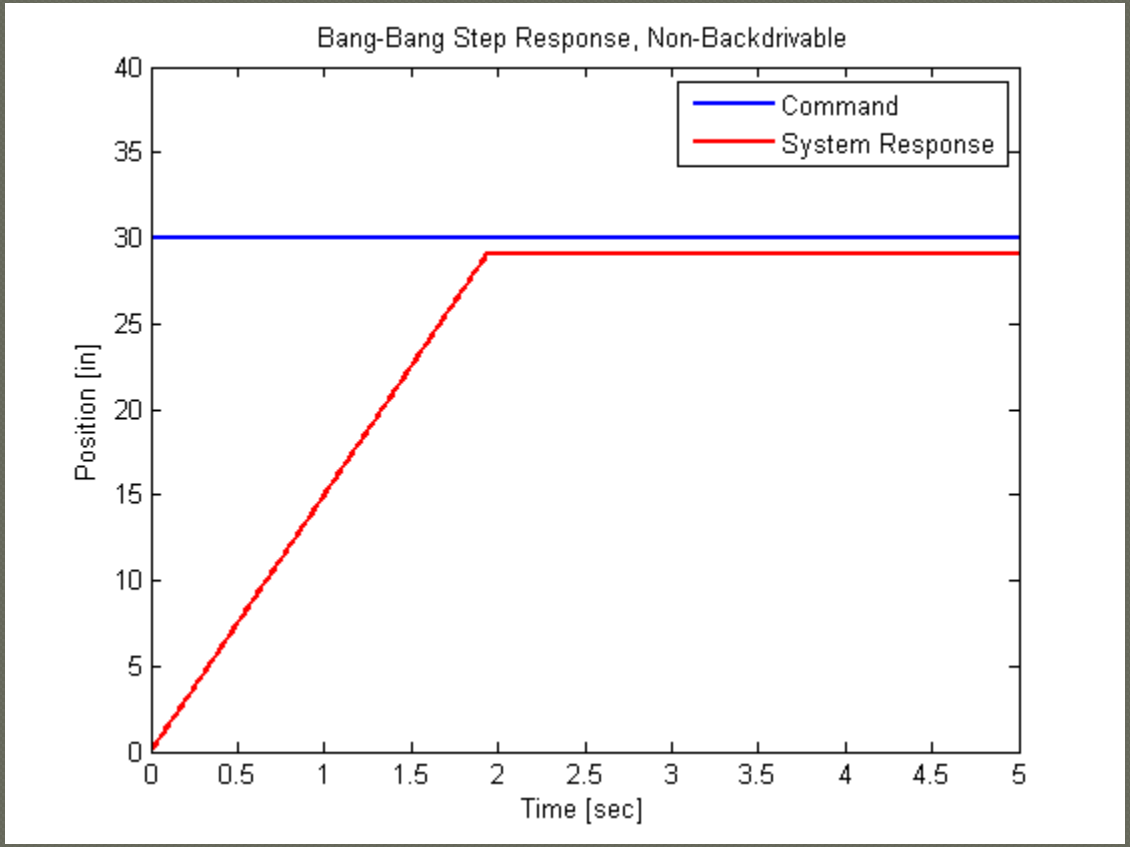
Bang-Bang Controllers

- On-Off control (or Forwards, Backwards, Off)
- Simplest closed-loop controller
- Works well for some applications

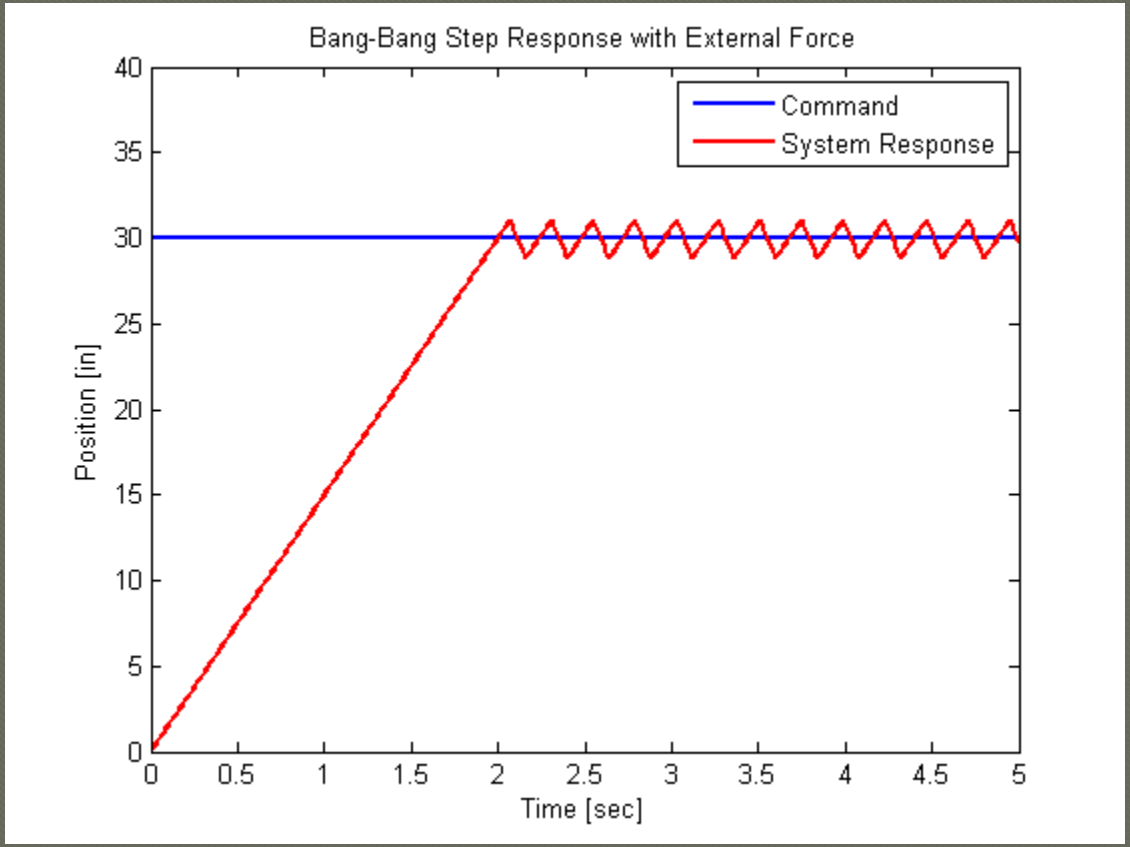
Common Applications



Non-Backdrivable System (or No External Forces)

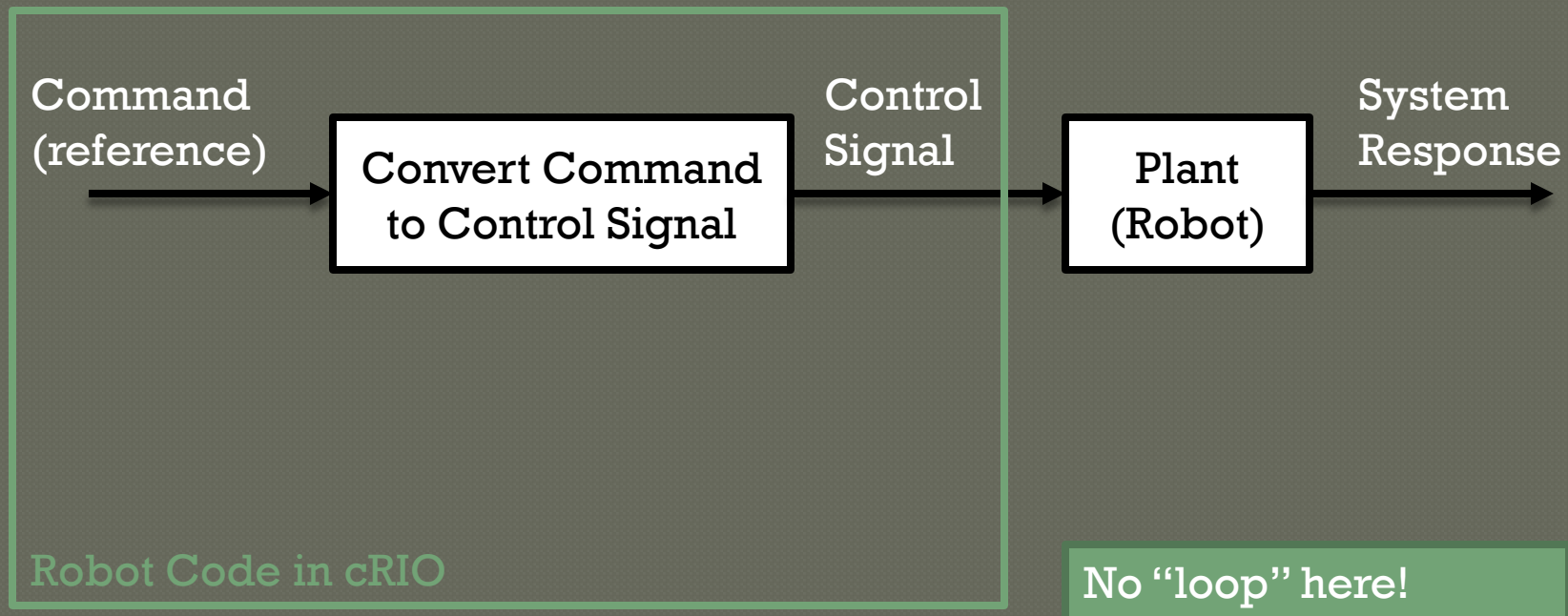


Backdrivable System



Intro to PID Controllers

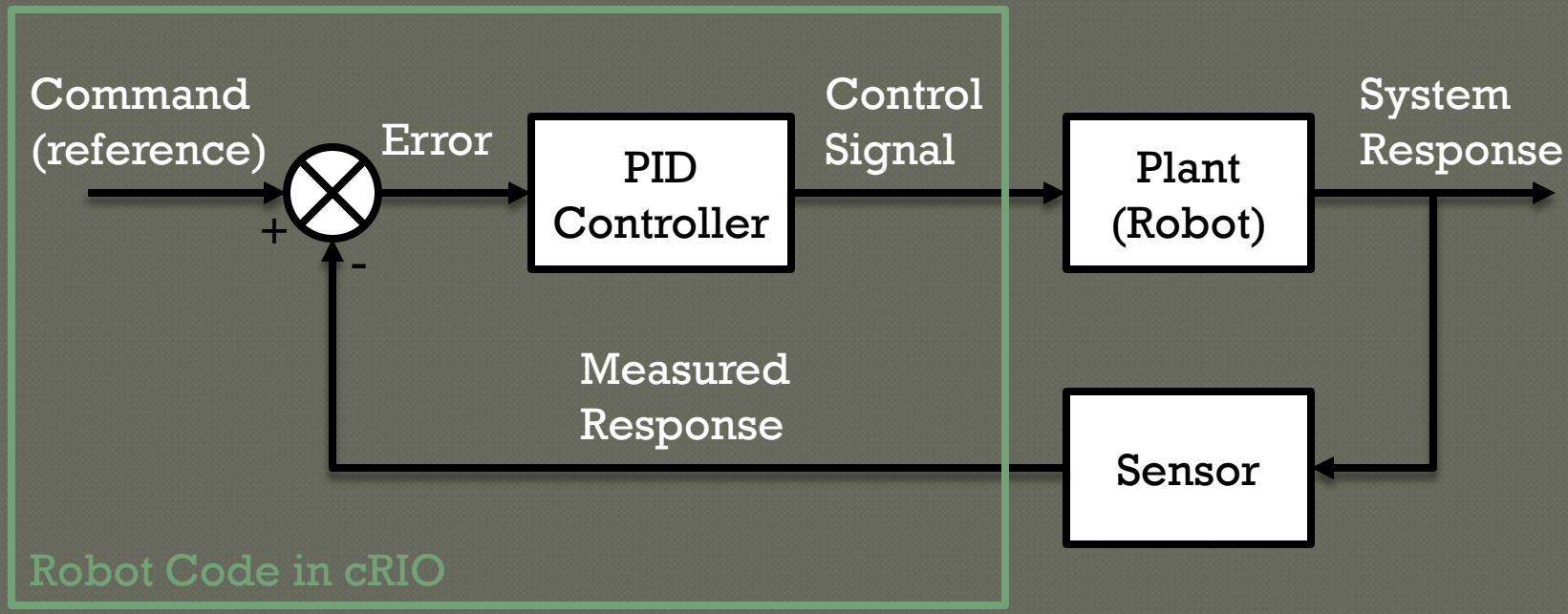
Open-loop



No "loop" here!
Driver is "in the loop"

PID Controllers

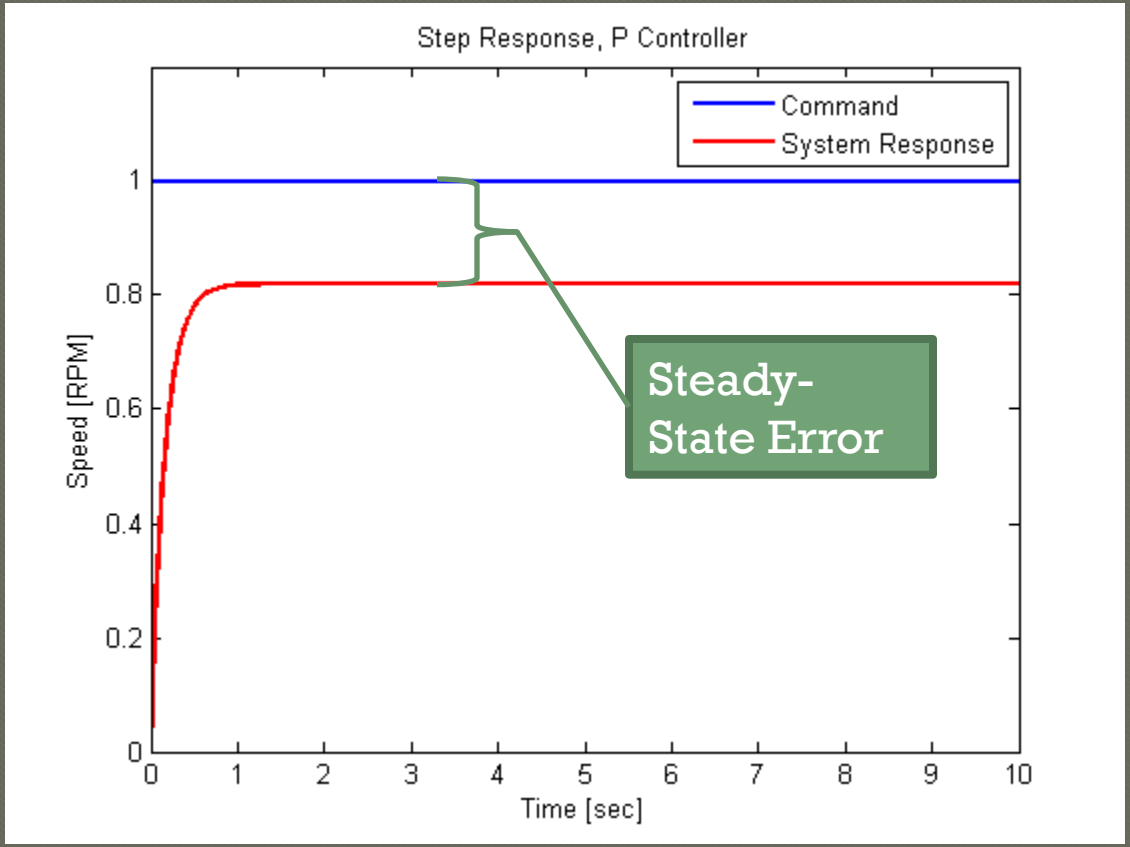
Closed-loop (PID)



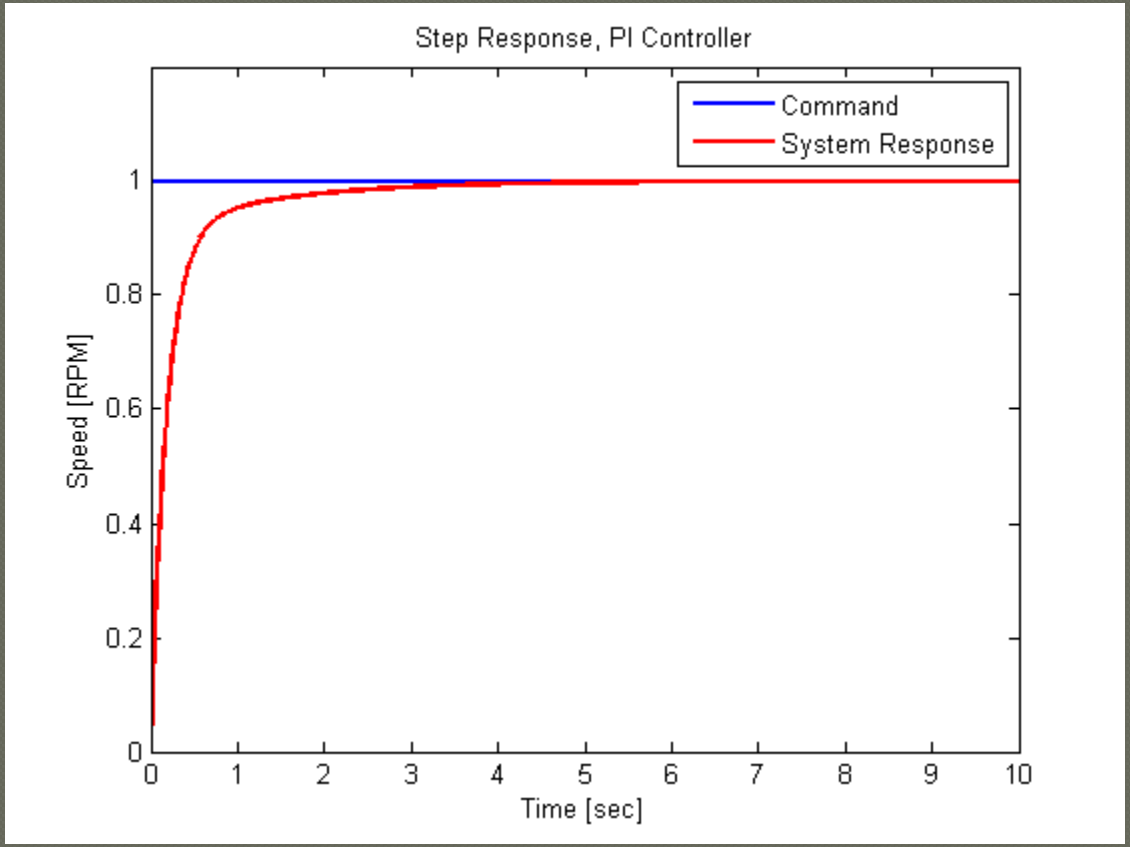
Tuning PID Controllers

- Adjust gains
  - Proportional gain – high speed gain
  - Integral gain – steady-state error gain
  - Derivative gain – damping gain
- Step response
  - Command has infinite acceleration
  - Hard to follow – taxing for controller
  - Linear system assumption
- Lots of information online

P Controllers



PI Controllers



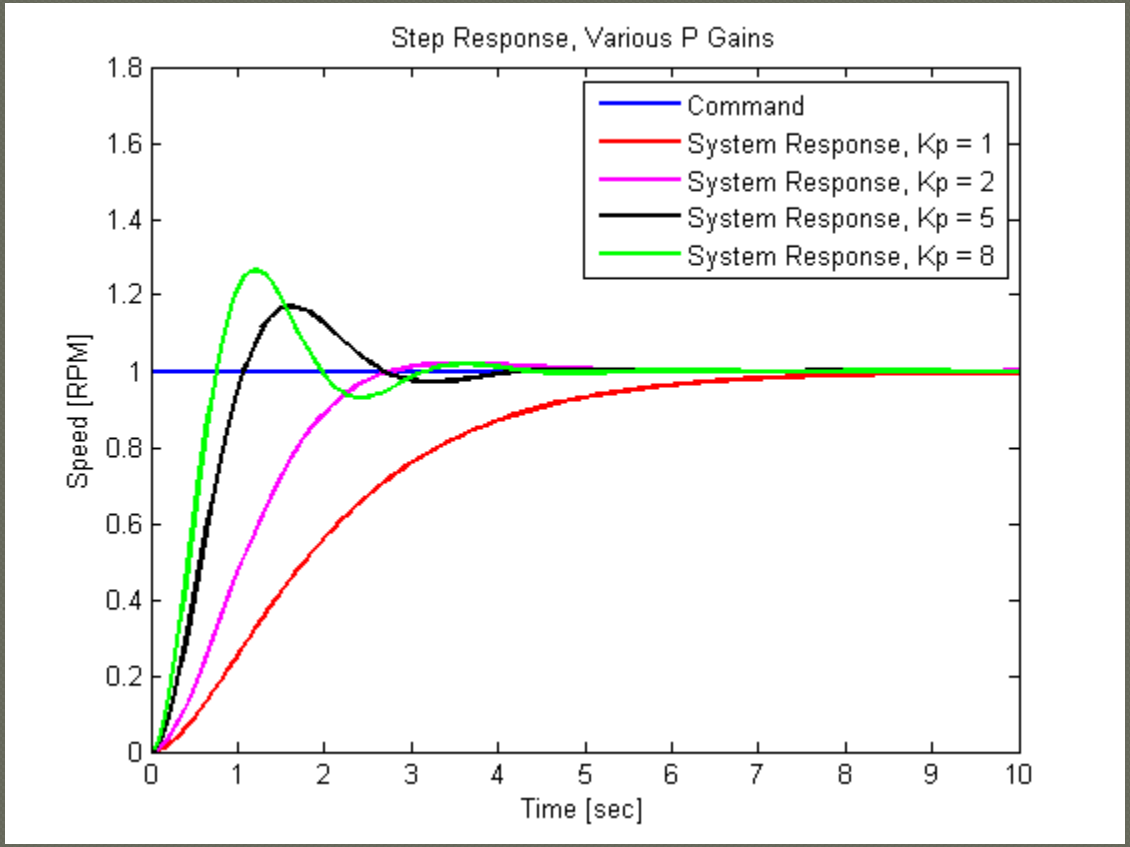
Notes on Integral Term

- Linear system assumption
  - Probably not valid, but generally close enough
- Saturation
  - Motors have limited torque and speed
  - Non-linear effect
- Proportional term doesn't care as much
- Integral term **does** care
  - Error grows without bounds (windup)
  - Artificially limit error integral (anti-windup)

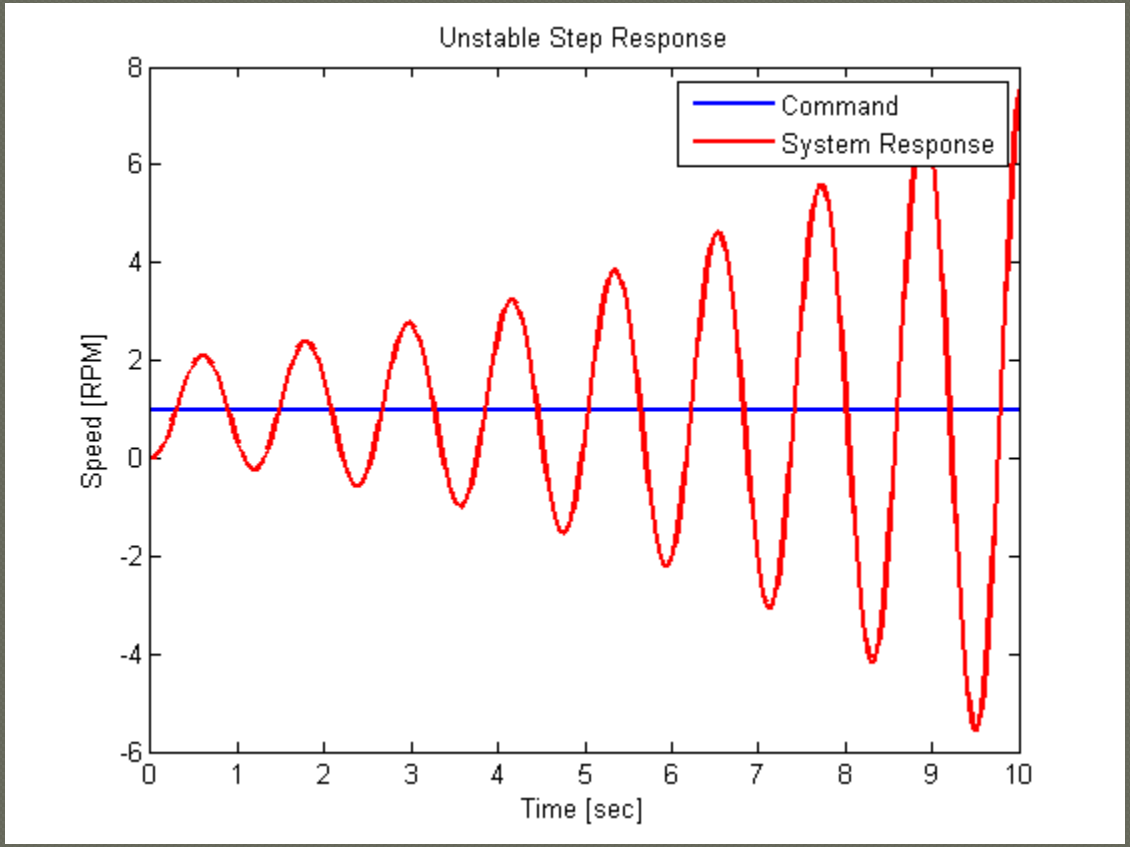
PID Control

- Skipping D term – limited benefit vs. added complexity
- Necessary in some cases
- Not used for High Hopes, but capability exists in utility class

Effect of Adjusting Gains

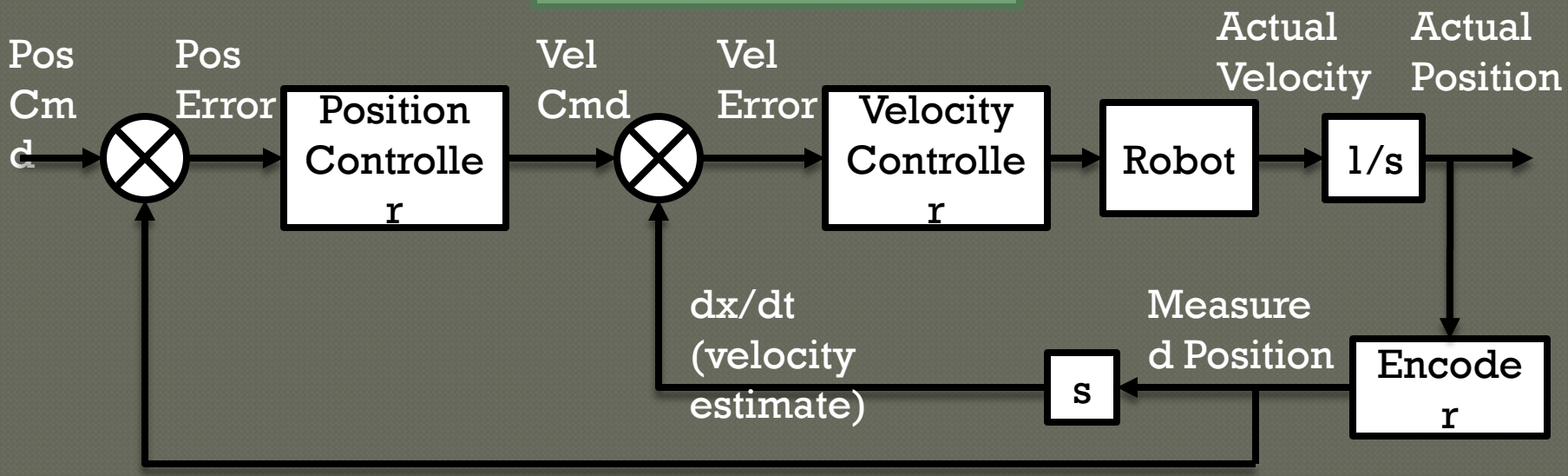


Stability


Unstable Step Response
Command
System Response
Speed [RPM]
Time [sec]
8
6
4
2
0
-2
-4
-6
0
1
2
3
4
5
6
7
8
9
10

Cascaded Loops

Like an "early warning" system – improves dynamic response



Cascaded Loops

- Can do "mapping" between loops
- Think of output of position loop as 3-element vector
- Think of input to velocity loop as 4-element vector
- Based on robot geometry and gearing, transform position loop output into velocity loop input

PID Uses on High Hopes

- Velocity loop for each wheel (4)
- Position loop for each DOF (3)
  - Cascaded loops
  - Position update from various sensors
    - Wheel encoders
    - IR Sensors (minibot deployment)

Notes on Control Strategies

- Can't just keep increasing gains
- Fast responding, high accuracy systems
  - Hard drive readers
  - Printer cartridge carriages
  - Solid-state electronics (motor control, power supplies)
- How are they realized?
  - Good **system** design
  - Stiff mechanical components, short drivelines
  - Low-lag sensors
  - Minimize/eliminate backlash
  - Add damping (friction)
- PID is only the beginning…

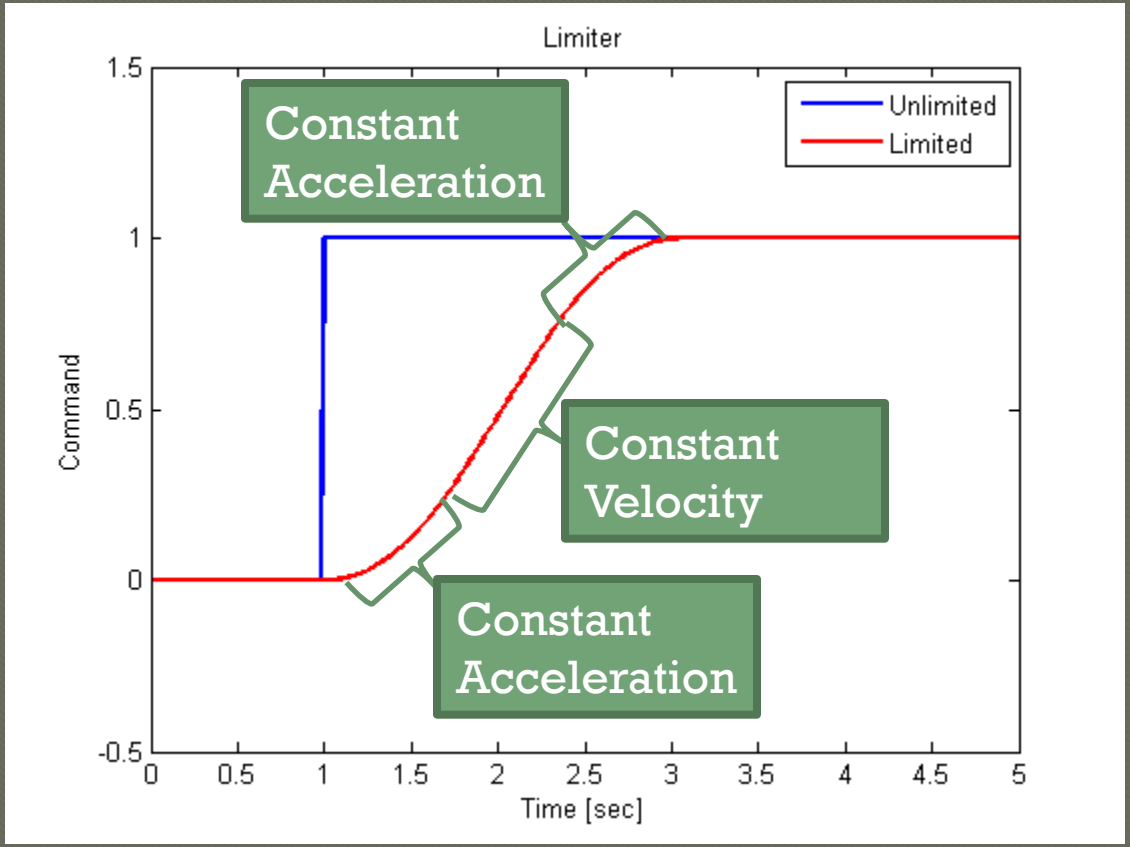
Limiters

- Step commands are hard to follow
- Never give a step command
- Example:
  - Operator slams joystick forward and left
  - Robot can't move as fast as joystick

    Limit acceleration command
  - Combination of full forward and left exceeds motor speed

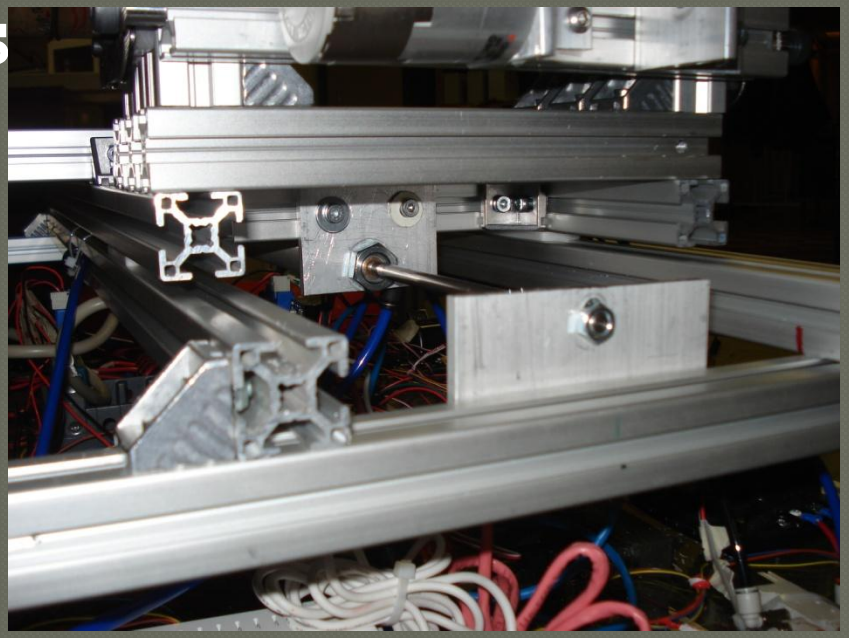
    Limit velocity command

Limiters


Limiter
Unlimited
Limited
Constant Acceleration
Constant Velocity
Constant Acceleration
Command
Time [sec]
1.5
1
0.5
0
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

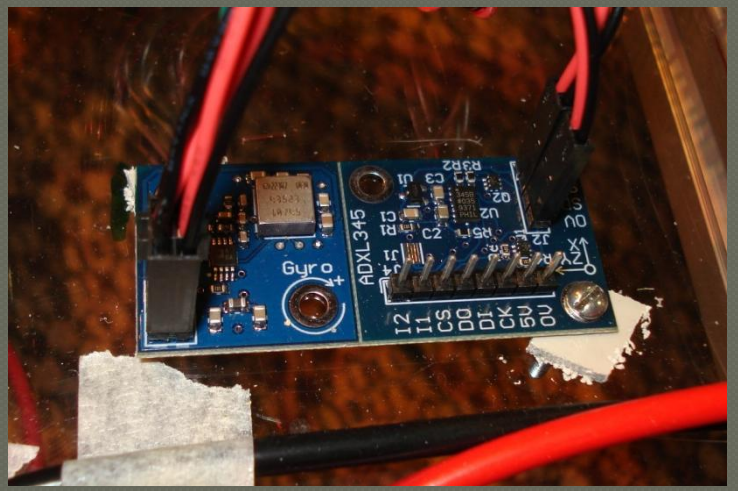
Pneumatic Cylinders

- Created for convenience
- Three possible two-way solenoid variations
- Five main methods
  - Extend()
  - Retract()
  - IsExtended()
  - IsRetracted()
  - Update()



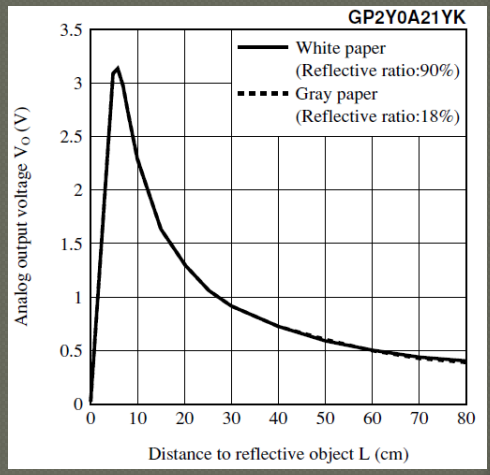
Sensors

- Encoders (4 wheels, 1 ladder spool)
- Optical IR line sensors (3)
- IR Distance sensors (4)
- Gyroscope (1)
- Pressure switch (1)



Sensors

- Wheel encoders
  - Measure relative position
    Update position estimate in Navigator class
  - Discrete derivatives yield wheel speeds
    Closing independent velocity loops for each wheel
- Gyroscope
  - Measures angular velocity
    Updates position estimate in Navigator class
- IR Distance sensors
  - Measure distance
    Update position estimate in certain tasks



Sensors

- Optical IR line sensors
  - Provide TRUE/FALSE for "I see the line"
    - Update position estimate in TaskFollowLine class
- Pressure switch
  - Provides TRUE/FALSE for "Pressure is OK"

  Used by libWPI Compressor class for bang-bang control of air pressure

Quadrature Encoders

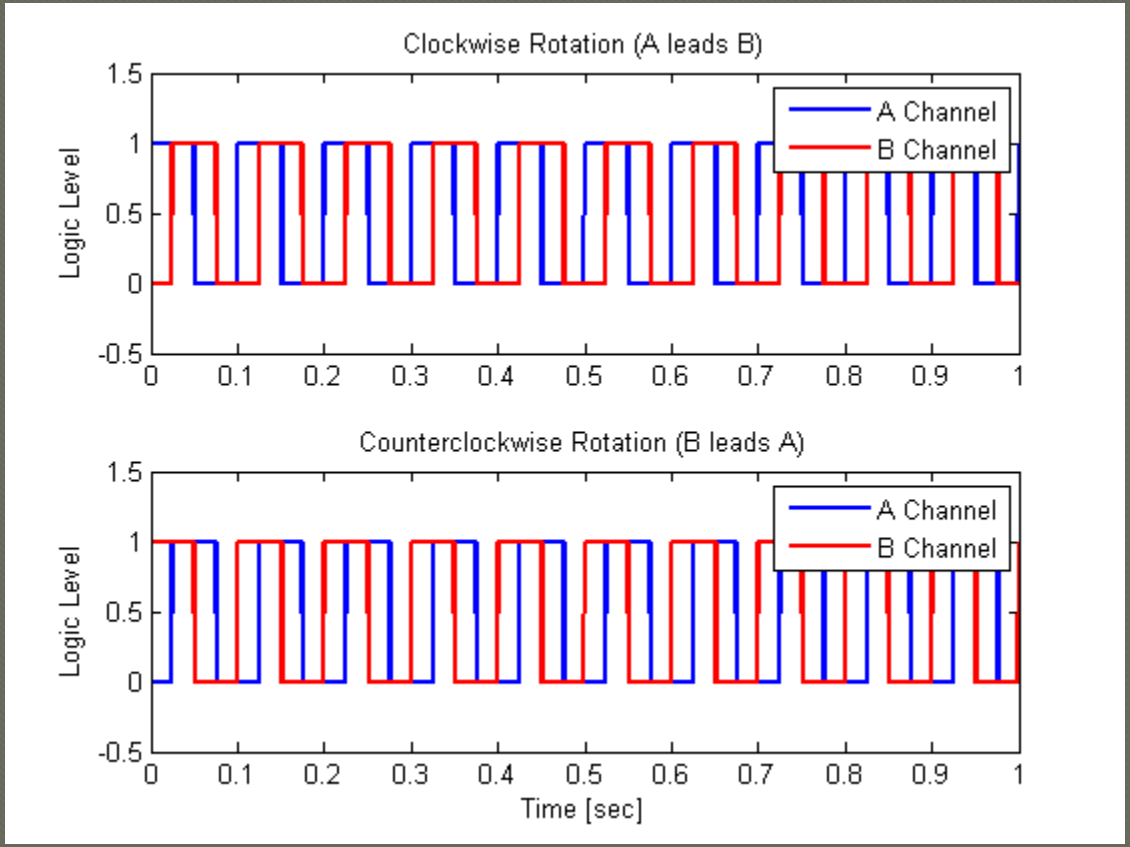
- Pulses or counts per revolution
  - Incremental (relative position) with direction
- Naïve solution:

```
if (lastA == 0 && currentA == 1)// Channel A rising edge
{
    // Check value of channel B to determine direction
    if (currentB == 0)
        position += pulseDistance;
    else
        position -= pulseDistance;
}
```

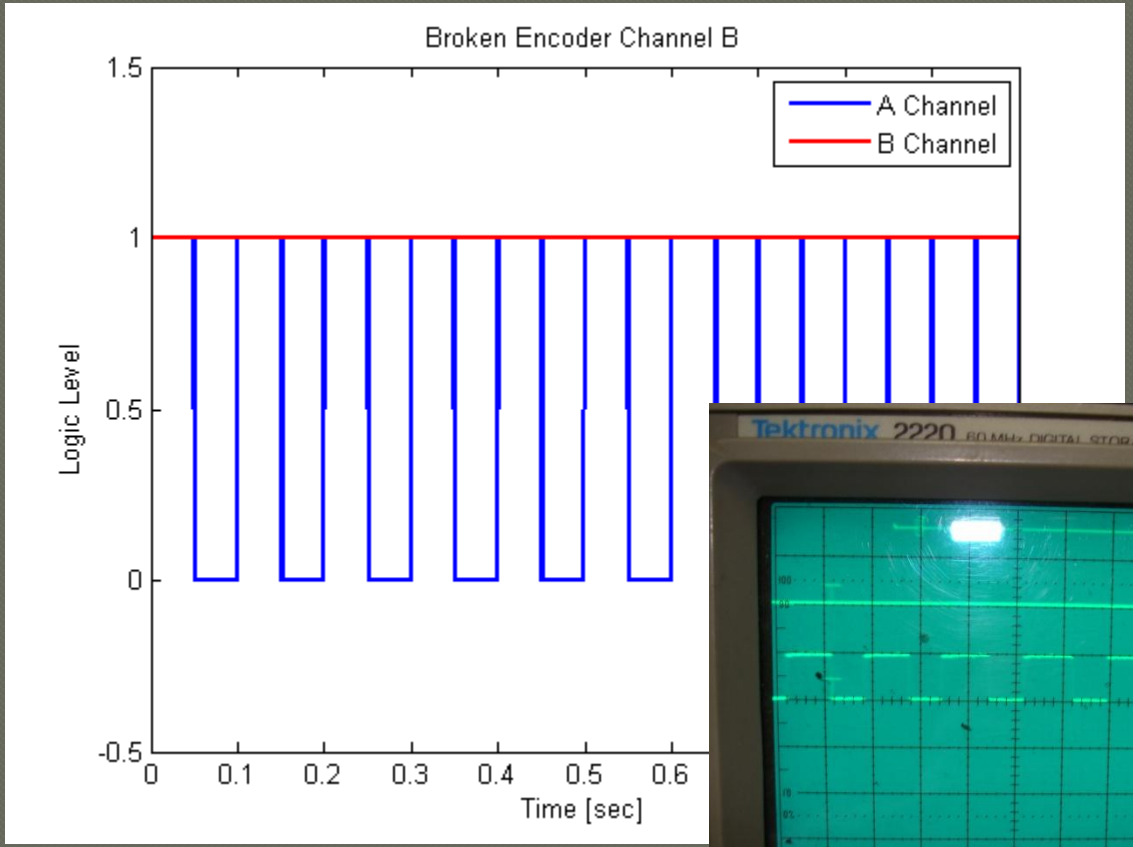
Quadrature Encoders

- Better: State machine
- Look for transitions:
  - State [A high, B low] to [A high, B high]
    +1/4 Increment (clockwise)
  - State [A high, B low] to [A low, B low]
    -1/4 Increment (counter clockwise)
  - State [A high, B low] to [A low, B high]
    - Something is wrong!

Quadrature Encoders



Quadrature Encoder Failures



Code Maintenance and Testing

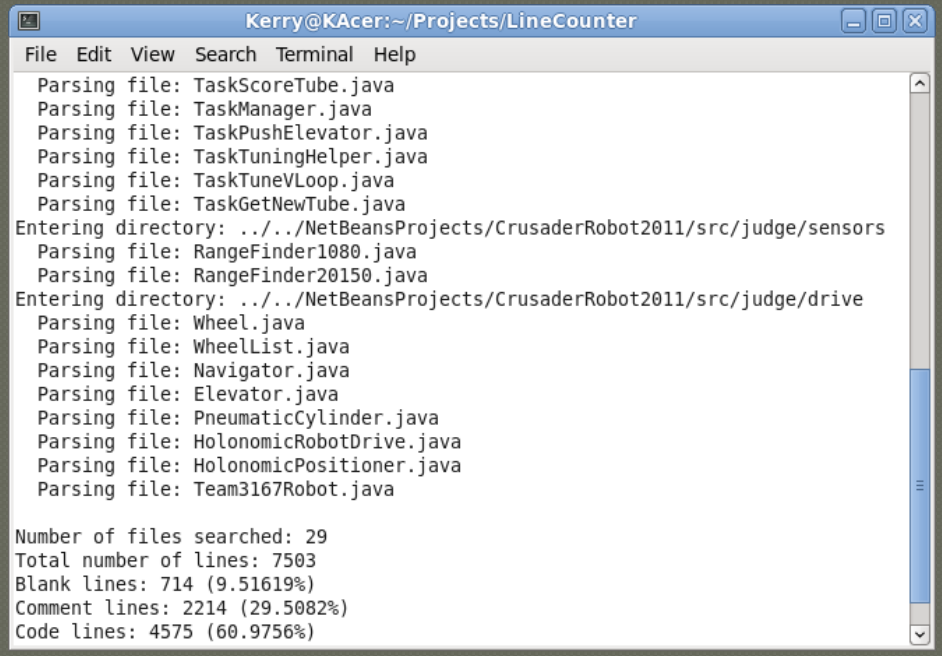
- Unit tests
- Javadoc
- (Good) Comments
- Good names for classes, methods, fields and local variables
- useCamelCase
- http://www.squarebox.co.uk/download/javatips.html
- http://www.perlmonks.org/?node_id=727817

Debugging and Maintaining Code

- Helpful practices:
  - Clear, readable code
  - Good variable names
    - No "magic numbers"
  - Neat, consistent whitespace
  - Detailed comments
    - Coordinate systems
    - Dimensions/units
    - Math derivation
    - Things that don't work
  - Detailed error messages

Importance of Maintainable Code

- Scorecard
  - 29 Files
  - 7503 Lines
    - Whitespace: 714 Lines
    - Comments: 2214 Lines
    - Code: 4575 Lines



Questions?