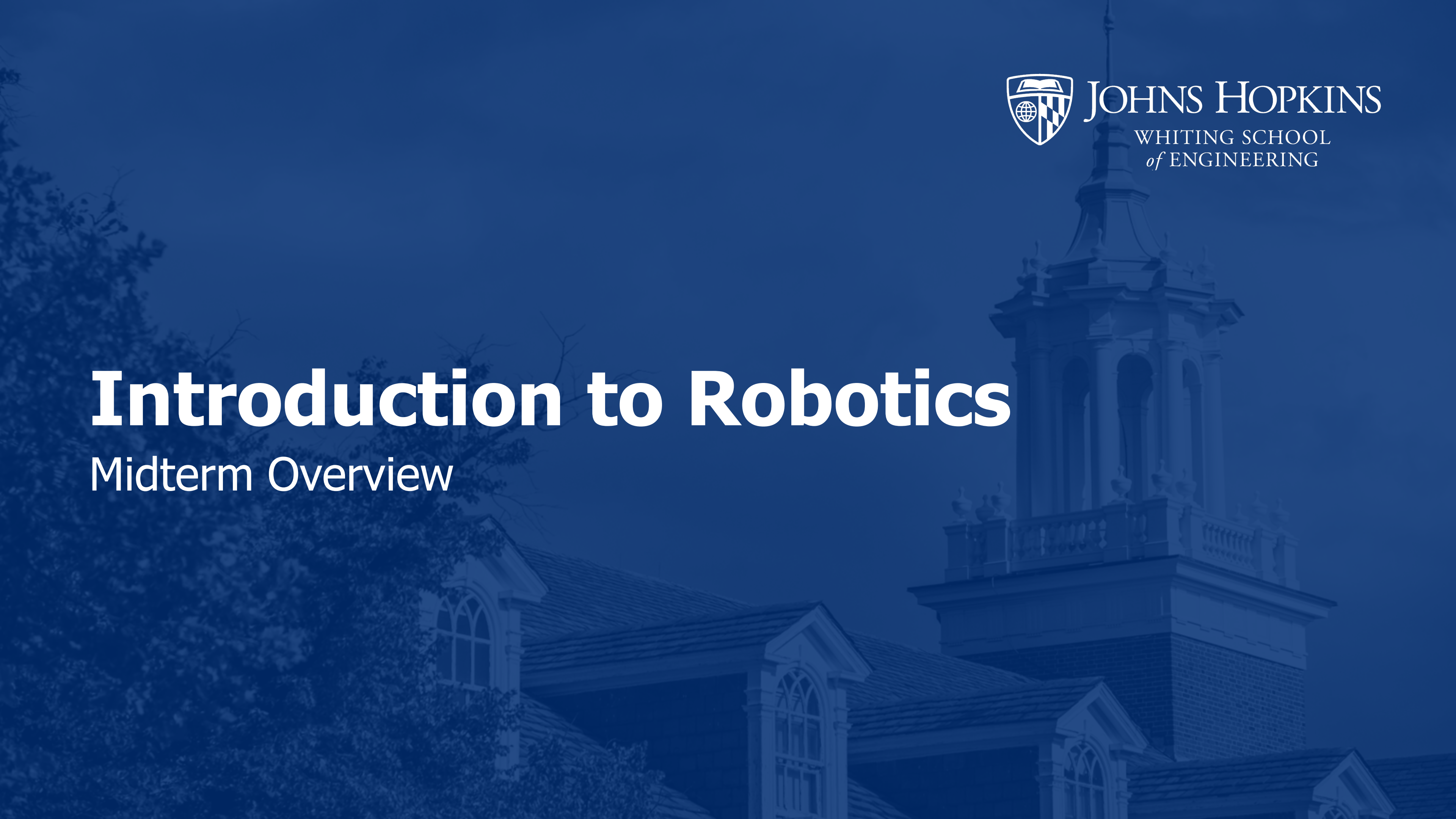
JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Introduction to Robotics

Controllers



JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Introduction to Robotics

Midterm Overview

Midterm Overview

Objective: Control a Differential Drive

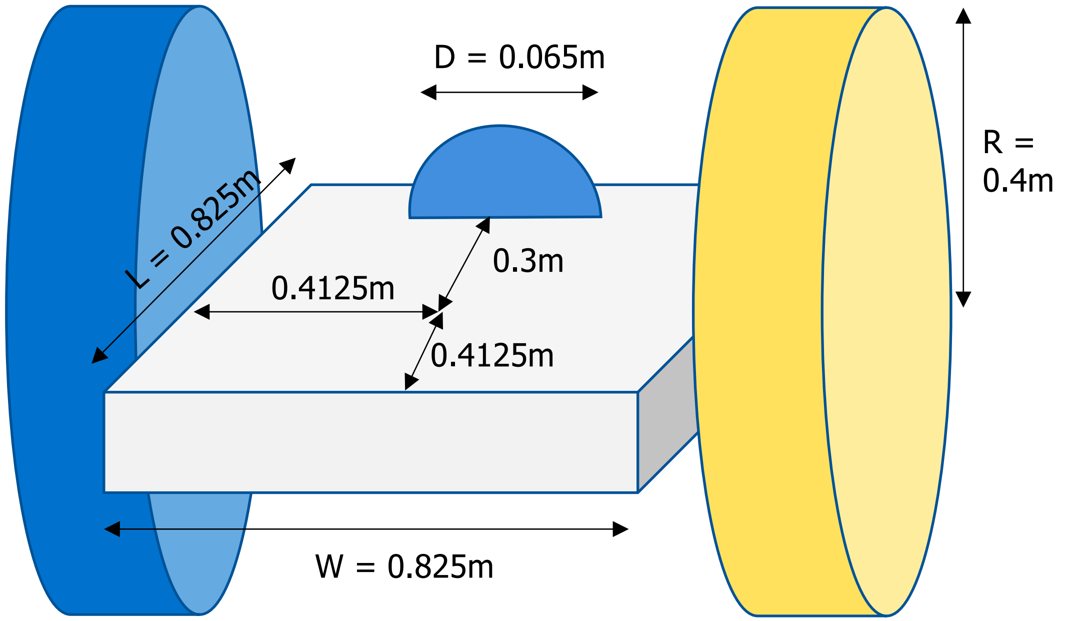
Due: End of Module 6

1. Create a Robot URDF
   - Load via robot_state_publisher
   - Visualize in RVIZ
2. Write a Robot Simulator Node in ROS2
   - Write the forward and inverse kinematics for a differential drive vehicle
   - Publish transforms and joint states
   - Control using teleop_twist_keyboard
3. Write a Robot PID Node in ROS2
   - Subscribe to transforms to get robot state
   - Receive 2D goal commands from RVIZ
   - Publish desired velocity commands

- **Start early**
  - Many problems you encounter may be difficult to debug. Find them early so we can address them during class or office hours

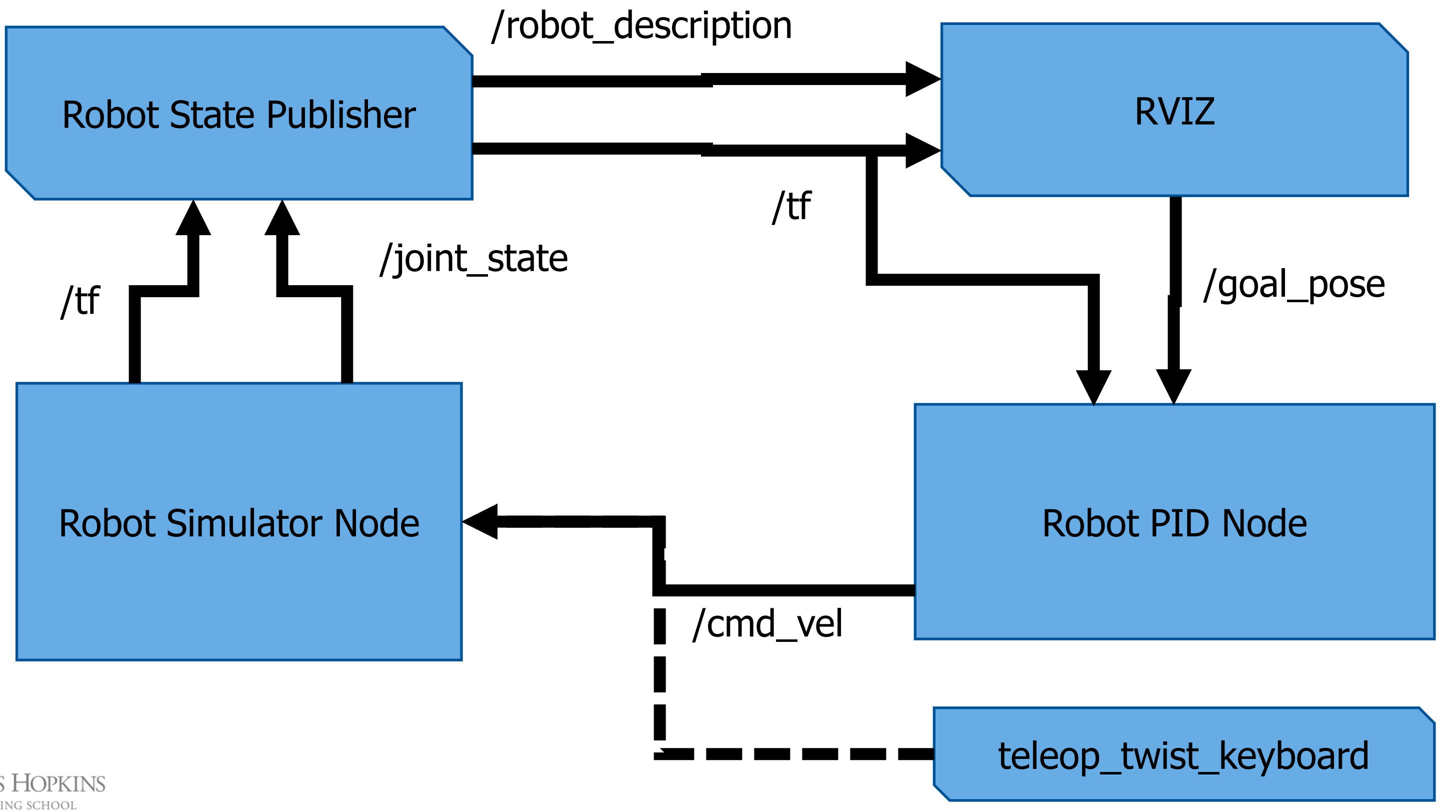
Skills Learned:
- Differential Drive Kinematics
- PID Control
- URDF Files
- Robot state definitions
- Publish and subscribe to TF

Midterm Part 1 – Create a URDF File



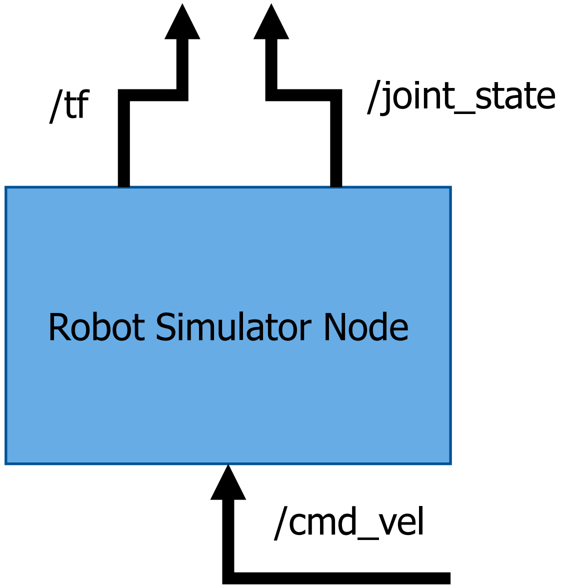
- Two wheeled robot with box body and spherical dome on top near the front
- All the bodies are touching and not intersecting
- X axis is forward for body
- Wheels use Z axis for rotation
- Link names must be
  - chassis
  - left_wheel
  - light_wheel
  - lidar_dome

Midterm Part – ROS Node Architecture



Midterm Part 2 – Robot Simulator Node

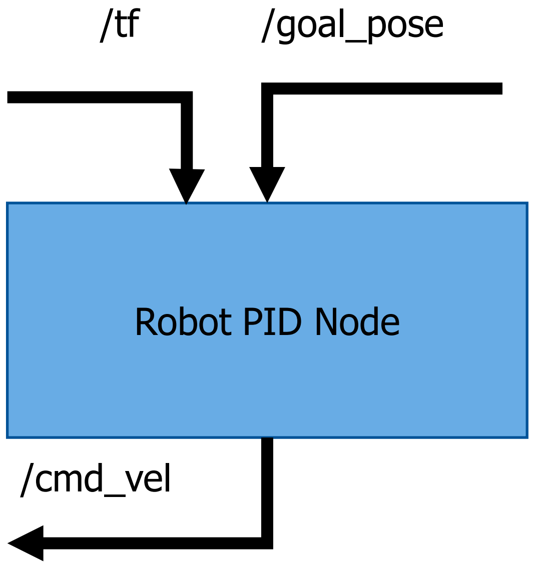
- Implements the Forward and Inverse Kinematics of a Differential Drive Robot
  - Velocities are defined in the robot frame as $[v, \omega]$ the forward speed and the turn rate
- Sets the robot to $[x, y, \theta] = [0, 0, 0]$ at **init**
- Subscribes to **/cmd_vel** and uses the latest value as the desired velocity command
- Runs at 30 hz
  - Computes new wheel velocities
  - Updates the robots state (position and joints)
  - Publishes the transform between 'odom' and 'chassis' frames
- Test using **teleop_twist_keyboard**

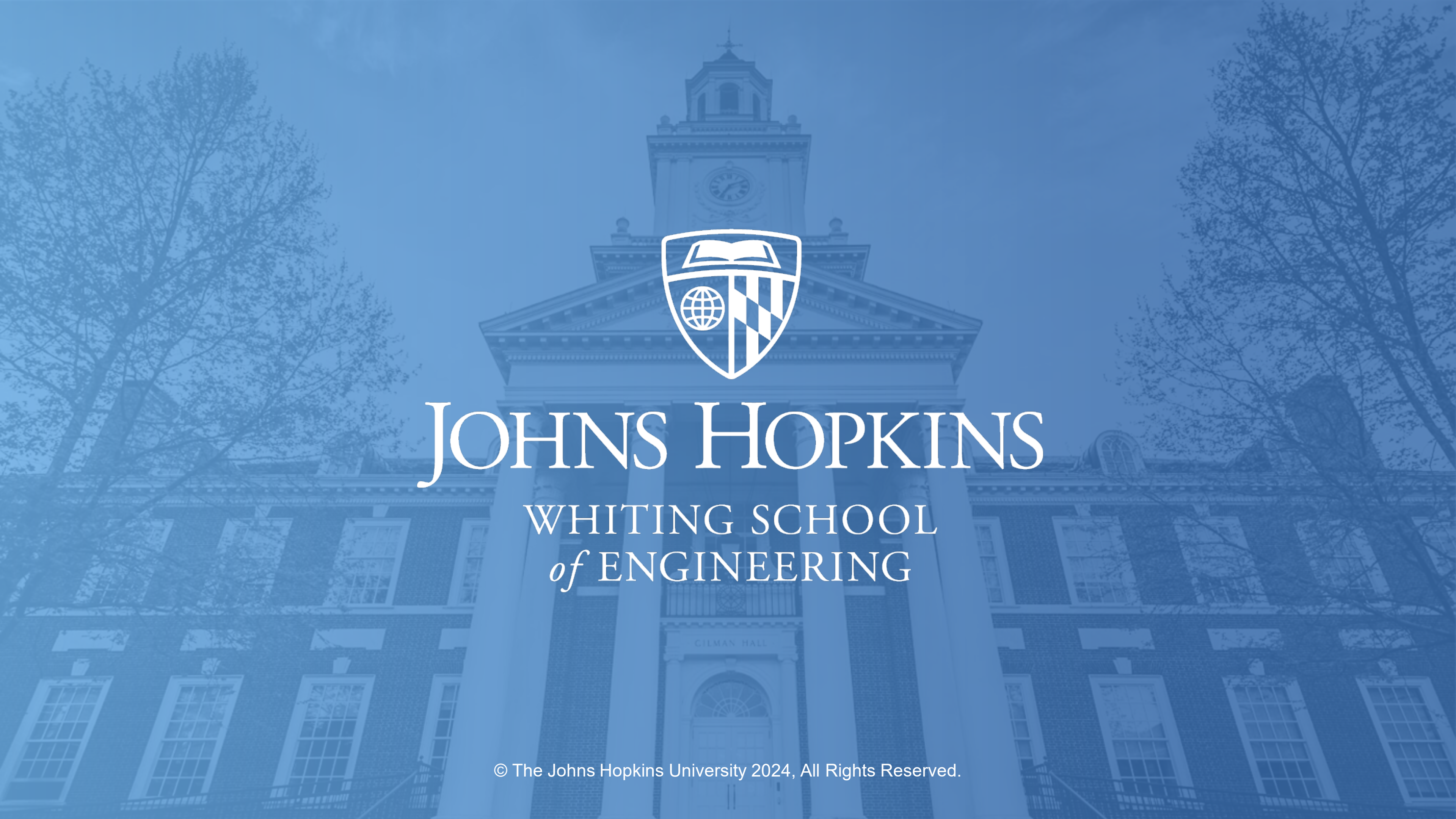


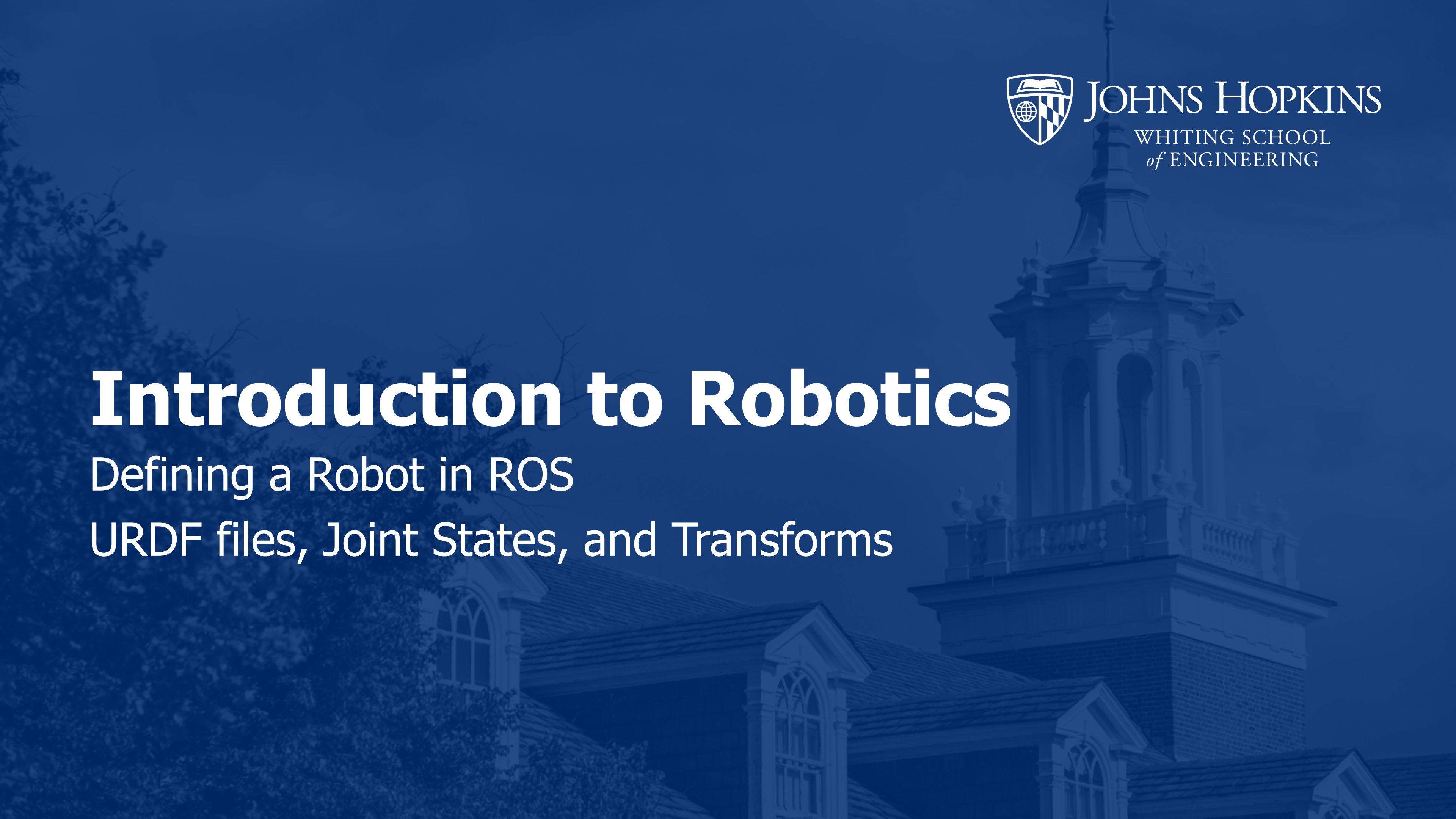
JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Midterm Part 3 – Robot Controller Node

- Implements a PID controller for a differential drive robot
  - Select pose offset and gains yourself
  - Only use proportional and derivative terms
- Subscribes to /goal_pose topic and uses the latest value as the desired goal command
- Runs at 30 hz
  - Gets the pose of the robot from the transform between 'odom' and 'chassis' frames
  - Estimate the robot velocity using the difference between the previous state and current state
  - Compute the desired forward speed and turn rate $[v,\omega]$
  - Publish the desired velocities to /cmd_vel




JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING
GILMAN HALL



JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

Introduction to Robotics

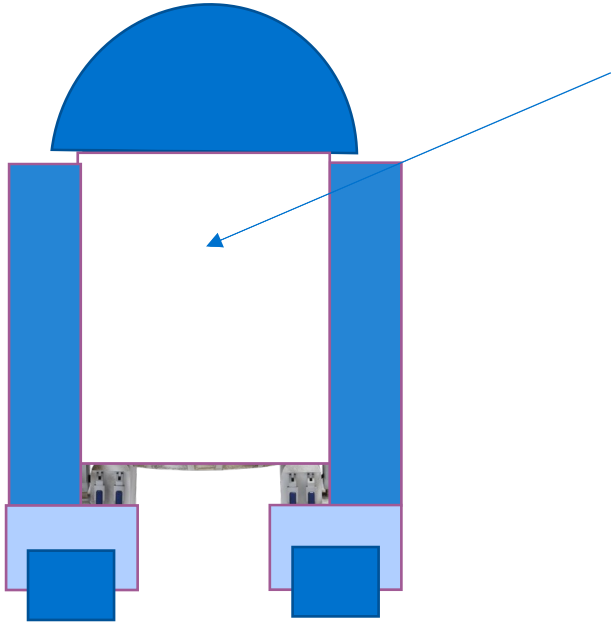
Defining a Robot in ROS

URDF files, Joint States, and Transforms

ROS WIKI Tutorials

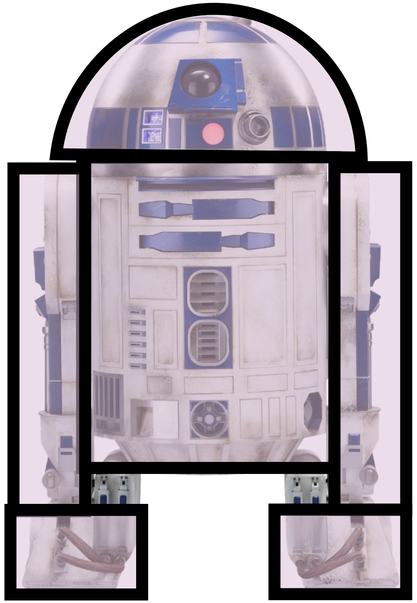
When in doubt check the official tutorials, linked in Canvas

URDF – Collision / Basic Visual Geometry



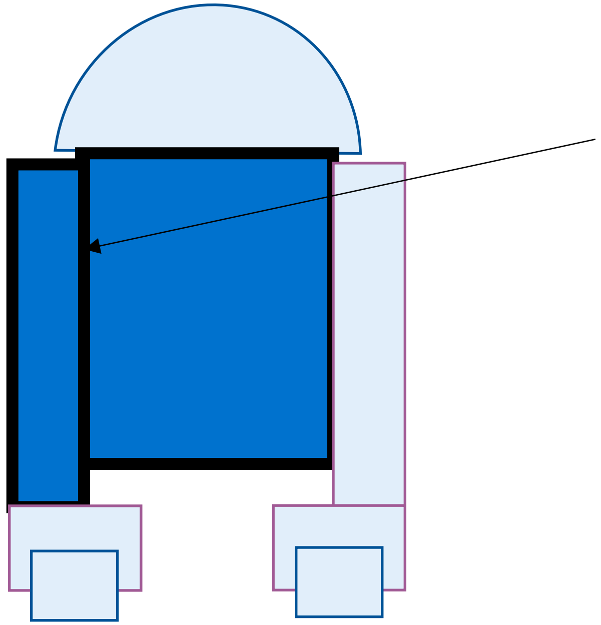
```
<?xml version="1.0"?> <robot name="visual">
<material name="blue"> <color rgba="0 0 0.8 1"/> </material>
<link name="base_link">
  <collision>
   <origin xyz="0 0 0" rpy="0 0 0 " />
     <geometry>
      <cylinder length=“0.6” radius=“0.2”/>
     </geometry>
   </collision>
   <visual>
    <origin xyz="0 0 0" rpy="0 0 0 " />
    <geometry>
      <cylinder length=“0.6” radius=“0.2”/>
     </geometry>
          <material name="blue"/>
   </visual>
</link>
…..other links and joints….
</robot>
```

URDF - Visual Geometry / Meshes



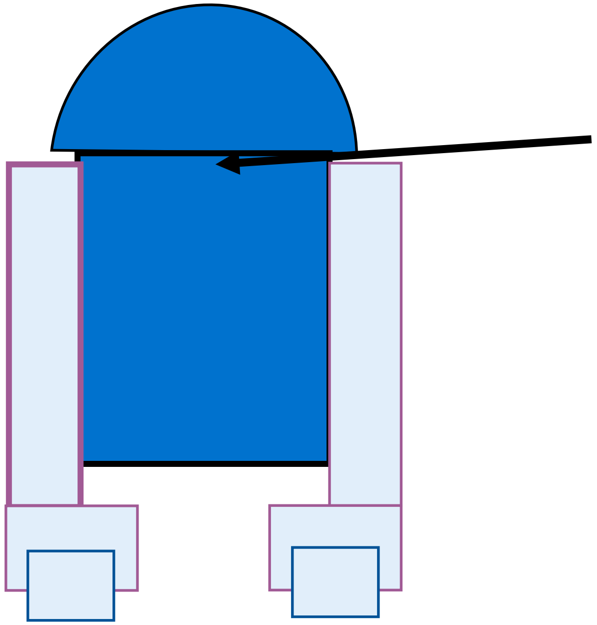
```
<!-- specifies resource location of the mesh file -->
<resource type="stl_meshes" location="/home/r2d2/meshes"/>
 <link name="base_link">
   <visual>
     <origin xyz="0 0 0" rpy="0 0 0 " />
       <geometry>
      <mesh filename=”body.stlb" scale="0.001 0.001 0.001" />
        </geometry>
          <material name="my_color">
          <color rgb="255 255 255" alpha="0.5"/>
          </material>
       </visual>
... inertial/joint/collision ...
</link>
```

URDF – Fixed Joints



```
<link name="base_link">
        <visual> <geometry>
        <cylinder length="0.6" radius="0.2"/>
        </geometry>
        <material name="blue"/> </visual> </link>
<link name="right_leg">
        <visual> <geometry>
        <box size="0.6 0.1 0.2"/> </geometry>
         <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
        <material name="white"/> </visual></link>
<joint name="base_to_right_leg"
type="fixed">
        <parent link="base_link"/>
        <child link="right_leg"/>
        <origin xyz="0 -0.22 0.25"/>
        </joint>
```

URDF – Revolute Joints



```
<link name="base_link">
        <visual> <geometry>
        <cylinder length="0.6" radius="0.2"/>
        </geometry>
        <material name="blue"/> </visual> </link>
<link name="head">
        <visual> <geometry>
        <sphere radius="0.2"/> </geometry>
        <material name="white"/> </visual> </link>
```

<joint name="head_swivel" type=”continuous">

```
<joint name="head_swivel" type=”continuous">
        <parent link="base_link"/>
        <child link="head"/>
        <origin xyz="0 0 0.3"/>
        </joint>
```

URDF Visualizing the Results

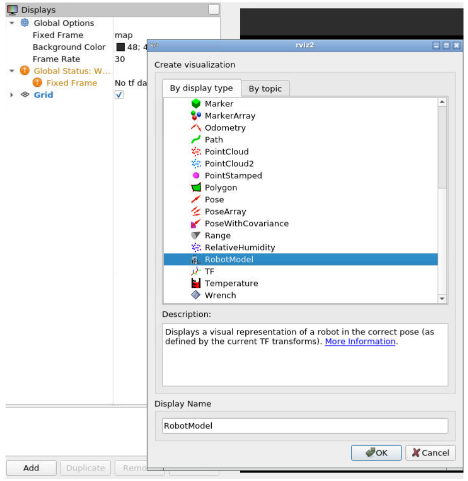
1. Open URDF with robot_state_publisher
    - Publishes the geometry information
    - Computes the transforms between nodes
    - ros2 run joint_state_publisher joint_state_publisher <my_urdf_file>

2. Open URDF with joint_state_publisher
    - Provides the joint values so the transforms can be calculated
    - ros2 run joint_state_publisher joint_state_publisher <my_urdf_file>
    - You may need to install this with apt install ros2-foxy-joint-state-publisher
    - This is not necessary when running your simulator node

URDF Visualizing the Results (cont)

3. Open RVIZ

- Add the RobotModel display
- Set the topic to /robot_description
- Set the Fixed Frame field to your base_link name



Displays
- Global Options
  - Fixed Frame: map
  - Background Color: ■ 48; 4
  - Frame Rate: 30
- Global Status: W...
  - Fixed Frame: No tf da
- Grid ✓

rviz2

Create visualization

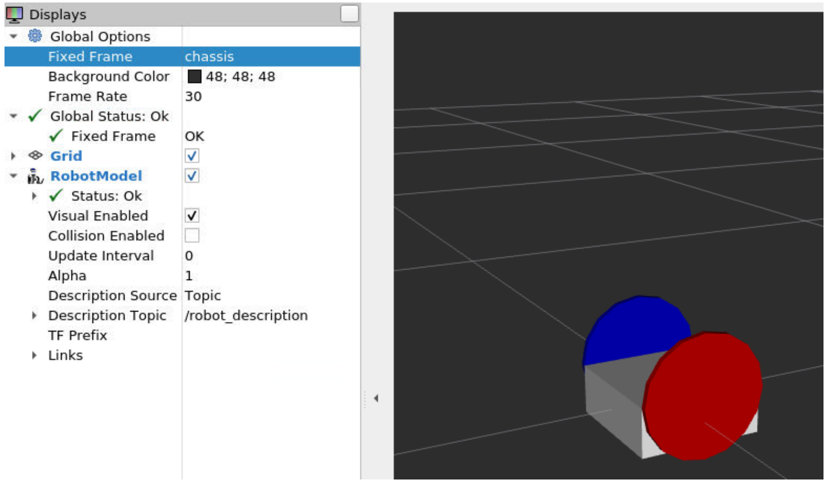
By display type | By topic

- Marker
- MarkerArray
- Odometry
- Path
- PointCloud
- PointCloud2
- PointStamped
- Polygon
- Pose
- PoseArray
- PoseWithCovariance
- Range
- RelativeHumidity
- RobotModel
- TF
- Temperature
- Wrench

Description:

Displays a visual representation of a robot in the correct pose (as defined by the current TF transforms). More Information.

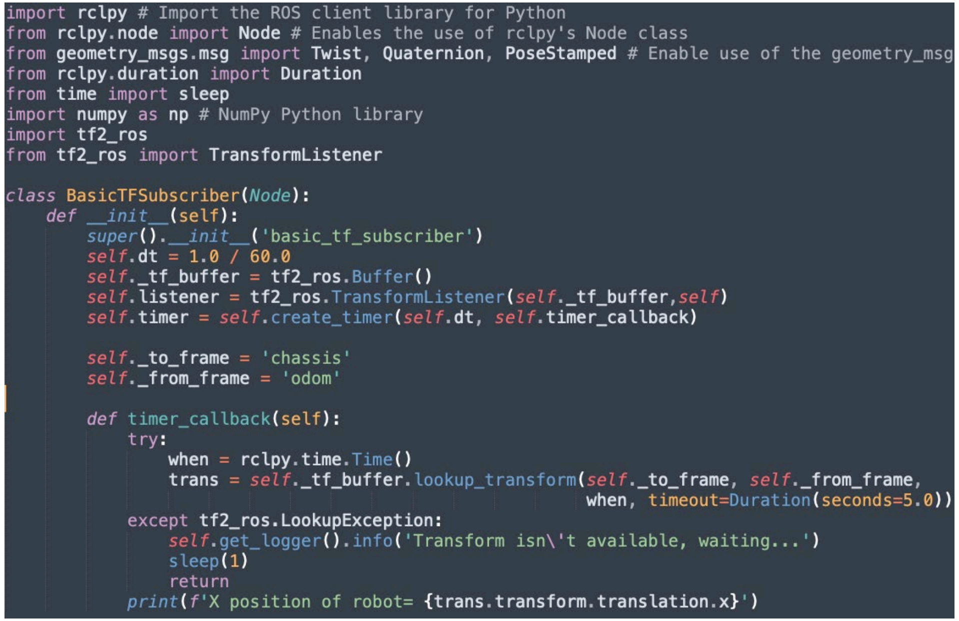
Display Name: RobotModel



Displays
- Global Options
  - Fixed Frame: chassis
  - Background Color: ■ 48; 48; 48
  - Frame Rate: 30
- Global Status: Ok
  - Fixed Frame: OK
- Grid ✓
- RobotModel ✓
  - Status: Ok
  - Visual Enabled: ✓
  - Collision Enabled: ☐
  - Update Interval: 0
  - Alpha: 1
  - Description Source: Topic
  - Description Topic: /robot_description
  - TF Prefix
  - Links

Writing a Loop

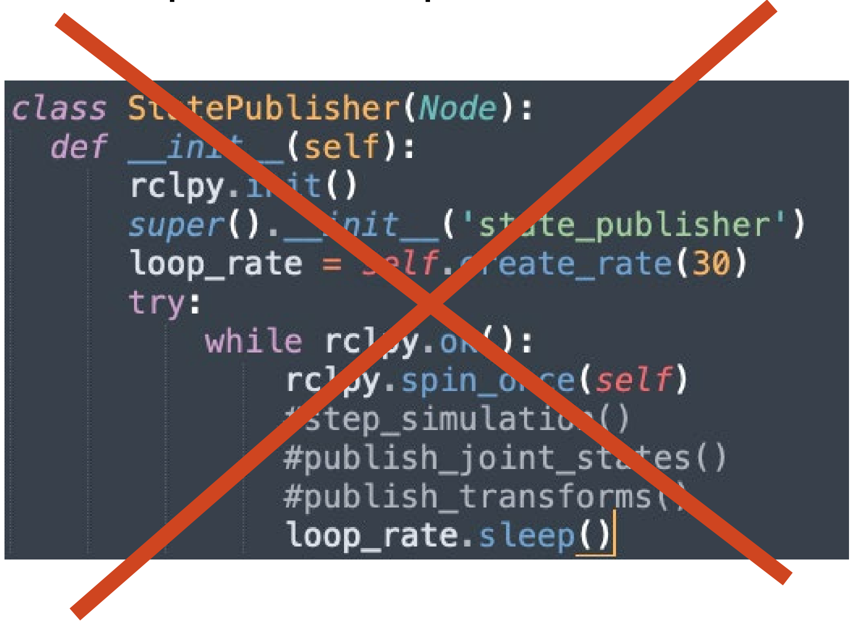
Option 1: Timer



```
import rclpy # Import the ROS client library for Python
from rclpy.node import Node # Enables the use of rclpy's Node class
from geometry_msgs.msg import Twist, Quaternion, PoseStamped # Enable use of the geometry_msg
from rclpy.duration import Duration
from time import sleep
import numpy as np # NumPy Python library
import tf2_ros
from tf2_ros import TransformListener

class BasicTFSubscriber(Node):
    def __init__(self):
        super().__init__('basic_tf_subscriber')
        self.dt = 1.0 / 60.0
        self._tf_buffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self._tf_buffer,self)
        self.timer = self.create_timer(self.dt, self.timer_callback)

        self._to_frame = 'chassis'
        self._from_frame = 'odom'

    def timer_callback(self):
        try:
            when = rclpy.time.Time()
            trans = self._tf_buffer.lookup_transform(self._to_frame, self._from_frame,
                                                     when, timeout=Duration(seconds=5.0))
        except tf2_ros.LookupException:
            self.get_logger().info('Transform isn\'t available, waiting...')
            sleep(1)
            return
        print(f'X position of robot= {trans.transform.translation.x}')
```

The preferred method

Option 2: Loop in __init__

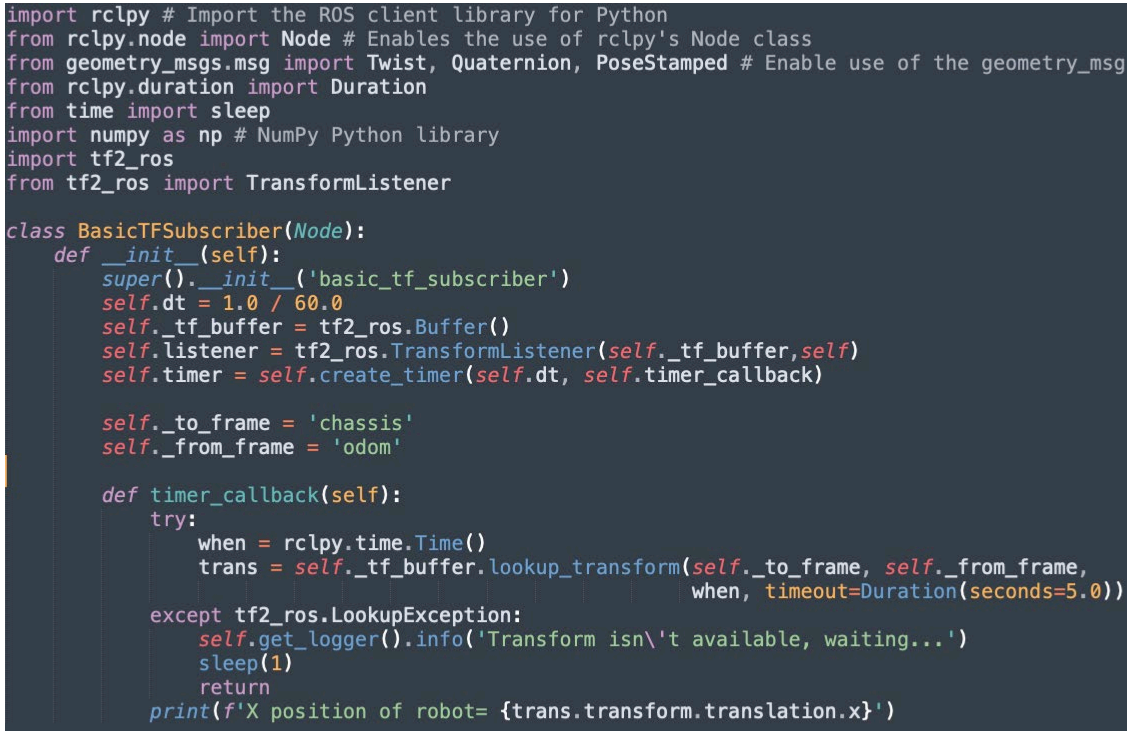


```
class StatePublisher(Node):
    def __init__(self):
        rclpy.init()
        super().__init__('state_publisher')
        loop_rate = self.create_rate(30)
        try:
            while rclpy.ok():
                rclpy.spin_once(self)
                #step_simulation()
                #publish_joint_states()
                #publish_transforms()
                loop_rate.sleep()
```

Do not use: Will never execute subscriber callbacks

Subscribing to TF

- See Example in Assignment2 - coordinates.py



```python
import rclpy # Import the ROS client library for Python
from rclpy.node import Node # Enables the use of rclpy's Node class
from geometry_msgs.msg import Twist, Quaternion, PoseStamped # Enable use of the geometry_msg
from rclpy.duration import Duration
from time import sleep
import numpy as np # NumPy Python library
import tf2_ros
from tf2_ros import TransformListener

class BasicTFSubscriber(Node):
    def __init__(self):
        super().__init__('basic_tf_subscriber')
        self.dt = 1.0 / 60.0
        self._tf_buffer = tf2_ros.Buffer()
        self.listener = tf2_ros.TransformListener(self._tf_buffer,self)
        self.timer = self.create_timer(self.dt, self.timer_callback)

        self._to_frame = 'chassis'
        self._from_frame = 'odom'

        def timer_callback(self):
            try:
                when = rclpy.time.Time()
                trans = self._tf_buffer.lookup_transform(self._to_frame, self._from_frame,
                                                         when, timeout=Duration(seconds=5.0))
            except tf2_ros.LookupException:
                self.get_logger().info('Transform isn\'t available, waiting...')
                sleep(1)
                return
            print(f'X position of robot= {trans.transform.translation.x}')
```

Publishing Joint states and TF

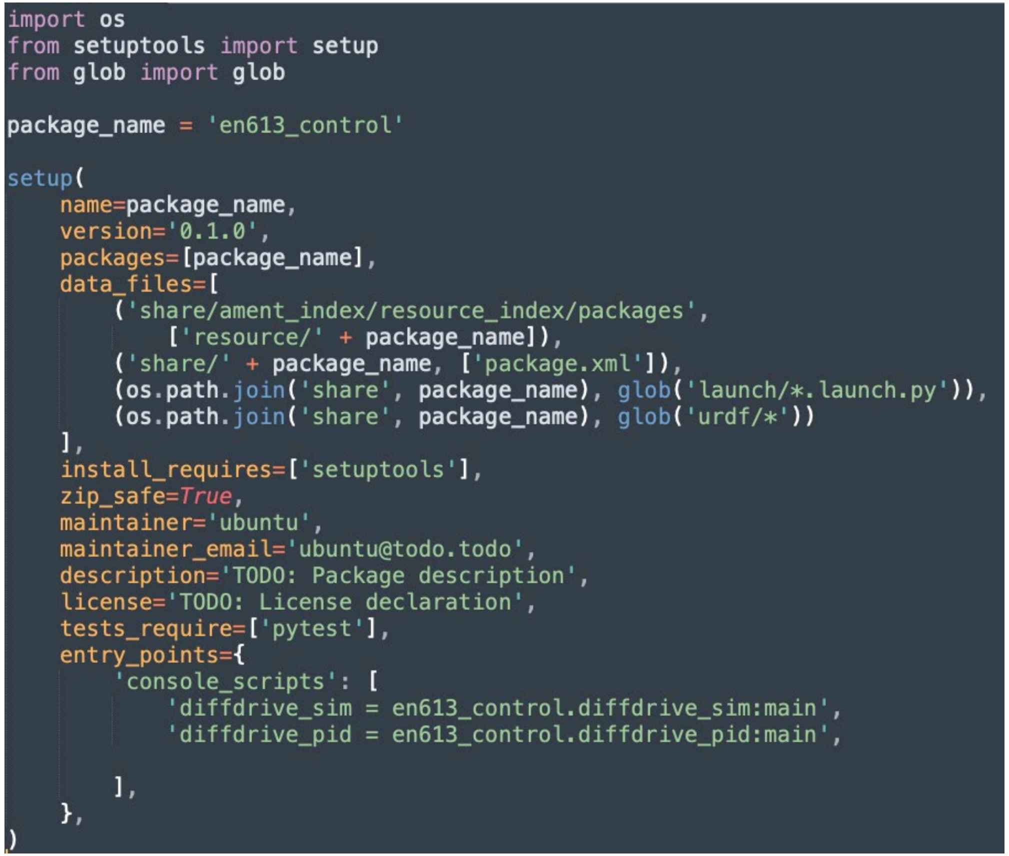
- Refer to the tutorial
- https://docs.ros.org/en/foxy/Tutorials/URDF/Using-URDF-with-Robot-State-Publisher.html

```python
class StatePublisher(Node):

    def __init__(self):
        rclpy.init()
        super().__init__('state_publisher')

        qos_profile = QoSProfile(depth=10)
        self.joint_pub = self.create_publisher(JointState, 'joint_states', qos_profile)
        self.broadcaster = TransformBroadcaster(self, qos=qos_profile)
        self.nodeName = self.get_name()
        self.get_logger().info("{0} started".format(self.nodeName))
```

```python
        # message declarations
        odom_trans = TransformStamped()
        odom_trans.header.frame_id = 'odom'
        odom_trans.child_frame_id = 'axis'
        joint_state = JointState()
```

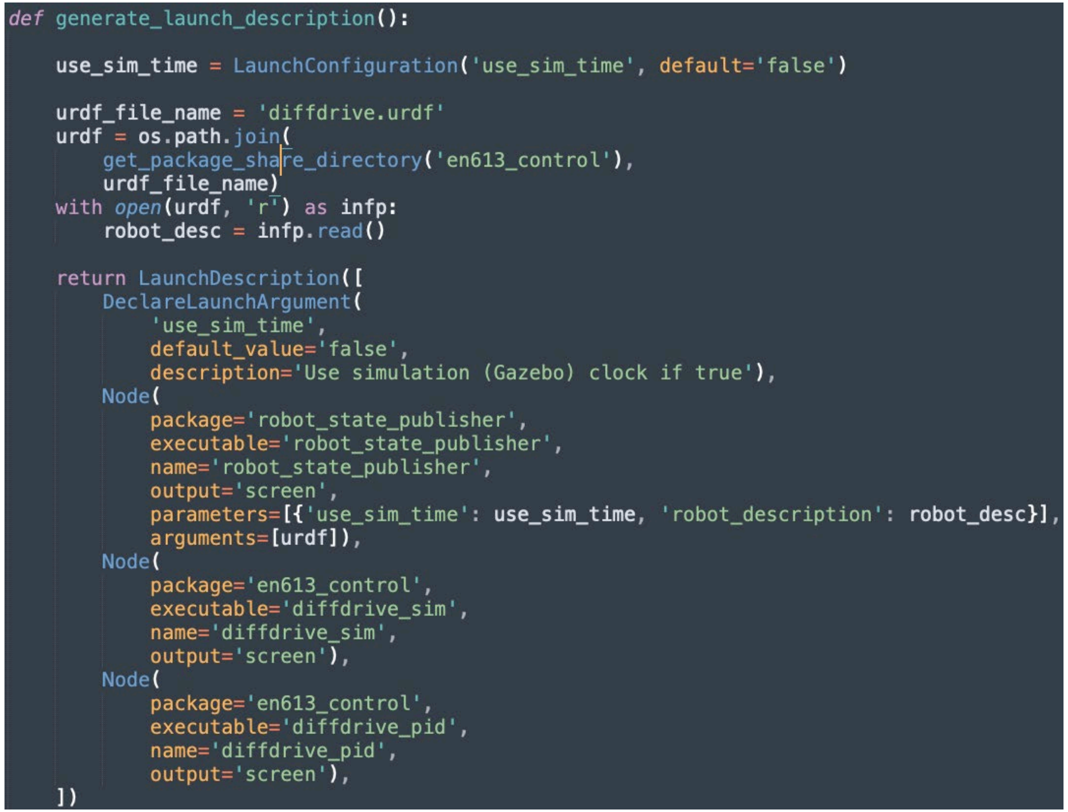
```python
# update joint_state
now = self.get_clock().now()
joint_state.header.stamp = now.to_msg()
joint_state.name = ['swivel', 'tilt', 'periscope']
joint_state.position = [swivel, tilt, height]

# update transform
# (moving in a circle with radius=2)
odom_trans.header.stamp = now.to_msg()
odom_trans.transform.translation.x = cos(angle)*2
odom_trans.transform.translation.y = sin(angle)*2
odom_trans.transform.translation.z = 0.7
odom_trans.transform.rotation = \
    euler_to_quaternion(0, 0, angle + pi/2) # roll,pitch,yaw

# send the joint state and transform
self.joint_pub.publish(joint_state)
self.broadcaster.sendTransform(odom_trans)
```

Setup.py Example

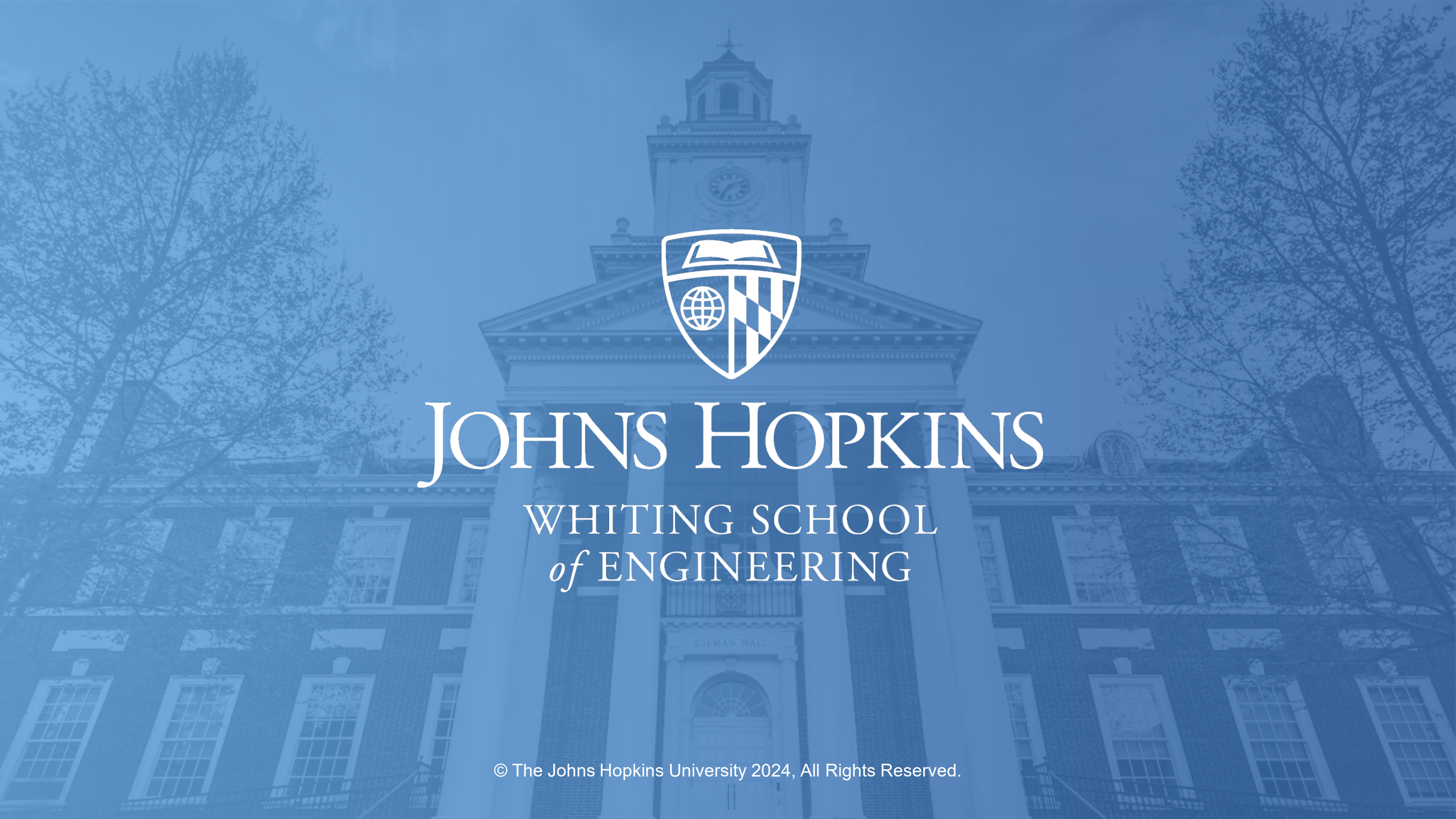


```
import os
from setuptools import setup
from glob import glob

package_name = 'en613_control'

setup(
    name=package_name,
    version='0.1.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py')),
        (os.path.join('share', package_name), glob('urdf/*'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='ubuntu',
    maintainer_email='ubuntu@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'diffdrive_sim = en613_control.diffdrive_sim:main',
            'diffdrive_pid = en613_control.diffdrive_pid:main',

        ],
    },
)
```

Launch File Example



```python
def generate_launch_description():

    use_sim_time = LaunchConfiguration('use_sim_time', default='false')

    urdf_file_name = 'diffdrive.urdf'
    urdf = os.path.join(
        get_package_share_directory('en613_control'),
        urdf_file_name)
    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation (Gazebo) clock if true'),
        Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'use_sim_time': use_sim_time, 'robot_description': robot_desc}],
            arguments=[urdf]),
        Node(
            package='en613_control',
            executable='diffdrive_sim',
            name='diffdrive_sim',
            output='screen'),
        Node(
            package='en613_control',
            executable='diffdrive_pid',
            name='diffdrive_pid',
            output='screen'),
    ])
```



JOHNS HOPKINS

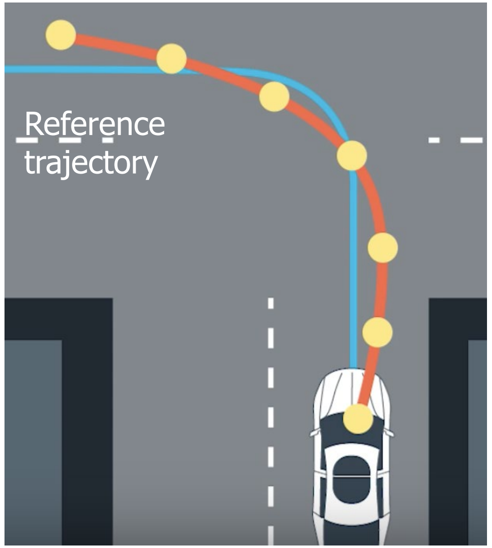
WHITING SCHOOL *of* ENGINEERING

JOHNS HOPKINS
WHITING SCHOOL *of* ENGINEERING

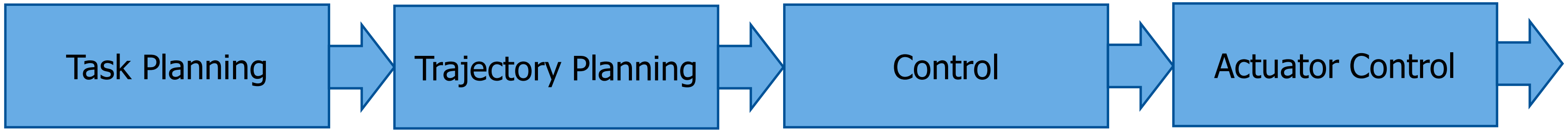
Introduction to Robotics

Proportional Integral Derivative Control

Controllers



Planning and Control Hierarchy



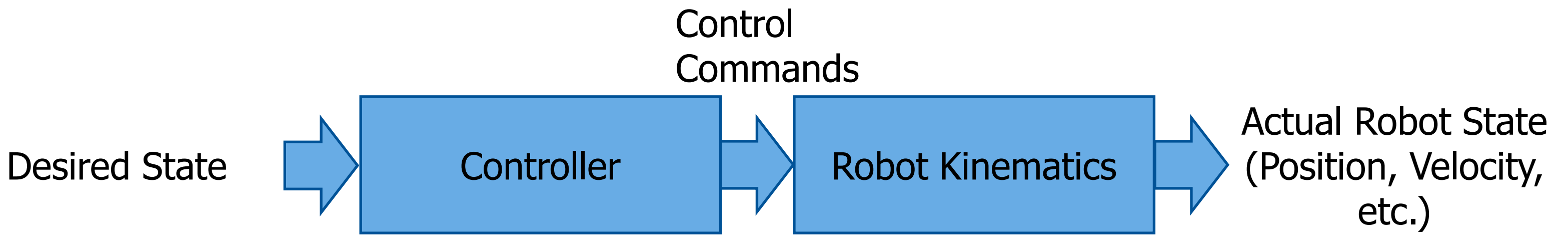
Output:
Final Goal

Output:
Robot States

Output:
Velocities

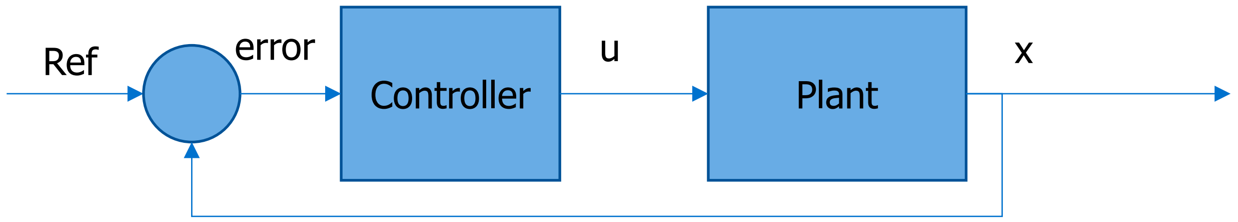
Output:
Voltage &
Current

Open Loop Controller



Controller tells your system to do something, but doesn't use the results of that action to verify the results or modify the commands to see that the job is done properly

Closed Loop Controller



Controller attempts to reduce error between the actual state and a reference state. Using feedback to drive the system towards zero error.

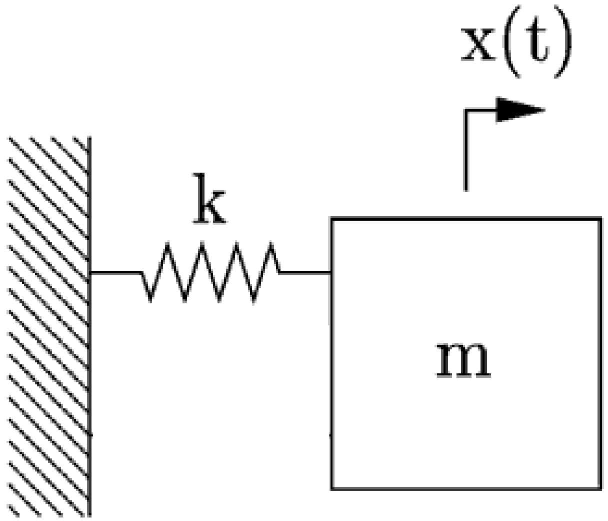
Mass-Spring

$$F = \frac{d}{dt}(m\dot{x}) = m\ddot{x}$$

$$F = -kx$$

$$\ddot{x} + \frac{k}{m}x = 0$$

$$\omega_0 = \sqrt{\frac{k}{m}}$$ Natural Frequency



Mass-Spring Damper

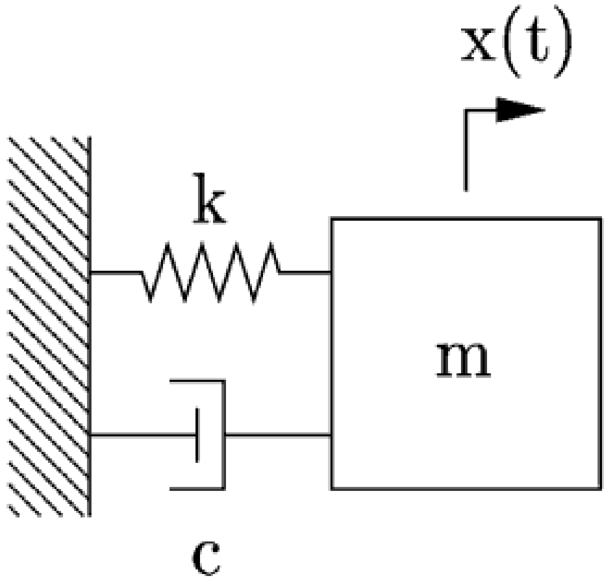
$$F = \frac{d}{dt}(m\dot{x}) = m\ddot{x}$$

$$F_{\text{spring}} = -kx$$

$$F_{\text{damper}} = -cv = -c\frac{dx}{dt} = -c\dot{x}$$

$$F_{\text{total}} = m\frac{d^2x}{dt^2} = m\ddot{x}$$

$$m\ddot{x} = -kx - c\dot{x}$$



Mass-Spring Damper (cont.)

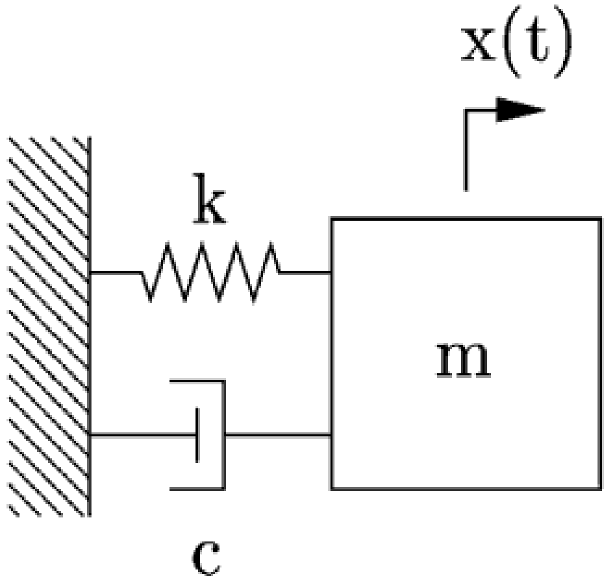
2nd order ode

$$\ddot{x} + \frac{c}{m}\dot{x} + \frac{k}{m}x = 0$$

$$\omega_0 = \sqrt{\frac{k}{m}}$$ Natural Frequency

$$\xi = \frac{c}{2\sqrt{mk}}$$ Dampling ratio

$$\ddot{x} + 2\xi\omega_0\dot{x} + \omega_0^2 x = 0$$



2nd Order ODE Solutions

$$\ddot{x} + 2\xi\omega_0\dot{x} + \omega_0^2 x = 0$$

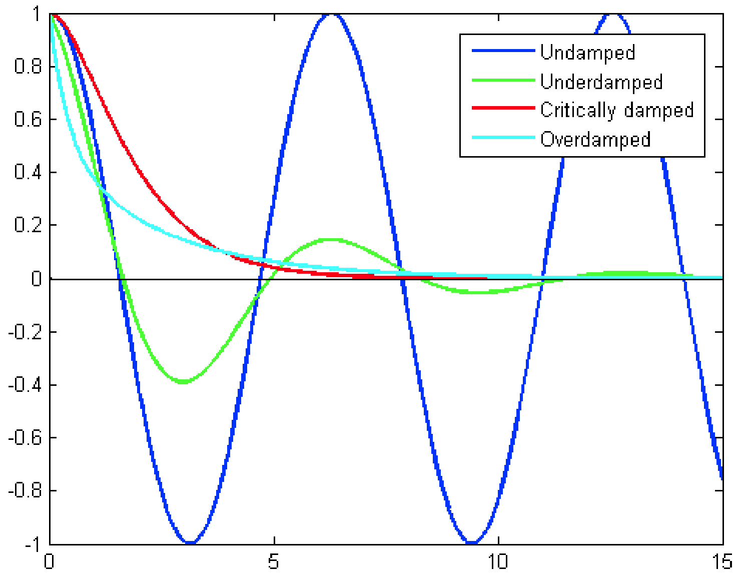
$$\omega_0 = \sqrt{\frac{k}{m}}$$ Natural Frequency

$$\xi = \frac{c}{2\sqrt{mk}}$$ Dampling ratio

Critically dampled $(\xi = 1)$

Overdamped $(\xi > 1)$

Underdampled $(\xi < 1)$



PID Feedback

P: Proportional

I: Integral

D: Derivative

$$\frac{d^2x}{dt^2} + 2\zeta\omega_0\frac{dx}{dt} + \omega_0 x = F(t) = -u(t)$$

$$u(t) = K_p e(t) + K_I \int e(t)dt + K_D \frac{d}{dt}e(t)$$

JOHNS HOPKINS WHITING SCHOOL of ENGINEERING

Proportional Feedback

$$\frac{d^2x}{dt^2} + 2\zeta\omega_0\frac{dx}{dt} + \omega_0^2 x = -Kx$$

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \left(\omega_0^2 + K\right)x = 0$$

$$u(t) = K_p e(t) + K_I \int e(t)dt + K_D \frac{d}{dt}e(t)$$

$$e(t) = x_{desired}(t) - x_{actual}(t)$$

Basic PID Control in Python

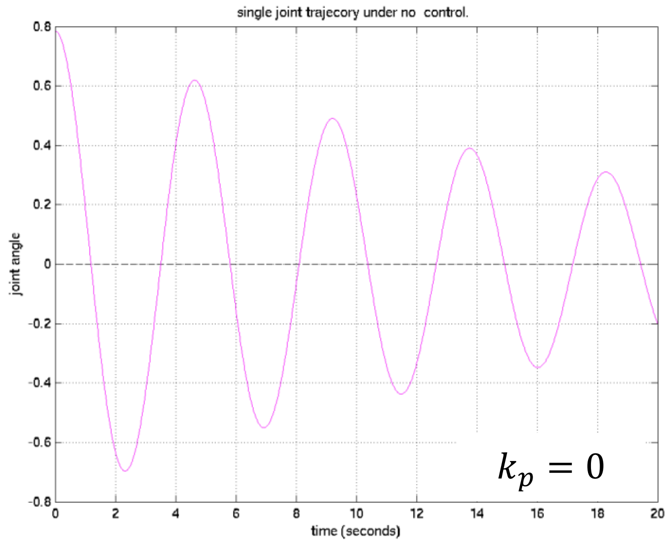
```python
class PIDController:
    def __init__(self, kp, ki, kd):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.last_error = 0
        self.integral = 0

    def update(self, error, dt):
        self.integral += error * dt
        derivative = (error - self.last_error) / dt
        output = self.kp * error + self.ki * self.integral + self.kd * derivative
        self.last_error = error
        return output
```

Note: This function takes *error* as input, not a desired set point

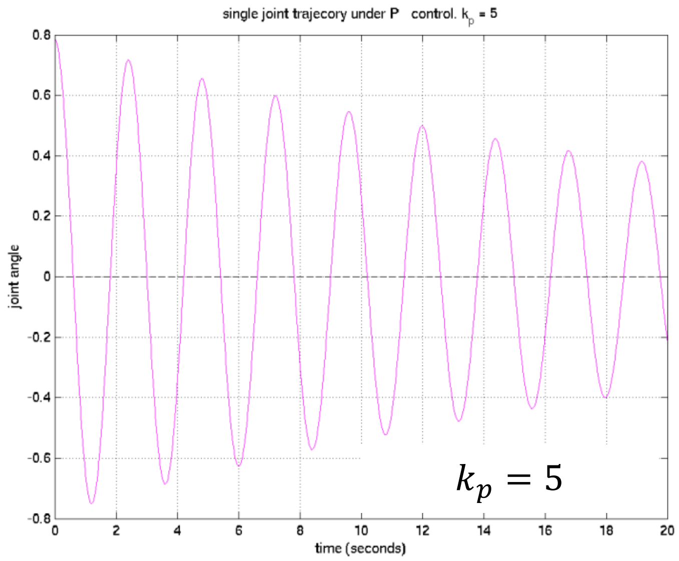
JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Proportional Feedback (cont.)

Set desired position to zero
Note that the oscillation dies out at approximately the same rate but has higher frequency.
This can be thought of as “stiffening the spring”.



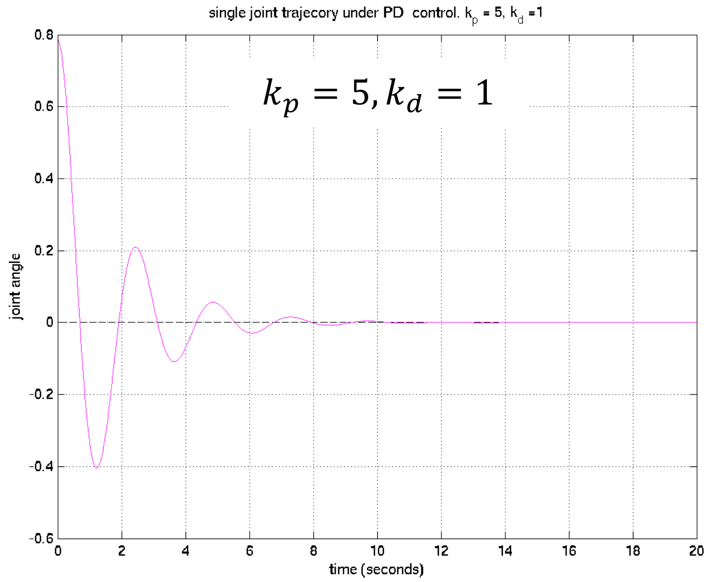
$k_p = 0$

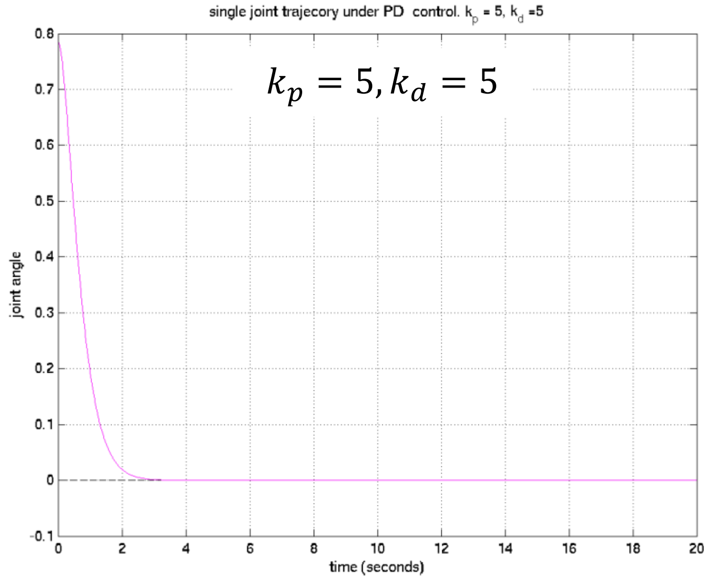


$k_p = 5$

Proportional/Damping

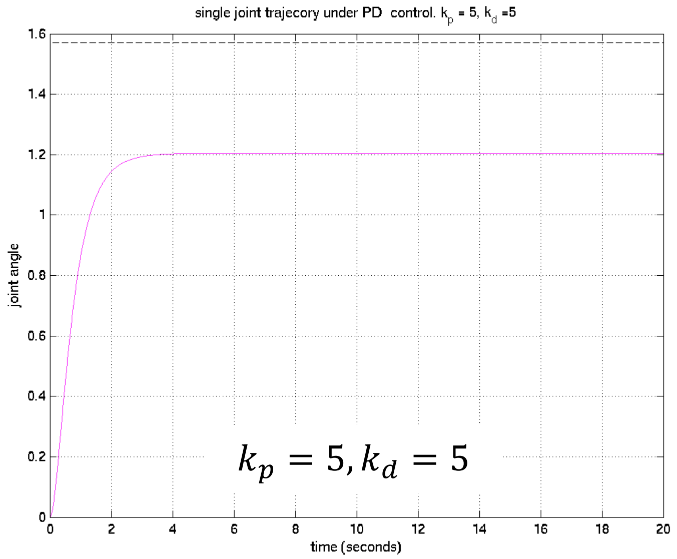
We can increase the damping (i.e., increase the rate at which the oscillation dies out)





Non-zero Desired PD

At set point, applying no force so end up settling at equilibrium that balances force due to error and force due to spring (damper goes away in steady state because depends on derivative). Crank up P gain, steady state error gets smaller, but that causes overshoot, oscillations, etc. which you don’t want

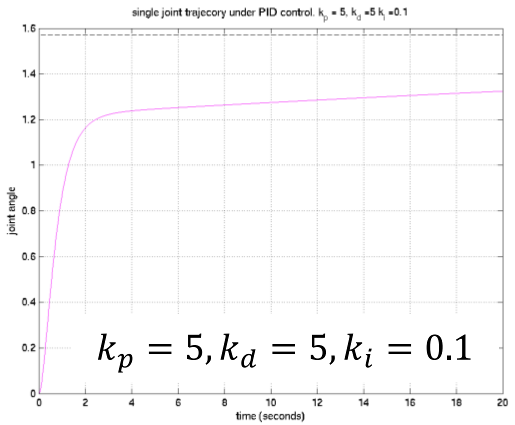


$Xd = 1.6$

Settle time same

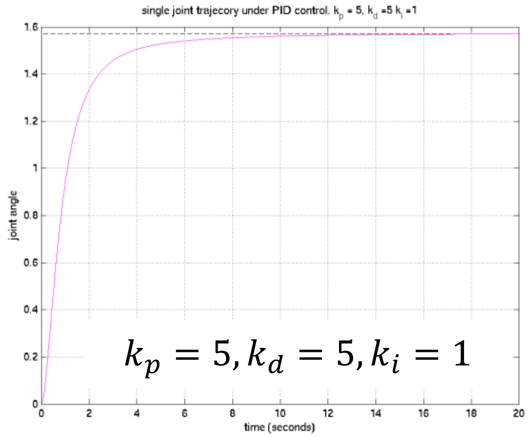
Steady state error!

PID Control



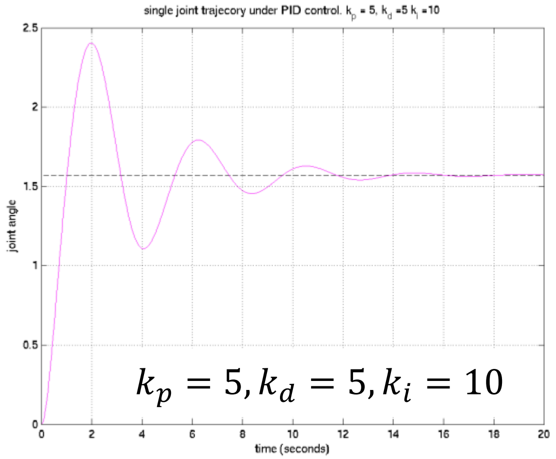
$k_p = 5, k_d = 5, k_i = 0.1$

System does its dynamic thing and then gradually integrates to correct for steady state error



$k_p = 5, k_d = 5, k_i = 1$

As increase I gain, gets faster, good response

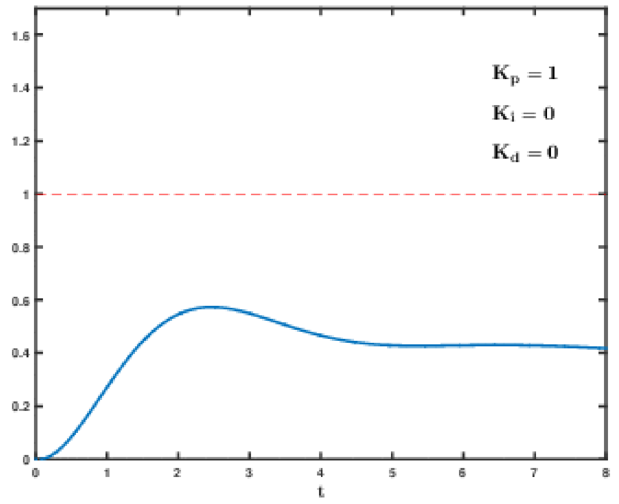


$k_p = 5, k_d = 5, k_i = 10$

Integral gets so bad, it starts to interfere with other dynamics, lead to unintended motions which could lead to instability

Quick and Dirty Tuning

- Basic Method
  - Tune P to get the rise time you want
  - Tune D to get the setting time you want
  - Tune I to get rid of steady state error
  - Repeat
- More rigorous methods – Ziegler Nichols, Self-tuning,
- Scary thing happen when you introduce the I term
  - Wind up (example with brick wall)
  - Instability around set point



Mobile Robot

- planar workspace
- position of robot and goal are known
- omni-directional robot
- control input is velocity:

$$\begin{bmatrix} u_x \\ u_y \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

Proportional (P) Control

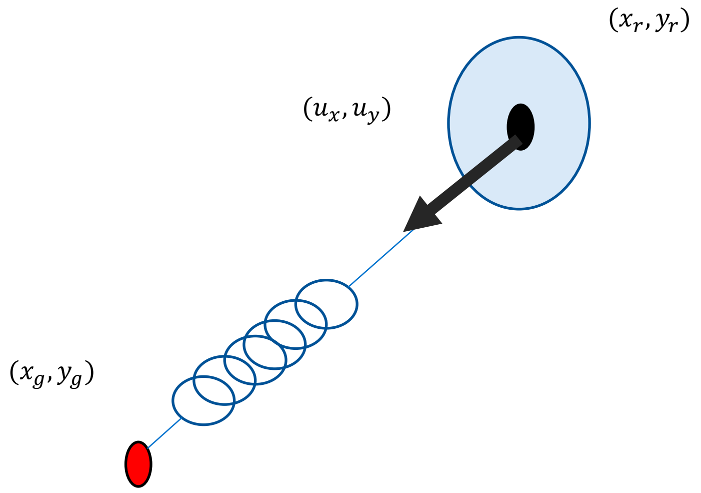
$$\begin{bmatrix} u_x \\ u_y \end{bmatrix} = -k_p \begin{bmatrix} x_g - x_r \\ y_g - y_r \end{bmatrix}$$

The equation above is called a **control law**

k_p is called the **proportional gain**

k_p is a tunable parameter

physically, k_p is the stiffness of the spring



Proportional-Derivative Control

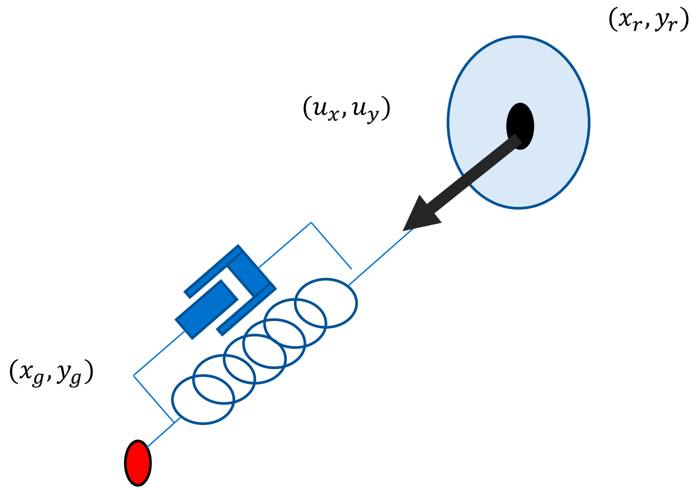
$$\begin{bmatrix} u_x \\ u_y \end{bmatrix} = -k_p \begin{bmatrix} x_g - x_r \\ y_g - y_r \end{bmatrix} - k_d \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

k_d is called the **derivative gain**

k_p and k_d are tunable parameters

physically, k_d is the damping term

all of the stuff about P control still applies



Add a damper to the system

Robot Inputs

So far we've assumed direct velocity control over the robots center of mass

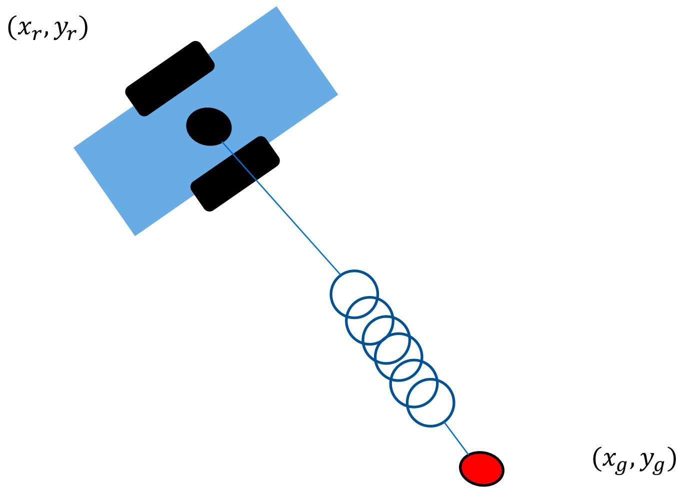
$$u = \begin{bmatrix} u_x \\ u_y \end{bmatrix} = \begin{bmatrix} \dot{x}_r \\ \dot{y}_r \end{bmatrix}$$

In reality what we control is the velocities of the wheels or turning angles

$$u = \begin{bmatrix} v_l \\ v_r \end{bmatrix} \rightarrow \begin{bmatrix} v_f \\ \omega \end{bmatrix}$$

The Problem

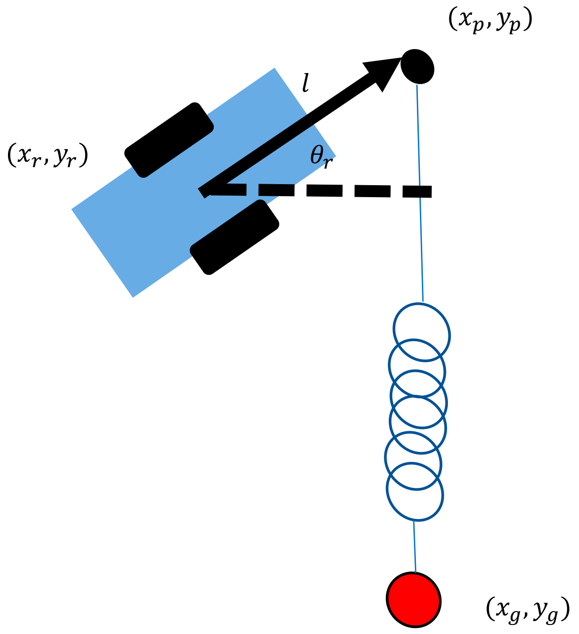
- The ‘force’ being applied is perpendicular to the direction of travel
- Vehicles with non-holonomic dynamics cannot travel sideways!
- P or PD control won’t work.
- No smooth control law will!



A Simple Solution

$$\begin{bmatrix} x_p \\ y_p \\ \theta_r \end{bmatrix} = \begin{bmatrix} x_r + l\cos\theta \\ y_r + l\sin\theta \\ \theta_r \end{bmatrix}$$

$$\begin{bmatrix} \dot{x}_p \\ \dot{y}_p \\ \dot{\theta}_r \end{bmatrix} = \begin{bmatrix} \dot{x}_r - l\dot{\theta}\sin\theta_r \\ \dot{y}_r + l\dot{\theta}\cos\theta_r \\ \omega_r \end{bmatrix} = \begin{bmatrix} v_f\cos\theta_r - wl\sin\theta_r \\ v_f\sin\theta_r + \omega l\cos\theta_r \\ \omega \end{bmatrix}$$



Like a rigid trailer hitch (not driving to point)

A Simple Solution (cont.)

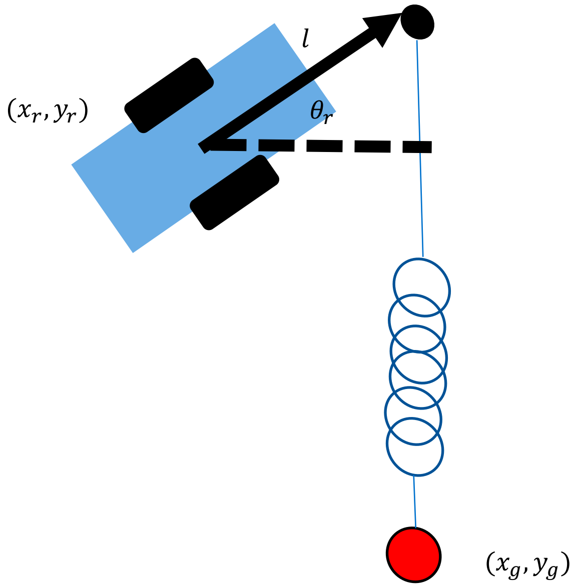
If we ignore orientation

$$\begin{bmatrix} \dot{x}_p \\ \dot{y}_p \end{bmatrix} = \begin{bmatrix} \cos\theta_r & -\sin\theta_r \\ \sin\theta_r & \cos\theta_r \end{bmatrix} \begin{bmatrix} v_f \\ \omega l \end{bmatrix}$$

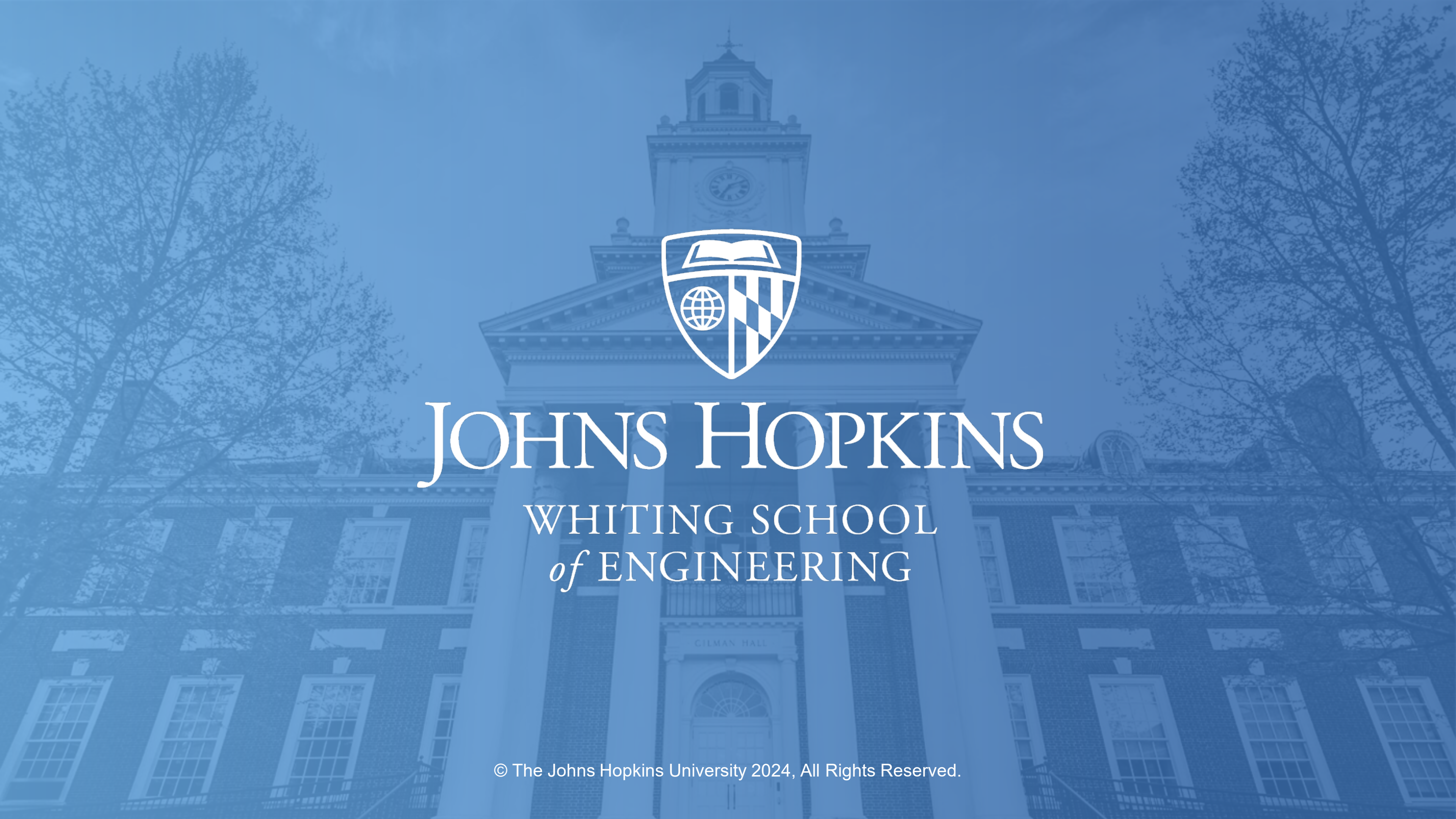
$$\begin{bmatrix} v_f \\ \omega l \end{bmatrix} = \begin{bmatrix} \cos\theta_r & \sin\theta_r \\ -\sin\theta_r & \cos\theta_r \end{bmatrix} \begin{bmatrix} \dot{x}_p \\ \dot{y}_p \end{bmatrix}$$

so we can implement the PD control law as

$$\begin{bmatrix} v_f \\ \omega l \end{bmatrix} = \begin{bmatrix} \cos\theta_r & \sin\theta_r \\ -\sin\theta_r & \cos\theta_r \end{bmatrix} \left(-k_p \begin{bmatrix} x_g - x_p \\ y_g - y_p \end{bmatrix} - k_d \begin{bmatrix} v_x \\ v_y \end{bmatrix} \right)$$



Like a rigid trailer hitch (not driving to point)



JOHNS HOPKINS

WHITING SCHOOL of ENGINEERING