Robotics 311 : How to build robots and make them move

Prof. Elliott Rouse

GSI Yves Nazon MS

Fall 2022

Photo credit: Brenda Ahearn / U-M CoE

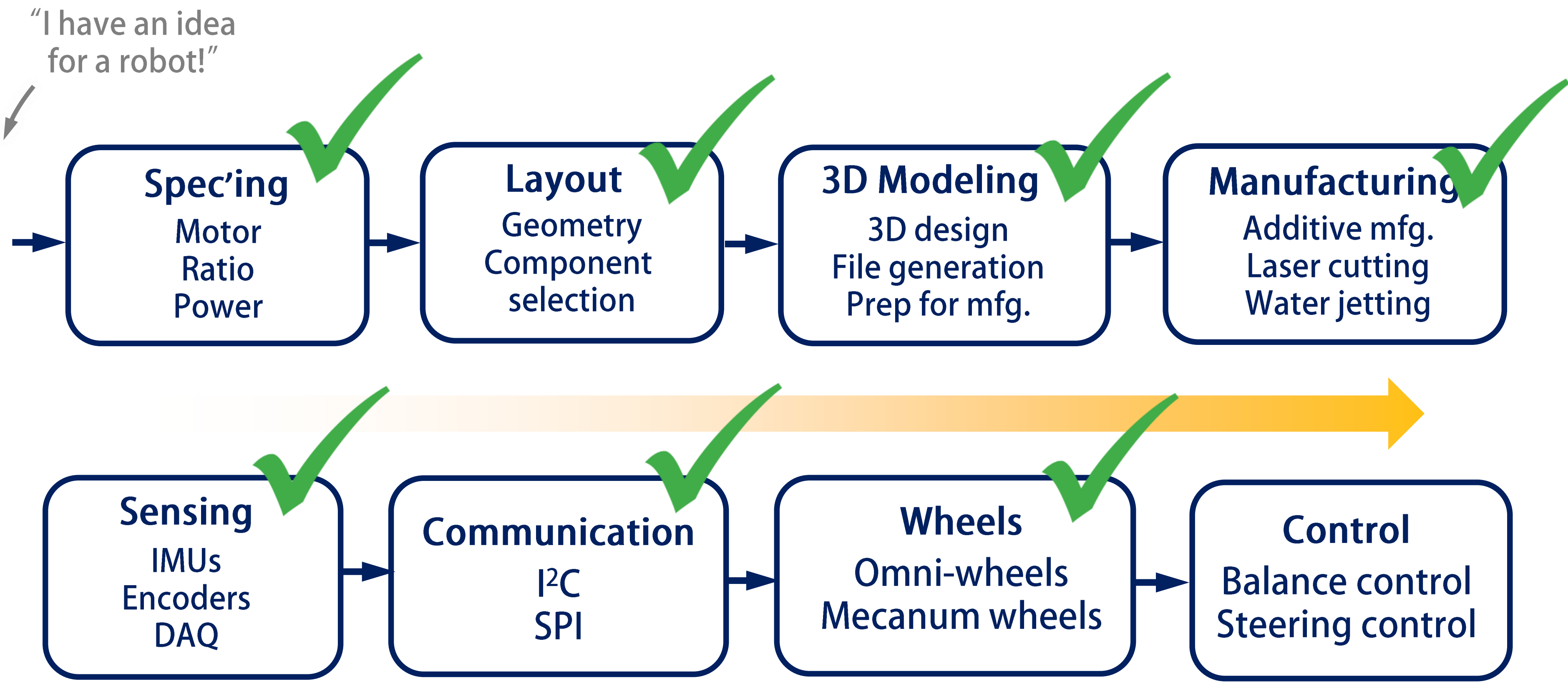
ROB 311 – Lecture 21

- Review PID control
- Understand control loop structure
- Learn PID implementation details

Announcements

- Feedback?
- HW 5 will be posted
- Do you know filtering?

Where We Are



- We've learned much of the spec, design, and make processes
- Now, we need to learn how to make robots do what we want
- At the low level, this involves feedback control

Control

- Today, we will go over the basics of 'low-level' control
- What is the job of low-level control? Track a specific reference signal
- Examples
    - A robot needs to move its arm to follow a trajectory
    - An exoskeleton is providing torque or current assistance
    - Regulate the temperature in a house
    - Cruise control on your car
- The job of a controller? Track the reference / act like unity (so reference = actual)

Reference trajectory →

What's in here?

- *Control law* – math function to describe how effort is added
- *Feedback* – output variables used in control law
- *Actuator* – dynamics of actuation producing system (motor)
- *Process* – dynamics of physical system being controlled

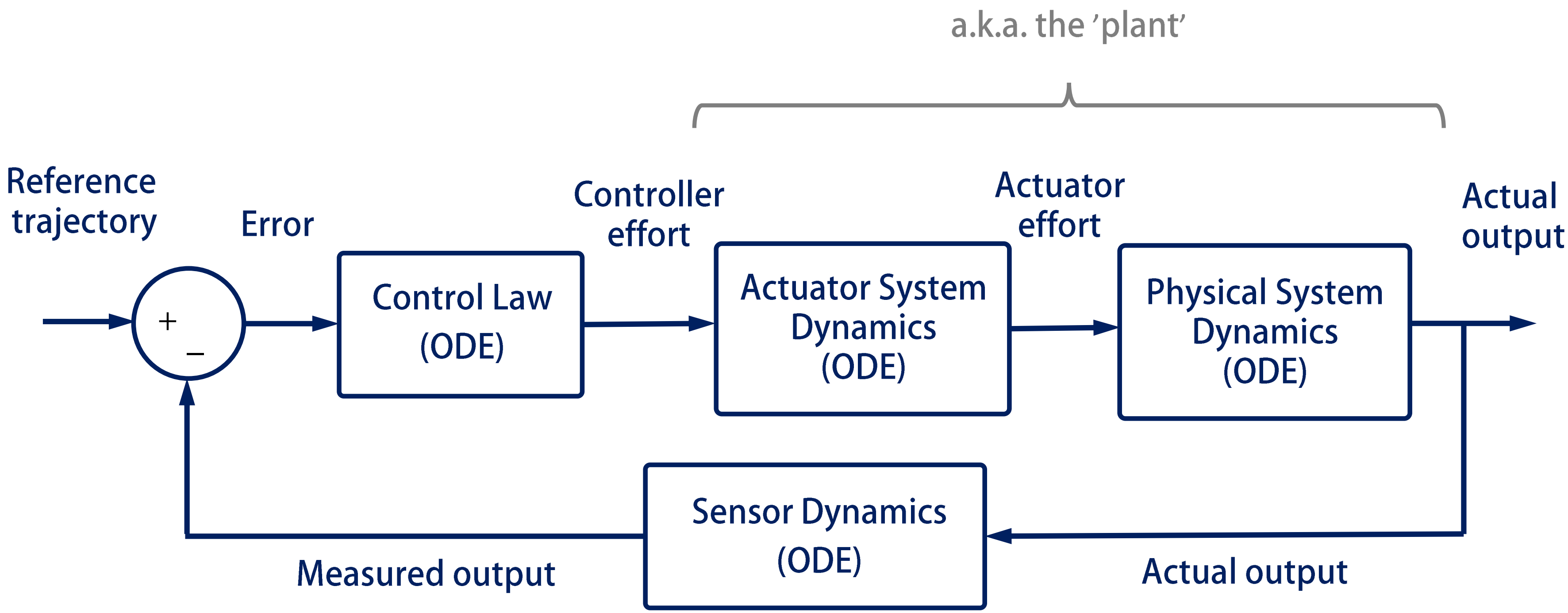
$\cong \times 1$

→ Actual trajectory

Feedback Control Diagram

Human movement example

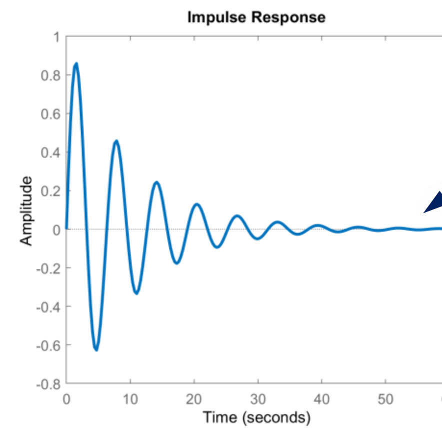
- Why is it hard to make a controller act like unity ($\times 1$)?
- Each one of these blocks is an ODE that maps the block input to the output
- Let's think of an example ODE for a physical system; where would it come from?
- Everyone's favorite: Newton's Second Law
- These systems may have 'dynamics,' which implies a derivative in their equation

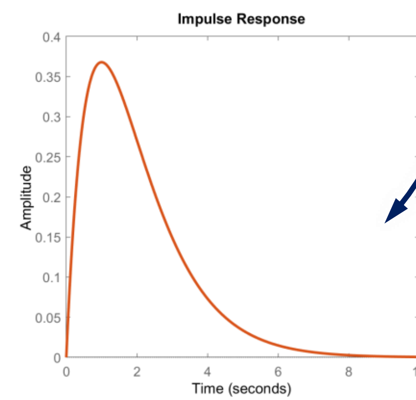


System Dynamics

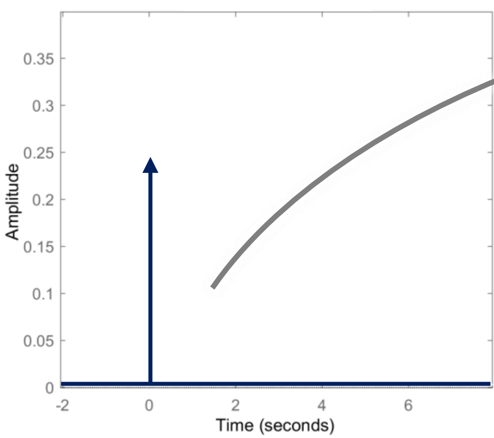
- If a system has dynamics / is an ODE, it will often include delays
- This usually slows the system down with potential oscillations
- We can view this behavior from the system's impulse response (IRF)
- This is the system response to a pure impulse
- Systems are fully described by their IRF
- Lets think about an example 2nd order system

Potential outputs (depending on exact m, k, and c)

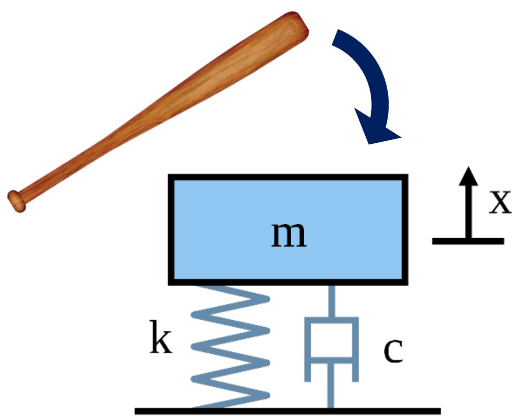




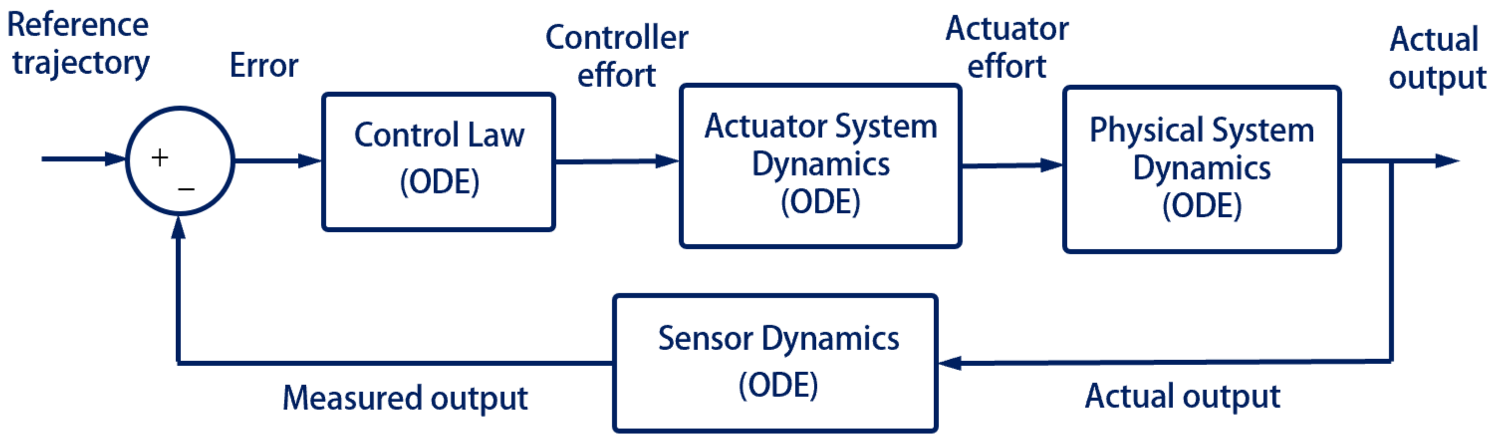
Force input (impulse function)



Delta function gets smoothed / delayed

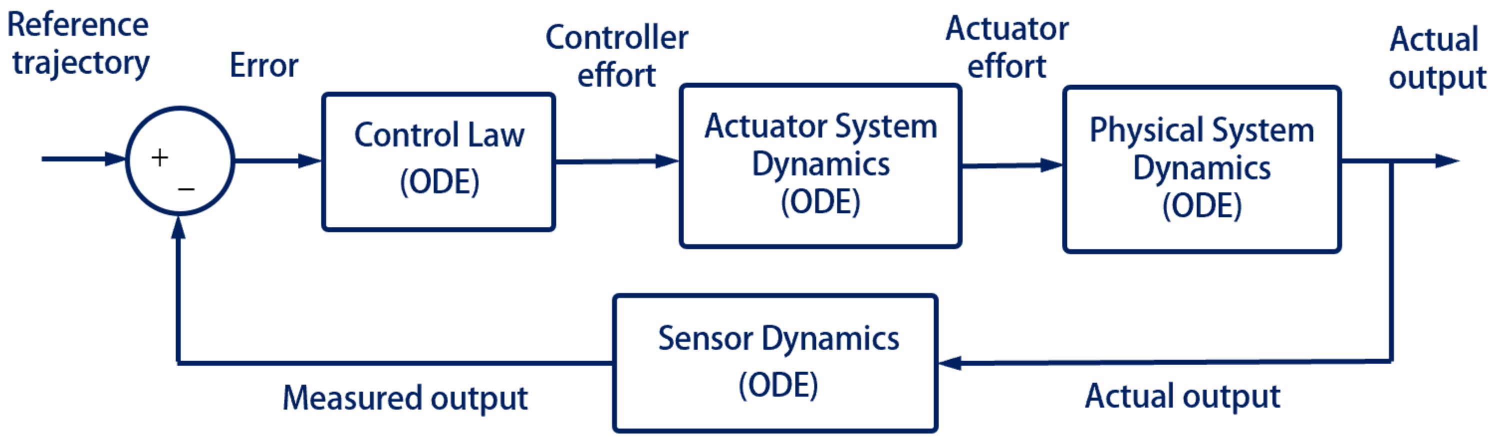


System Dynamics



- These dynamics / delays add challenge to tracking the reference
- Control Law – ODE comes from math instructions are provided, which often include derivatives
- Actuator – ODE that describes how the controller effort maps to actuator effort An example of this is the coupled electromechanical equations of a motor
- Physical system – ODE that describes how the physical system responds to the controller effort
- Sensor – ODE that describes how the output is measured. Most sensors have very little lag (high bandwidth)

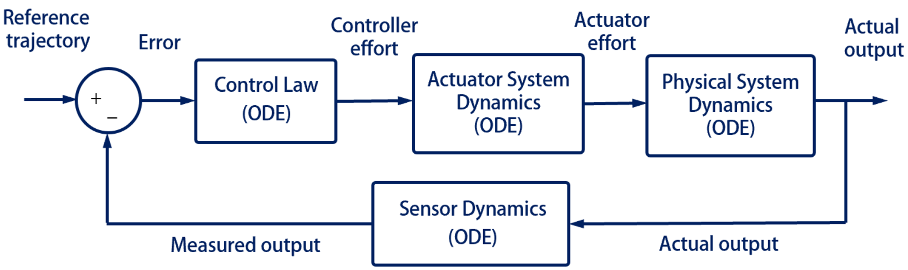
PID Control



- Objective: drive error to zero → if error is zero, system gain is unity
- Control laws are functions of the error signal $(e(t))$ – the difference between the reference and measured output
- We talked about proportional control—where the controller effort was just a gain on the error
- Most common control type: Proportional-Integral-Derivative (PID) control
- Controller effort is the sum of three terms
- Lets look at how the terms are created

PID Control

Controller effort: $u(t)$
Error: $e(t)$
Input: $r(t)$
Output: $y(t)$



- Proportional gain (k_p)
  - Simplest term – error scaled by k_p
  - Acts like a virtual spring (for position control)

$$u_p(t) = k_p e(t)$$

- Integral gain (k_i)
  - Integral of error scaled by k_i
  - Used to remove steady state error
  - Whatever is below reference must be subtracted by what is above reference

$$u_i(t) = k_i \int_{t_0}^{t} e(\tau)\, d\tau$$

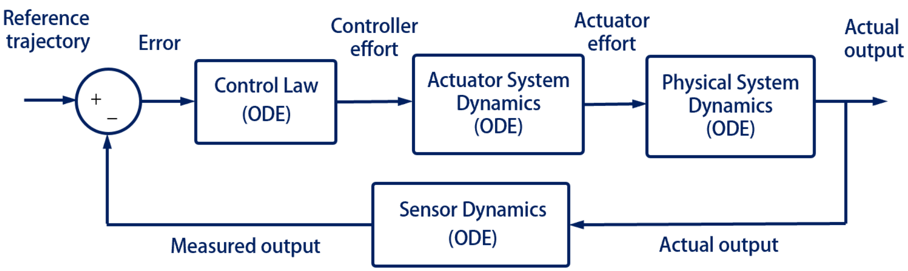
- Derivative gain (k_d)
  - Derivative of error scaled by k_d
  - Virtual damper
  - Never used alone / susceptible to noise

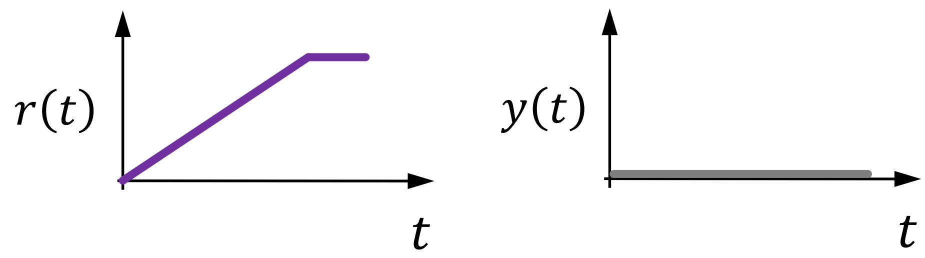
$$u_d(t) = k_d \frac{de(t)}{dt}$$

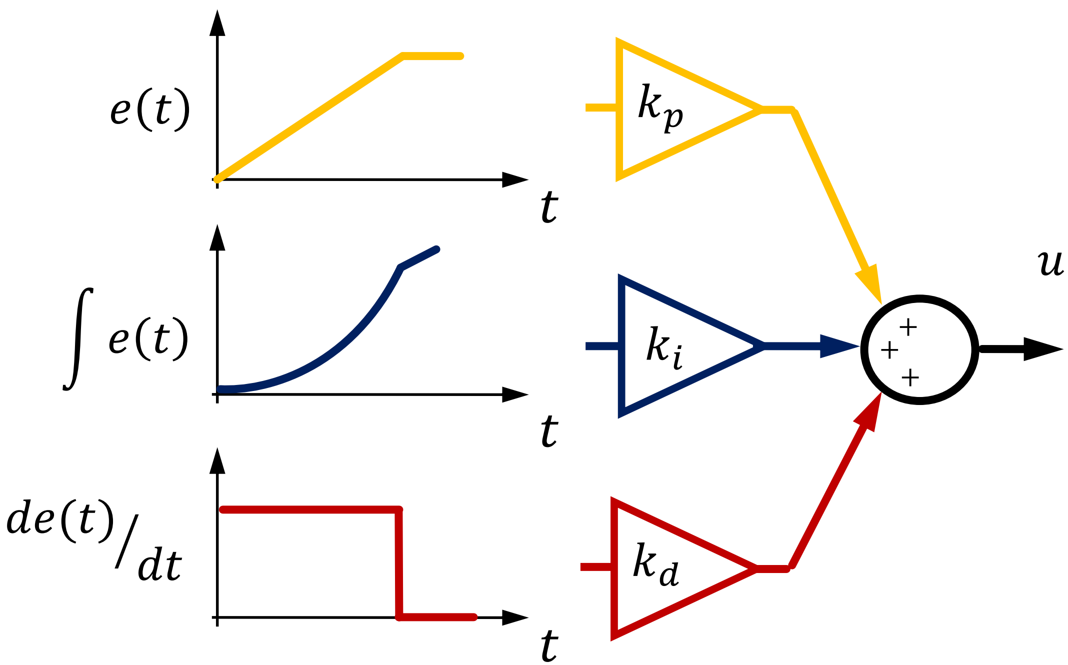
Controller effort: $u(t)$ is the sum of these terms

PID Control

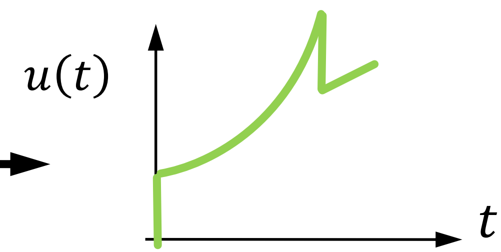
Controller effort: $u(t)$
Error: $e(t)$
Input: $r(t)$
Output: $y(t)$





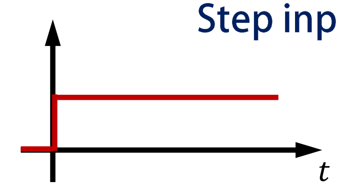


Combined controller output
$(k_p = k_i = k_d = 1)$

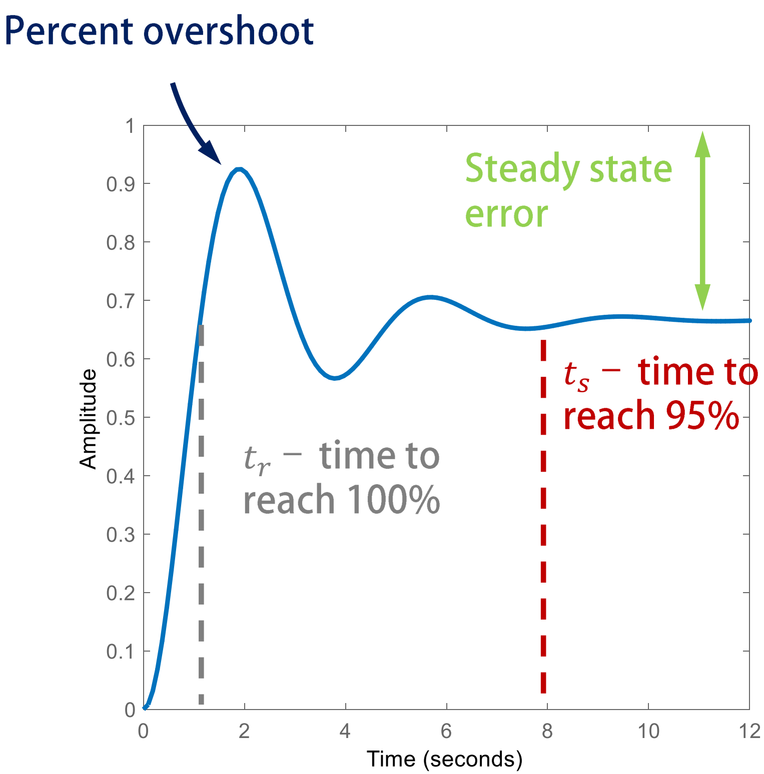


PID Control

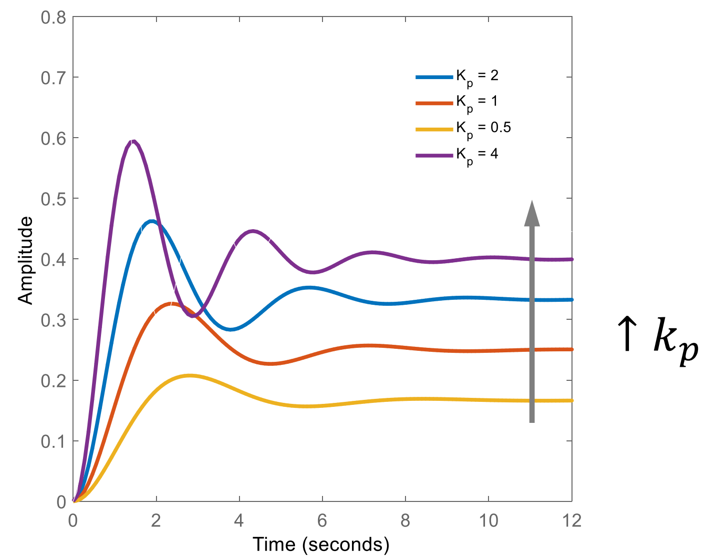
Step input



- Coefficients must be tuned, often using a step response
- Done by iteratively by trying different controller parameters – ‘tuning’
- Assessed often with time domain parameters
  - Rise time, settling time, percent overshoot, steady state error

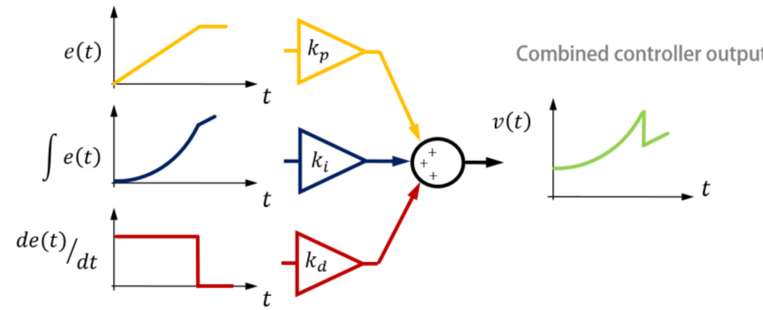


$$u(t) = k_p e(t) + k_i \int_{t_0}^{t} e(\tau)\, d\tau + k_d \frac{de(t)}{dt}$$



PID Control

- The coefficients in the control law can be adjusted to achieve certain performance
- Ziegler-Nichols has developed different tuning strategies (more info here)
- ① Increase k_p as high as comfortable
- ② Reduce k_p and add k_i to remove steady state error
- ③ Add k_d to remove oscillations
- By driving error to zero, the controller behaves like unity

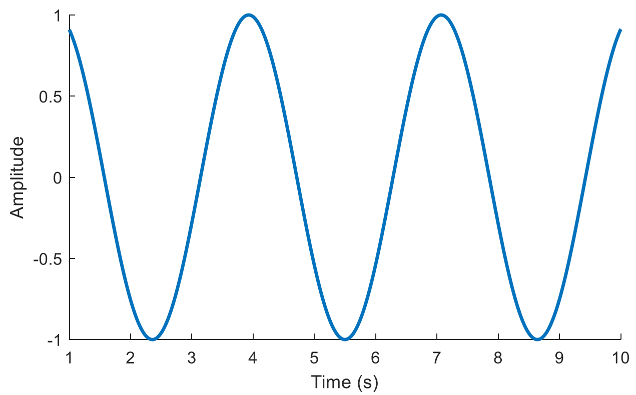


$$u(t) = k_p e(t) + k_i \int_{t_0}^{t} e(\tau)\, d\tau + k_d \frac{de(t)}{dt}$$

Parameter Increase	Rise time	Overshoot	Settling Time	S.S. Error
k_p	↓	↑	Small ↑	↓
k_i	↓	↑	↑	↓↓
k_d	Small change	↓	↓	Small change

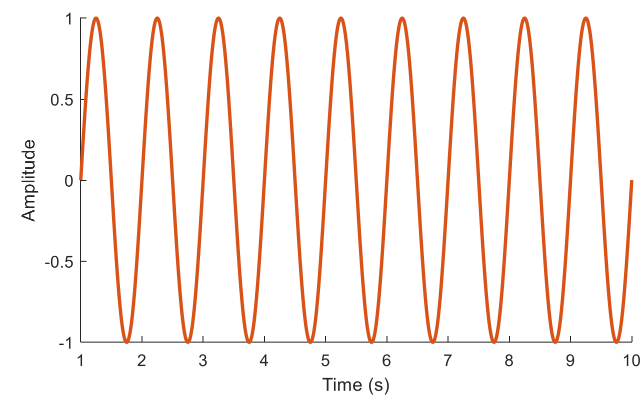
Reference Trajectories

- The trajectory of the reference has a significant impact on controller performance
- Oscillating references are harder to track
- The faster the frequency the harder it will be for the controller



Easier to track

Harder to track

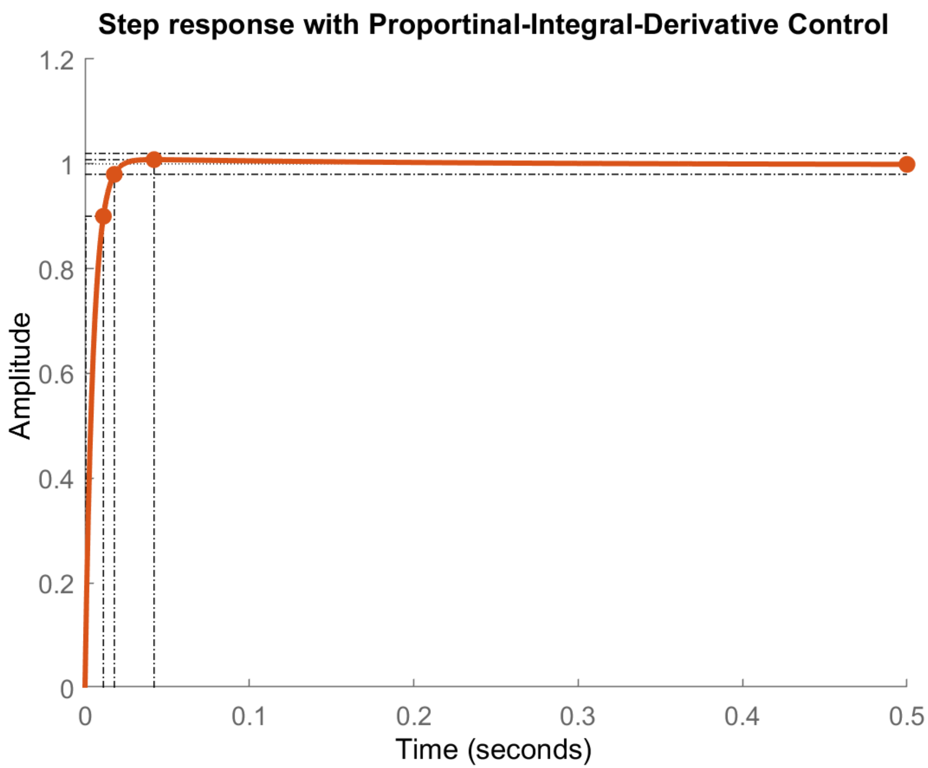


- Some references are unchanging (static) – these systems are called regulators (e.g. thermostat)
- What does our reference look like? Is it high frequency?
- The output should closely match the reference, but at high frequencies, the amplitude will begin to attenuate and the output sine wave will be delayed

PID Control

- MATLAB example
- Tuning PID gains

```
% PID control
kp = 5000;
ki = 100;
kd = 1000;
```

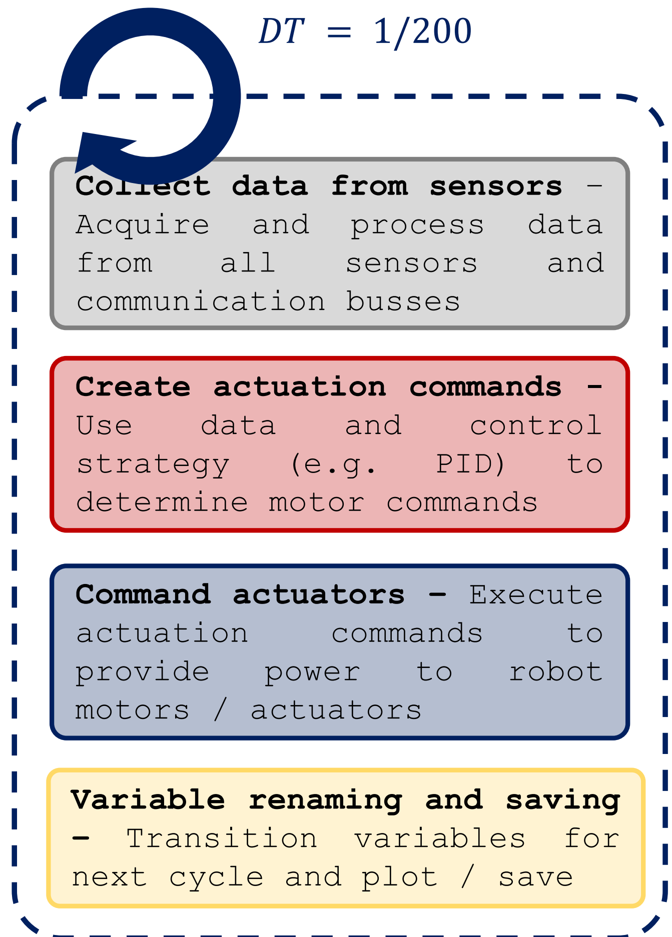


How To Implement Control

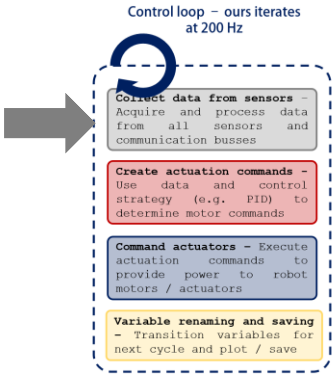
- How can we implement PID in code / best practices?
- Points to consider
  - How do we calculate the derivative?
  - How can we deal with nonlinearities (e.g. saturation)?
- Let's discuss the overall control loop structure
- Your control loop needs to do four things
  - Refresh data / communication
  - Use feedback to determine commands
  - Send commands to the motors
  - Save data and transition variables

Control loop – ours iterates at 200 Hz

$DT = 1/200$



Data Collection



- In our control loop, the first thing that must be done is to refresh data from all sensors
- This would be a series of communication library calls, typically one for each component
- In our system, your data are refreshed automatically
  - How? The Pico collects and sends data to the RPi
  - We have streamlined this process, but you could do it on your own

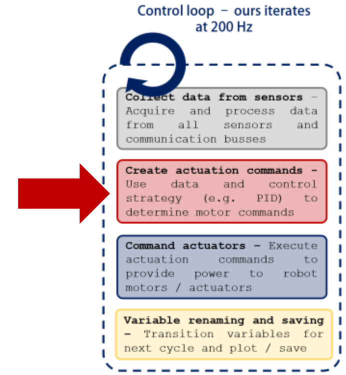
```
# Define variables for saving / analysis here - below you can create variables from the available states in message_defs.py

# Motor rotations
psi_1 = states['psi_1']
psi_2 = states['psi_2']
psi_3 = states['psi_3']

# Body lean angles
theta_x = (states['theta_roll'])
theta_y = (states['theta_pitch'])

# Controller error terms
error_x = desired_theta_x - theta_x
error_y = desired_theta_y - theta_y
```

Actuation Commands



- Setting actuation commands comes from our control law
- We will use PID
- Let's go through the calculation of each term
  - Proportional term

Easy to implement

$$u_p[k] = K_p \cdot (y[k] - r[k]) = K_p e[k]$$

  - Most of your controller effort will likely come from this term
  - Integral term

This is your integral—it can be a running sum

$$e_{sum} = e[k] + e_{sum}$$

The DT can be left out, which just scales K_i

$$u_i(t) = K_i \cdot e_{sum} \cdot DT = K_i \int_{t_0}^{t} e(\tau)\, d\tau$$

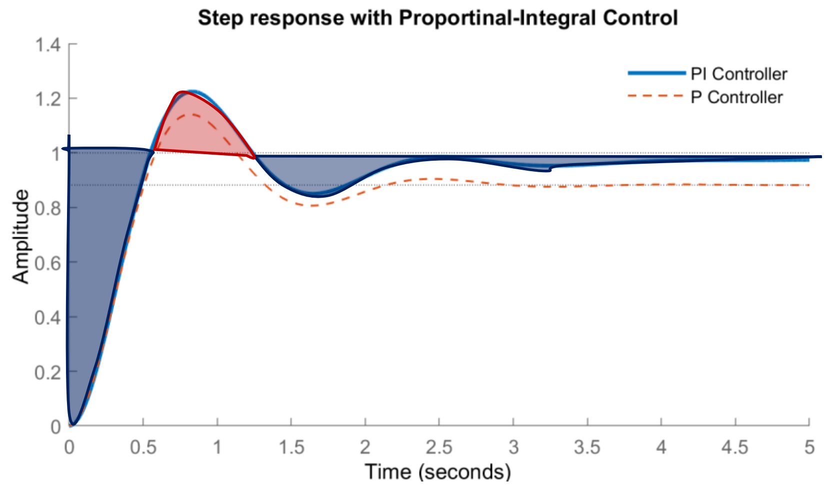
  - The integral term is susceptible to saturation
  - Saturation is a type of nonlinearity

$[k]$ used to mean the value of loop iteration k

M ROBOTICS

Integral Commands

- Integration term – saturation
- Reminder – the integral effort will keep track of the difference between the reference and the output



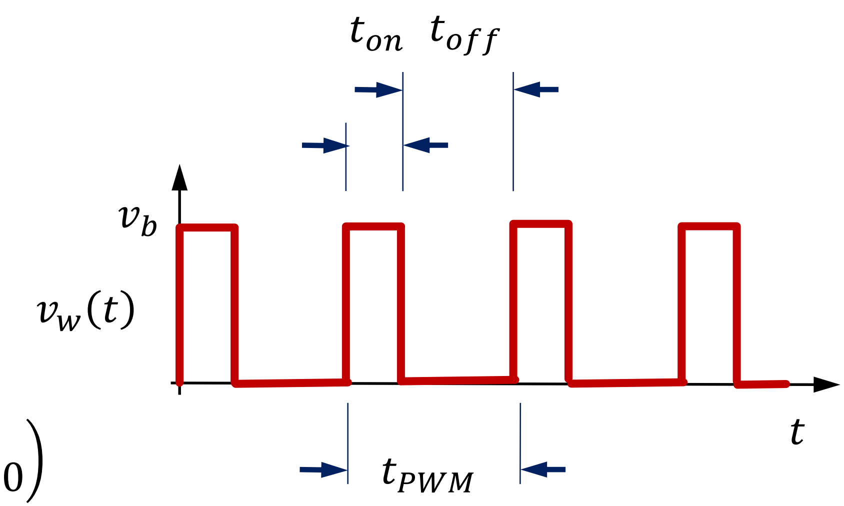
- This idea allows the controller to add as much effort as possible
- Steps could be arbitrarily large in your application—are can be very large
- Can your controller always provide this effort? Think about flooring your gas pedal...

PWM

Battery voltage: v_b

Applied motor voltage: v_w

- Pulse Width Modulation (PWM)
- This is how our actuation command (voltage) is provided to the motors
- A quickly-oscillating voltage that varies between 0 V and the battery voltage
- By varying the 'duty cycle' the applied voltage can be varied
- The dynamics of the motor and physical system smooth the rapidly oscillating voltage
- What's the max PWM value? 100%
- This creates a nonlinearity when the controller maxes out



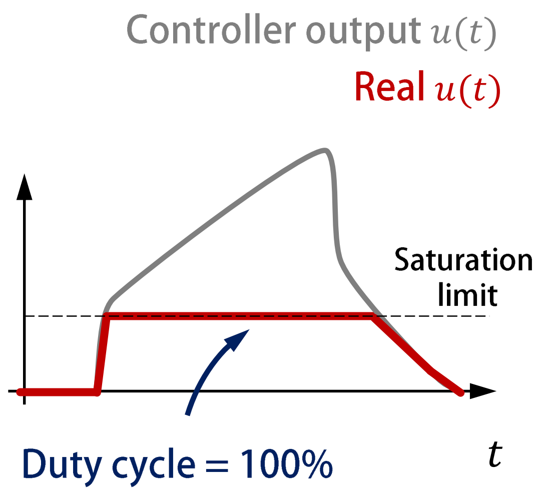
$$F_{PWM} = \frac{1}{t_{PWM}}$$

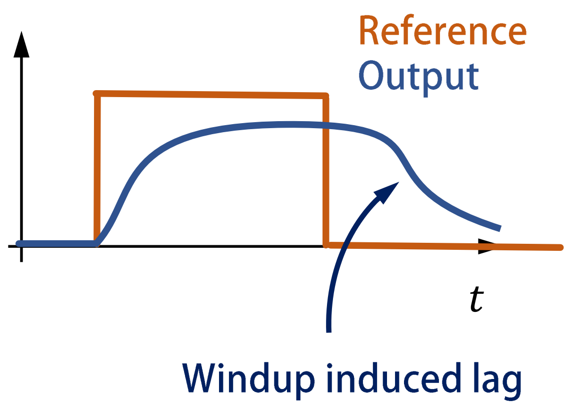
$$Duty\ Cycle = \frac{t_{on}}{t_{PWM}} \cdot 100$$

$$v_{applied} = v_b \cdot \left({}^{Duty\ Cycle}/_{100} \right)$$

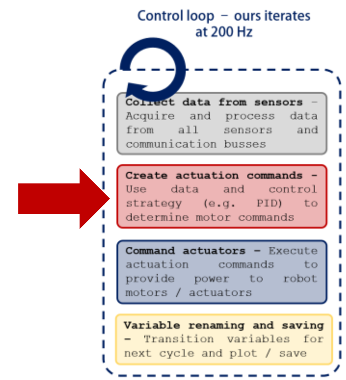
Saturation and Windup

- The maximum-effort nature of actuators is known as ‘saturation’
- Saturation can cause excessive overshooting / inefficiency
- To address saturation, sometimes a saturation value is used to limit $u_i(t)$
- To implement in Python, an `if-then` statement can be used to check the magnitude of $u_i[k]$
- You can limit $u_i[k]$ to a maximum of $X\%$ of the maximum effort
  - 50% could be a good starting point, but it will need to be adjusted
- Integral terms add a delay or lag
- Known as ‘windup’





Derivative Commands



- We've so far described the nuances of calculating P and I commands
- The derivative term requires a numerical derivative
- Most common is the two-point / finite difference method

This is the value from one loop iteration go

Finite difference derivative

$$\frac{de}{dk}[k] = \left(e[k] - e[k-1]\right)/DT$$

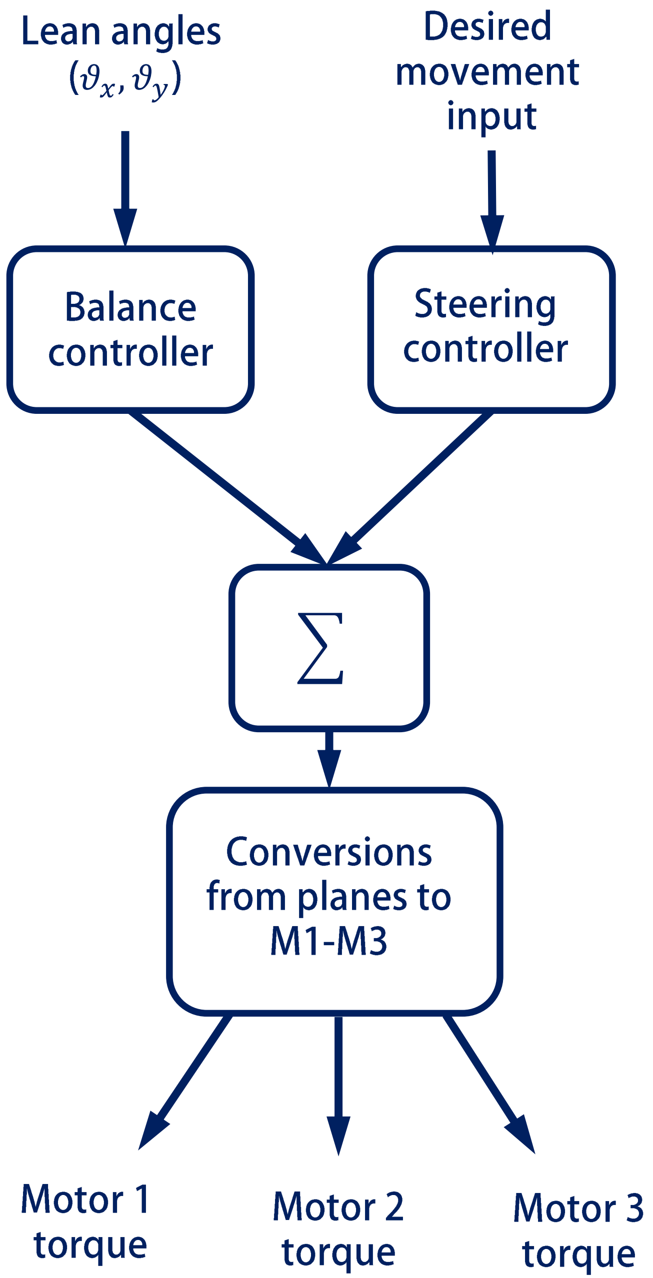
Variable saved / transitioned at the end of each loop

$$u_d[k] = K_d \cdot \frac{de}{dk}[k]$$

- Watch out for noise! Clean signals are needed to gain useful information
- Are our signals noisy?
  - IMU – no it's actually pretty clean - view the data to confirm
  - Encoder data – clean when viewed at larger time scales, but quantized by nature
- These data can be filtered, but this adds delay

Controller Architecture

- We break the controller into the two planes
- Each plane will be handled independently
- Each plane has two controllers that run in parallel
    - Balance controller / steering controller
    - They will be separate but will run simultaneously
- There will be four total controllers in parallel
- We will superimpose the torques from the balance and steering controllers
- Simultaneous balance and steering
- We will begin with the **balance controller**
- Let's think about how this controller should be designed

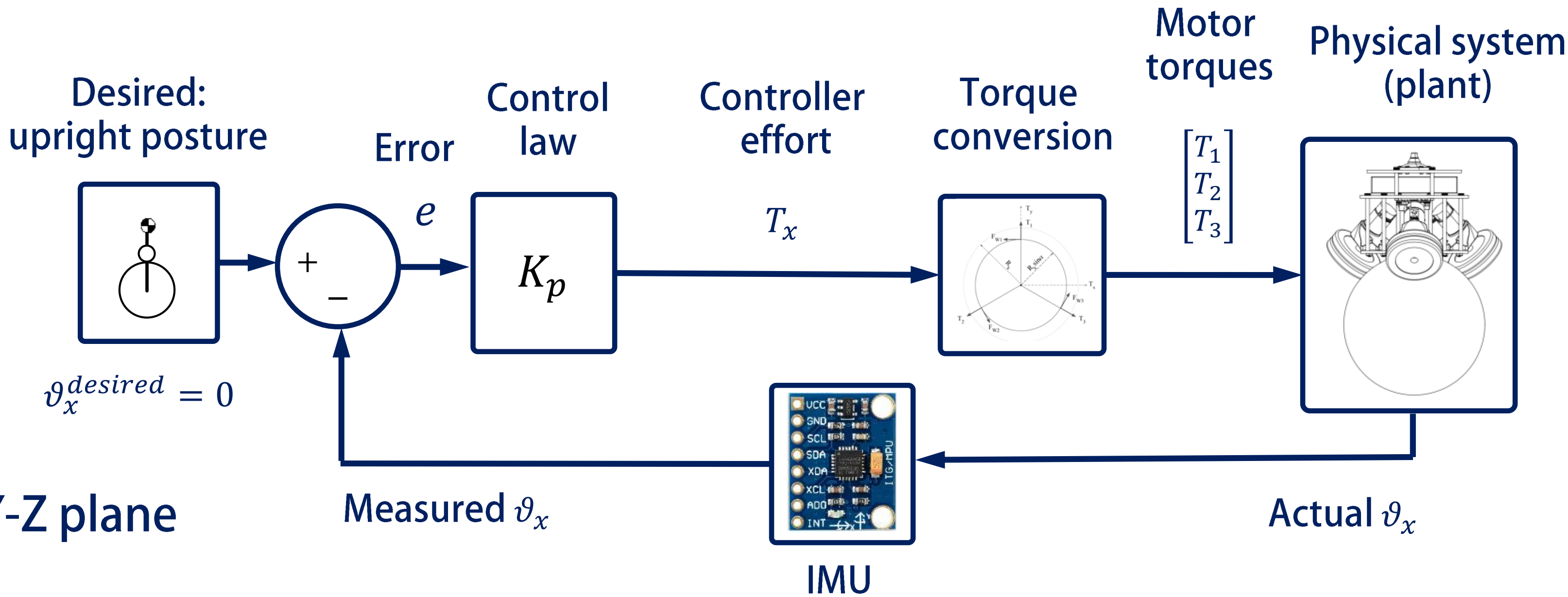


Balance Controller

- In lab, we began with a simple P-controller
- We knew we wanted the system to maintain upright balance

$$T[k] = -K_p \cdot \vartheta_{axis}[k]$$

- Our reference trajectory is upright posture ($\vartheta_x^{desired} = \vartheta_y^{desired} = 0$)
- Putting into the technical framework of feedback control
- Control law: $T[k] = e[k] \cdot K_p = \left(\vartheta^{desired}[k] - \vartheta[k]\right) \cdot K_p = -\vartheta[k] \cdot K_p$



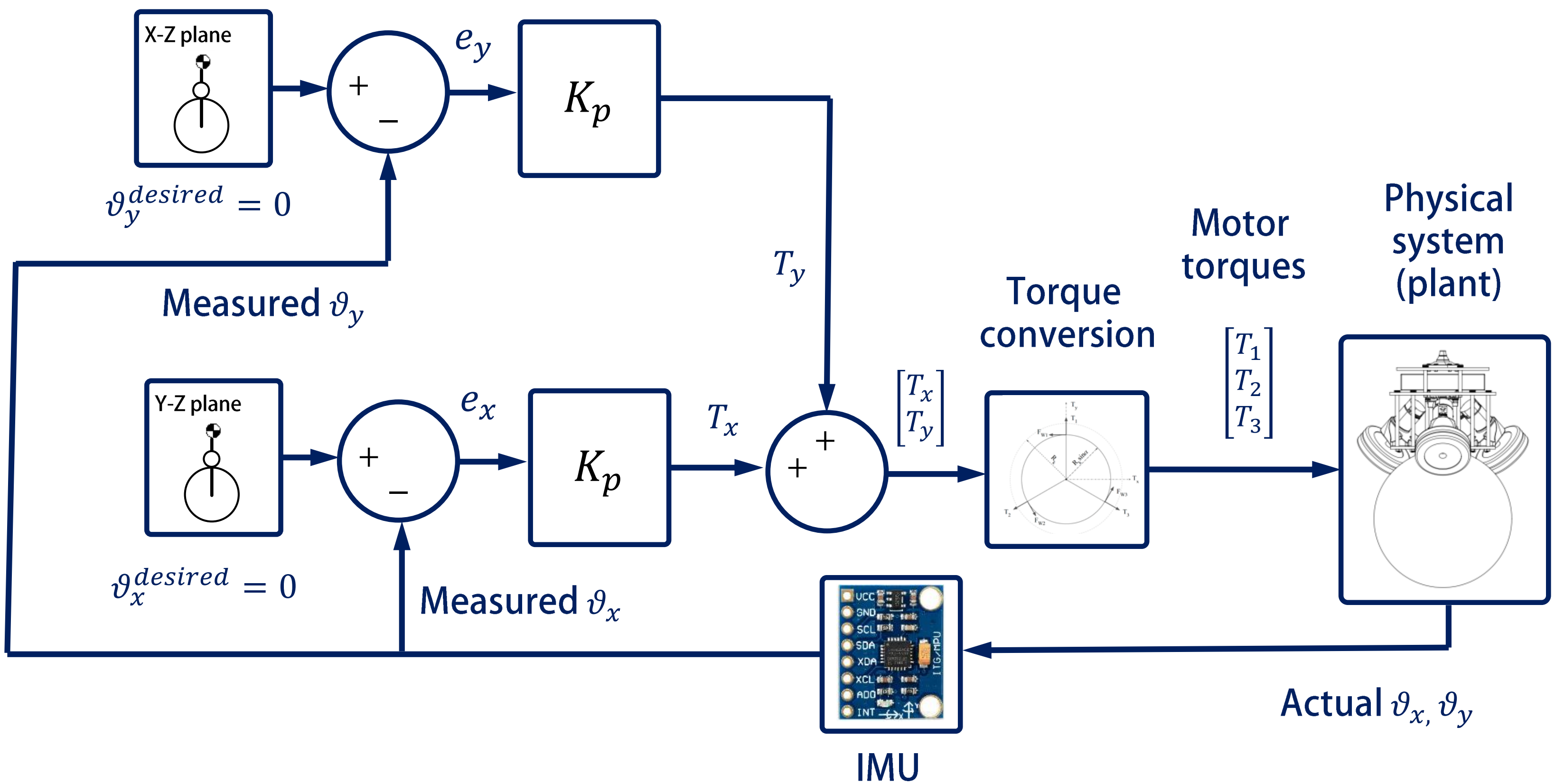
Y-Z plane

Signals get multiplied by blocks

M ROBOTICS

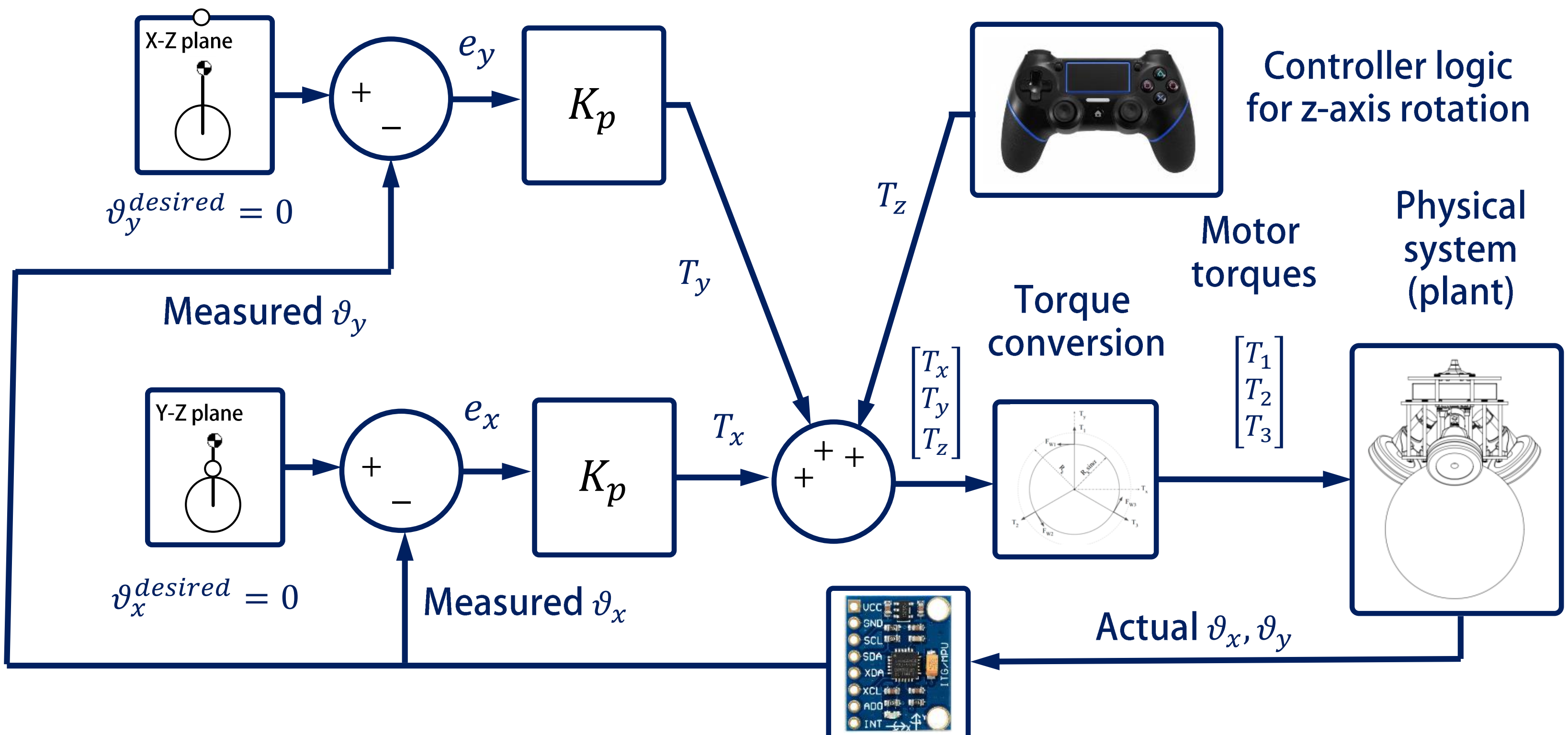
Balance Controller

- We extended to both planes at once (X-Z and Y-Z planes)
- Control law: $T[k] = e[k] \cdot K_p = \left(\vartheta^{desired}[k] - \vartheta[k]\right) \cdot K_p = -\vartheta[k] \cdot K_p$
- We learned to choose K_p by tuning the controller (lab)

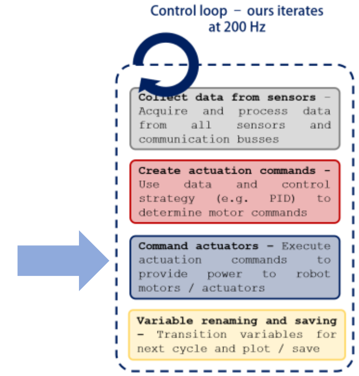


Controller for Z-Axis Torque

- Now we will add the z-axis torque from button commands you choose
- Triggers provide continuous values and buttons provide binary values
- Create a torque function using the button presses and add to the torque commands (or use the demos)
- You will need to set the z-axis torque to the torque value from the PS4 controller



Sending Actuator Commands



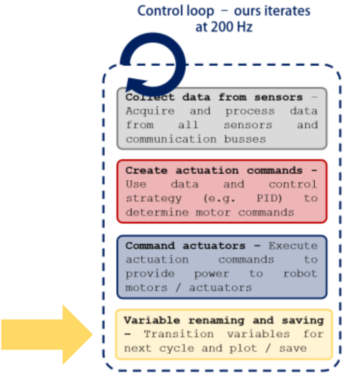
- After constructing the torque commands, they need to be sent to the motors
- We take care of this for you with an API, but you could also do it

```python
# ---------------------------------------------------------

print("Iteration no. {}, T1: {:.2f}, T2: {:.2f}, T3: {:.2f}".format(i, T1, T2, T3))
commands['motor_1_duty'] = T1
commands['motor_2_duty'] = T2
commands['motor_3_duty'] = T3
ser_dev.send_topic_data(101, commands) # Send motor torques
```

- This applies voltage to the motor, the command of which comes from the required torque → required current
- The torque commands are sent to the Pico, which converts them into a motor duty cycle command
- Sending commands is usually only a few lines of code

Saving Data and Transitioning Variables



- At the end of your loop, you will need to
  - Create your data matrix
  - Rename variables (any variables for prev. loop iteration)

```
# Construct the data matrix for saving - you can add more variables by replicating the format below
data = [i] + [t_now] + [theta_x] + [theta_y] + [T1] + [T2] + [T3] + [phi_x] + [phi_y] + [phi_z] + [psi_1] + [psi_2] + [psi_3]
dl.appendData(data)

# Transition variables
error_x_prev = error_x
error_y_prev = error_y
```

- If error values from the previous iteration are used, they need to be transitioned
- This is likely if you're taking a finite difference derivative in the loop
- This sets up the variables for your next loop iteration
- Data matrix gets created and appended to