lab6

November 26, 2019

1 EE 120 Lab 6: Control

v1 - Spring 2019: Jonathan Lee, Akash Velu, Babak Ayazifar

v2 - Fall 2019: Jonathan Lee, Murat Arcak

1.1 Background

Most interesting systems in the physical world use feedback to regulate or stabilize themselves. For example, imagine you're building the steering system for a car. Most modern cars are drive-by-wire, meaning that instead of a direct mechanical link from the steering wheel to the front wheels, there's a computer driving a motor to perform steering.

In an open-loop system, you would simply program the computer to have the wheel angle track the steering wheel and hope that the angles match. However, you might consider a closed-loop system instead where you use a sensor to measure the actual wheel angle, then drive the motor until it matches the reference angle.

- The controller design is greatly simplified, since the process we are trying to control (called a plant) can be treated as a "black-box".
- Open-loop systems tend to exhibit drift from their desired outputs over time due to modeling imperfections, disturbances, and noise. Under a closed-loop system, the input compensates for errors as they occur.
- Closed-loop control can correct for errors at much higher frequency than a human operator can.

1.1.1 Dependencies

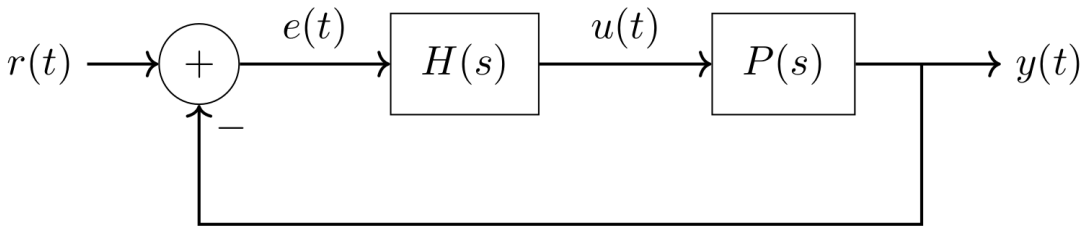
In addition to the `scipy` stack, we'll need the `python-control` package to visualize root locus. You can install the package with one of the following commands (depending on whether you use the `pip` Python package manager or Anaconda):

```
$ pip install --user control
$ conda install -c conda-forge control
```

If you're having trouble installing the package, post on Piazza with your OS, Python version, and other debug information, or get help at office hours.

[78]:
```
from __future__ import division, print_function, unicode_literals
from matplotlib.animation import FuncAnimation
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from IPython.display import HTML
import control

if 'ggplot' in plt.style.available:
    plt.style.use('ggplot')
%matplotlib inline
```

1.2 Q1: Proportional-Integral-Derivative (PID) Control



A block diagram of a general reference tracking scheme we're going to look at is shown above. - $P(s)$ is the Laplace transform (LT) of the plant. We assume the plant can be well-modeled as an LTI system (for example, a differential equation). - $H(s)$ is the LT of the controller. This is the part we'd like to design. - $r(t)$ is the desired reference signal (also known as the "setpoint"). - $y(t)$ is the output signal. - $e(t) = r(t) - y(t)$ is the error signal. - $u(t)$ is the control input.

Given a reference signal $r(t)$, we would like $r(t)$ to closely track $y(t)$. That is, ideally, we'd like $e(t) \approx 0$.

We can derive the closed-loop transfer function:

$$Y(s) = P(s)\,H(s)\,[R(s) - Y(s)] \tag{1}$$

$$(1 + P(s)\,H(s))\,Y(s) = P(s)\,H(s)\,R(s) \tag{2}$$

$$\frac{Y(s)}{R(s)} = \frac{P(s)\,H(s)}{1 + P(s)\,H(s)} \tag{3}$$

We can also derive the LT of the error signal:

$$E(s) = R(s) - Y(s) \tag{4}$$
$$= R(s) - \frac{Y(s)}{R(s)} R(s) \tag{5}$$
$$= R(s) \left[1 - \frac{P(s)\,H(s)}{1 + P(s)\,H(s)} \right] \tag{6}$$
$$= \frac{R(s)}{1 + P(s)\,H(s)} \tag{7}$$

PID is a popular controller used to design $H(s)$. The idea is that the control input should be a linear combination of the error signal, error derivative, and integrated error:

$$u(t) = K_p e(t) + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau$$

In some sense, these encode the current, future, and past error, respectively. $K_p, K_i, K_d \geq 0$ can be tuned to control the relative importance of each term.

1.2.1 Q1(a): Step Responses

First, we'll look at an example of controlling a simple second-order plant with PID:

$$P(s) = \frac{1}{(s+2)^2}$$

[79]:
```
def plot_step_response(ax, sys):
    """ Plot the step response of a given system. """
    t, y = control.step_response(sys)
    ax.plot(t, y)
    ax.plot(t, np.where(t > 0, np.ones(y.shape), np.zeros(y.shape)))
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Response')

plant = control.tf([1], [1, 4, 4])
```

To complete this question, implement `make_pid_tf` and `make_overall_tf` according to the instructions in the docstring. You'll want to familiarize yourself with the `control` package documentation: * Creating transfer functions (also, see above for how `plant` is defined) * Composing transfer functions

The polynomials accepted by `control.tf` should be encoded as coefficients of powers of s, in decreasing order of degree. For example:

$3s^3 + 2s^2 + s$ is encoded as `[3, 2, 1, 0]`

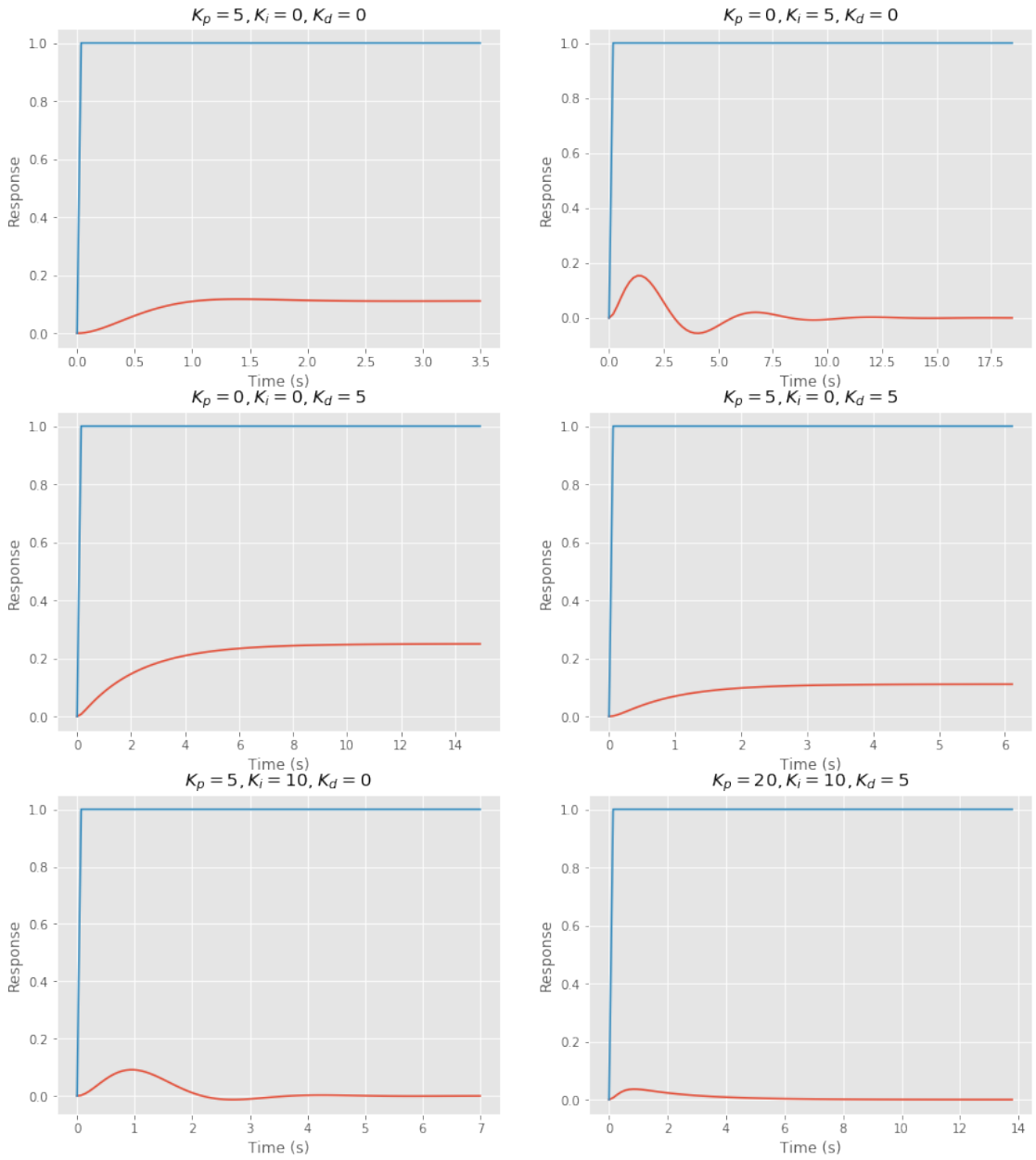
This means you'll need to derive $H(s)$ in a simple rational form.

[92]:
```
def make_pid_tf(Kp, Ki, Kd):
    """
    Make the PID controller transfer function H(s) = U(s)/E(s)
    (as defined above) with proportional, integral, and derivative
    coefficients `Kp`, `Ki`, and `Kd`.
    """
    return control.tf([Kd, Kp, Ki], [1, 0])

def make_cl_tf(plant_tf, controller_tf):
    """
    Make the closed-loop transfer function Y(s)/R(s) given the
    transfer functions of the controller and plant.

    Hint: `control.feedback(sys1, sys2)` represents this system:

                 +------+
        ----+--> | sys1 |---+--->
          - ^    +------+   |
            |               |
            |    +------+   |
            +----| sys2 |<--+
                 +------+

    If you leave out the `sys2` argument, then the output is simply
    subtracted from the closed-loop input (unit gain feedback).
    """
    return control.feedback(plant_tf, controller_tf)  # TODO: Your code here.
```

Now, we'll examine the step response $y(t)$ as we tweak the parameters K_p, K_i, K_d.

[93]:
```
gain_configs = [
    (5, 0, 0),  # P control (purely proportional)
    (0, 5, 0),
    (0, 0, 5),
    (5, 0, 5),  # PD
    (5, 10, 0),  # PI
    (20, 10, 5),  # PID
]
rows, columns = 3, 2

figure, axes = plt.subplots(rows, columns, figsize=(14, 16))
for i, gains in enumerate(gain_configs):
    controller = make_pid_tf(*gains)
    cl_tf = make_cl_tf(plant, controller)
    row, column = i//columns, i%columns
    ax = axes[row, column]
```

```
    plot_step_response(ax, cl_tf)
    ax.set_title('$K_p = {}, K_i = {}, K_d = {}$'.format(*gains))
```



Qualitatively, what effect does the derivative term seem to have? Compare the step responses of $K_d = 0$ against $K_d > 0$.

An increase in the derivative term seems to prevent oscillation of the step-response.

What effect does the integral term seem to have? Compare the step responses of $K_i = 0$ against $K_i > 0$.

The integral-term seems to affect the speed of the time to peak overshoot of the system often at the cost of stability.

Feel free to continue playing around with the coefficients and seeing how they affect the step response. As you might notice, some step responses might not converge to the reference input at all. This is due to steady-state error, which is simply a consequence of the final value theorem. In the case of the P controller,

$$\lim_{t\to\infty} e(t) = \lim_{s\to 0^+} sE(s) \tag{8}$$

$$= \lim_{s\to 0^+} s\frac{R(s)}{1+P(s)\,H(s)} \tag{9}$$

$$= \lim_{s\to 0^+} \frac{1}{1+K_pH(s)} \tag{10}$$

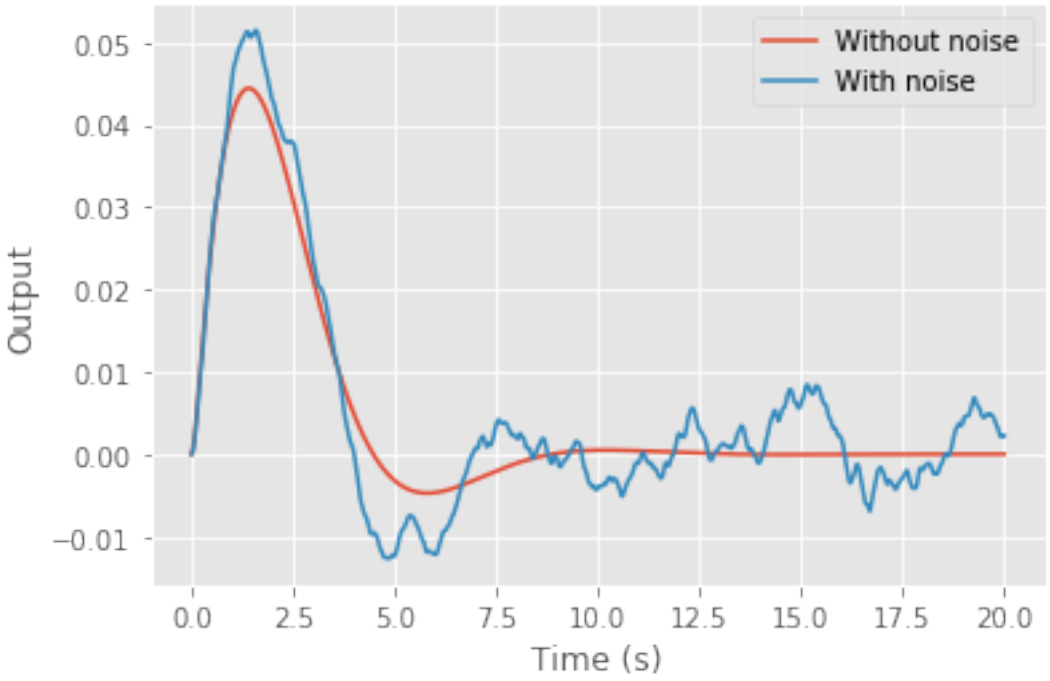
$$= \lim_{s\to 0^+} \frac{(s+2)^2}{(s+2)^2+K_p} \tag{11}$$

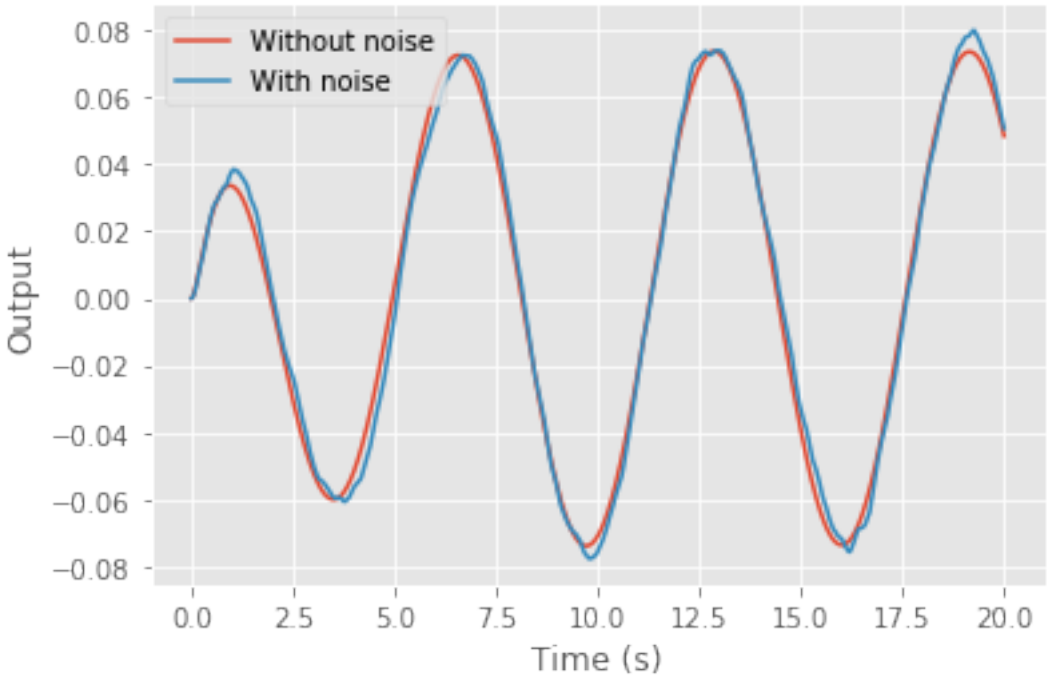
$$= \frac{4}{4+K_p} \neq 0 \tag{12}$$

1.2.2 Q1(b): Robust Differentiation

In practice, adding a pure differentiation term to the PID expression is generally not a good idea. As you saw in homework 3, differentiation can amplify the high-frequency components in noise and drown out the differentiated signal. Here, we'll take the control scheme in 1(a) and introduce some noise to the feedback path (see note 20), which will be subtracted from the reference signal.

[94]:
```
controller = make_pid_tf(10, 10, 10)
cl_sys = make_cl_tf(plant, controller).returnScipySignalLTI()[0][0]

def plot_response_with_noise(cl_sys, t_in, ref, noise):
    """
    Plot the step response of a system with and without noise,
    superimposed on the same plot.
    """
    plt.plot(*cl_sys.output(ref, t_in)[:2], label='Without noise')
    plt.plot(*cl_sys.output(ref + noise, t_in)[:2], label='With noise')
    plt.xlabel('Time (s)')
    plt.ylabel('Output')
    plt.legend()

t_in = np.linspace(0, 20, 1000)
step, cos = np.ones(t_in.shape), np.cos(t_in)
noise = np.random.normal(0, 0.5, t_in.shape)
plot_response_with_noise(cl_sys, t_in, step, noise)
plt.figure()
plot_response_with_noise(cl_sys, t_in, cos, noise)
```



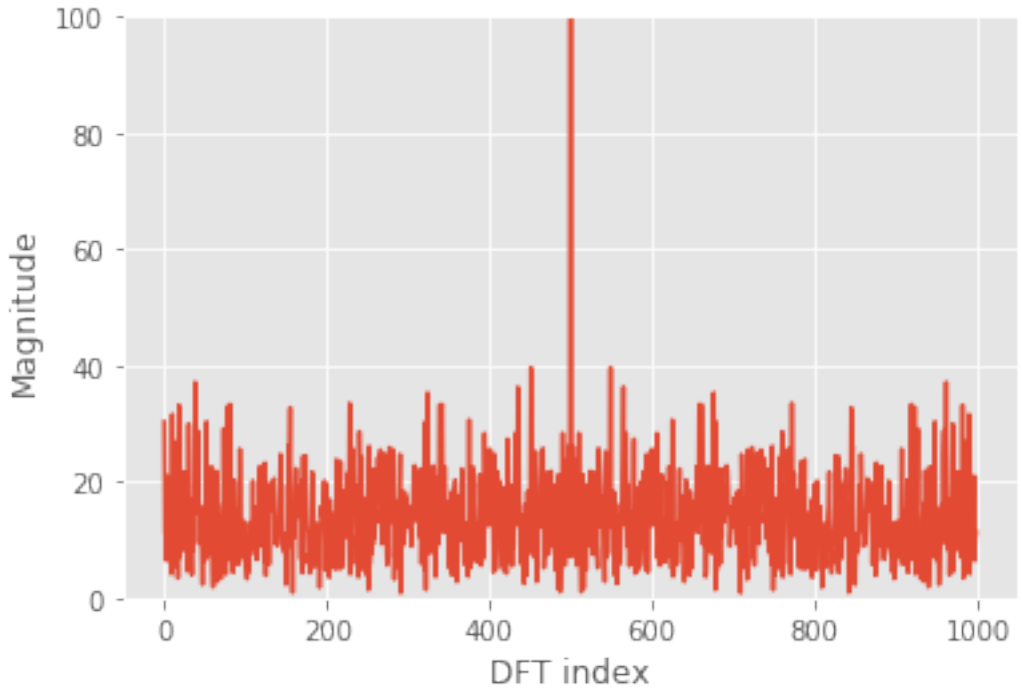


One way to deal with this is to apply a low-pass filter before differentiating. For example, we can replace the term $K_d s$ with $\frac{K_d s}{s/\omega_c+1}$, where ω_c is some cutoff frequency. We can interpret the

factor $(s/\omega_c + 1)^{-1}$ as a first-order low-pass filter connected in series with the regular derivative term. Typically, we assume the frequencies in the reference/output will be much lower than the frequencies that comprise the noise.

[95]:
```
plt.plot(np.abs(np.fft.fftshift(np.fft.fft(step + noise))))
plt.xlabel('DFT index')
plt.ylabel('Magnitude')
plt.ylim(0, 100)  # Ensure the entire spectrum is visible.
```

[95]: (0, 100)



Here, we've plotted the spectrum of the reference input with the noise. With `np.fft.fftshift`, the lowest-frequency component is now at the middle index.

Explain the spike you see above.

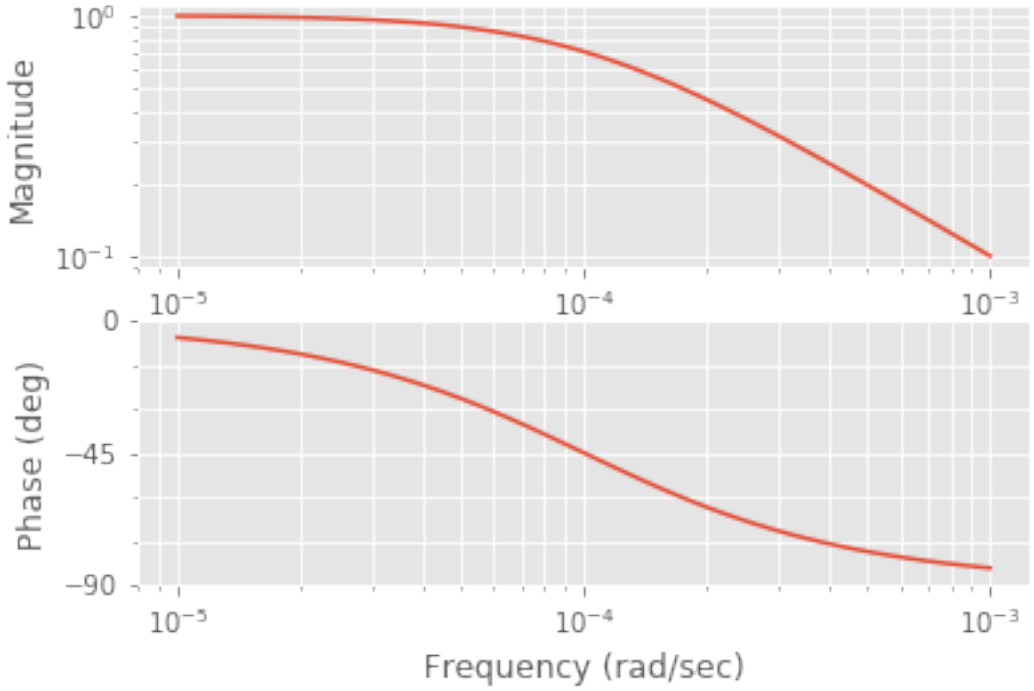
The spike corresponds to the magnitude of the lowest frequency region of the signal, which is the region that corresponds to the true signal.

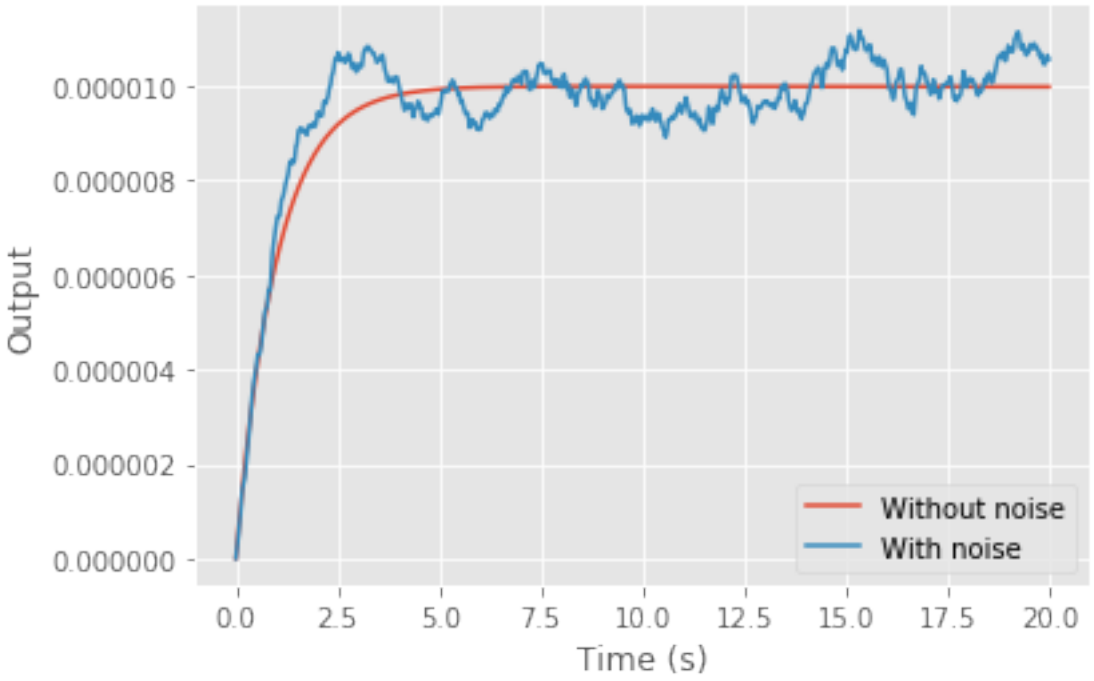
Your task here is to implement `make_filtered_pid_tf`, which is exactly like `make_pid_tf`, but uses the filtered derivative with a value of ω_c you should determine empirically. Remember that you can make ω_c quite low because we're trying to eliminate almost everything you see in the spectrum of the noisy reference signal. Again, you will need to derive a simple rational form of $H(s)$ that can be plugged into `control.tf`.

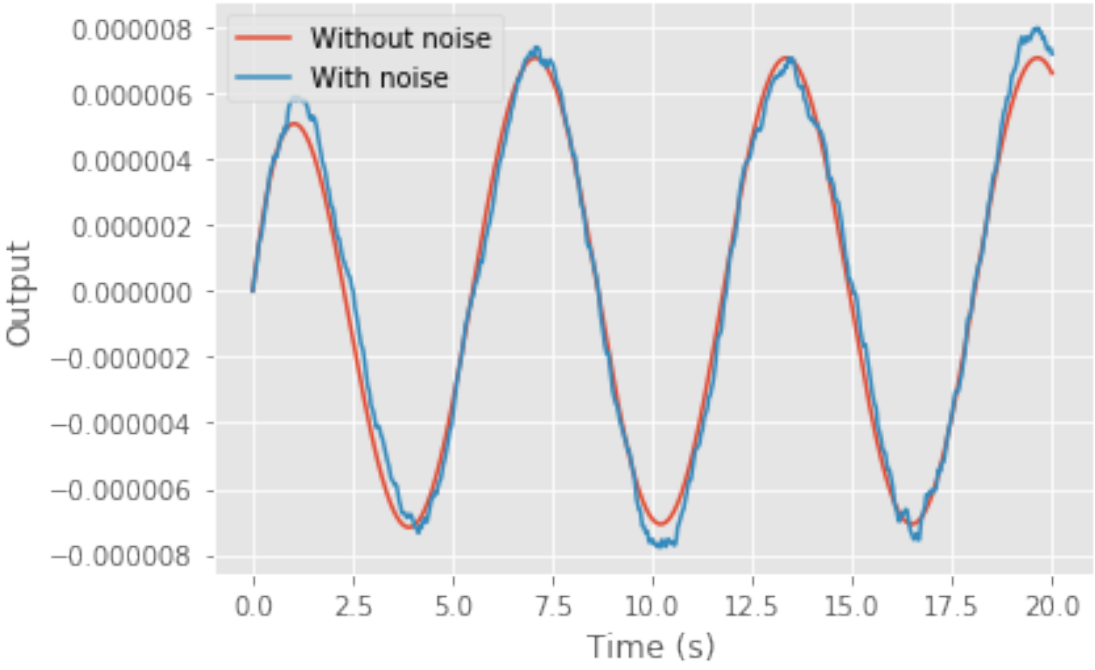
If you've implemented the filter correctly, the response should be cleaned up considerably with some low-frequency sinusoids remaining. Your noiseless step response might have changed slightly

as well.

[96]:
```
wc = 10e-5# TODO: determine a suitable \omega_c

def make_filtered_pid_tf(Kp, Ki, Kd):
    """
    Make the PID controller transfer function H(s) with a first-order
    lowpass filter on the derivative term.
    """
    return control.tf([(Kp/wc) + Kd, Kp + (Ki / wc), Ki], [wc,1,0]) # TODO

control.bode_plot(control.tf([1], [1/wc, 1]))
plt.figure()
cl_sys = make_cl_tf(plant, make_filtered_pid_tf(10, 10, 10)).
 ↪returnScipySignalLTI()[0][0]
plot_response_with_noise(cl_sys, t_in, step, noise)
plt.figure()
plot_response_with_noise(cl_sys, t_in, cos, noise)
```




0.000010
0.000008
0.000006
0.000004
0.000002
0.000000
Output
Without noise
With noise
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time (s)


0.000008
0.000006
0.000004
0.000002
0.000000
−0.000002
−0.000004
−0.000006
−0.000008
Output
Without noise
With noise
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time (s)

1.2.3 Q1(c): Stabilizing an Unstable System

Finally, we'll look at using feedback to stabilize an unstable plant:

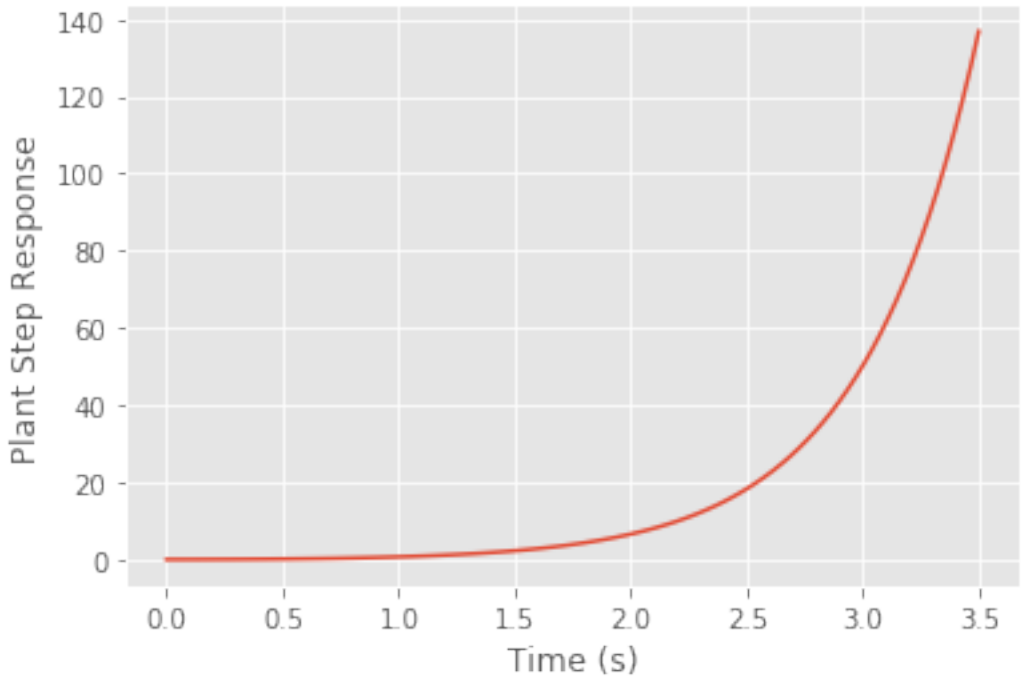
$$P(s) = \frac{1}{(s+2)(s-2)}$$

Assuming the system is causal, the RoC is $\text{Re}(s) > 2$, which does not include the $j\omega$-axis. One naive solution might involve not using feedback at all and cascading the plant with the transfer function $H(s) = s - 2$ to cancel the unstable pole. However, we typically don't know the *exact* placement of the unstable pole, so this method will fail to remove the instability.

The step response of the plant is shown below. As you can see, the pole at $s = 2$ adds an exponential that grows without bound.

[98]:
```
plant = control.tf([1], [1, 0, -4])
plt.plot(*control.step_response(plant)[:2])
plt.xlabel('Time (s)')
plt.ylabel('Plant Step Response')
```

[98]:
```
Text(0, 0.5, 'Plant Step Response')
```

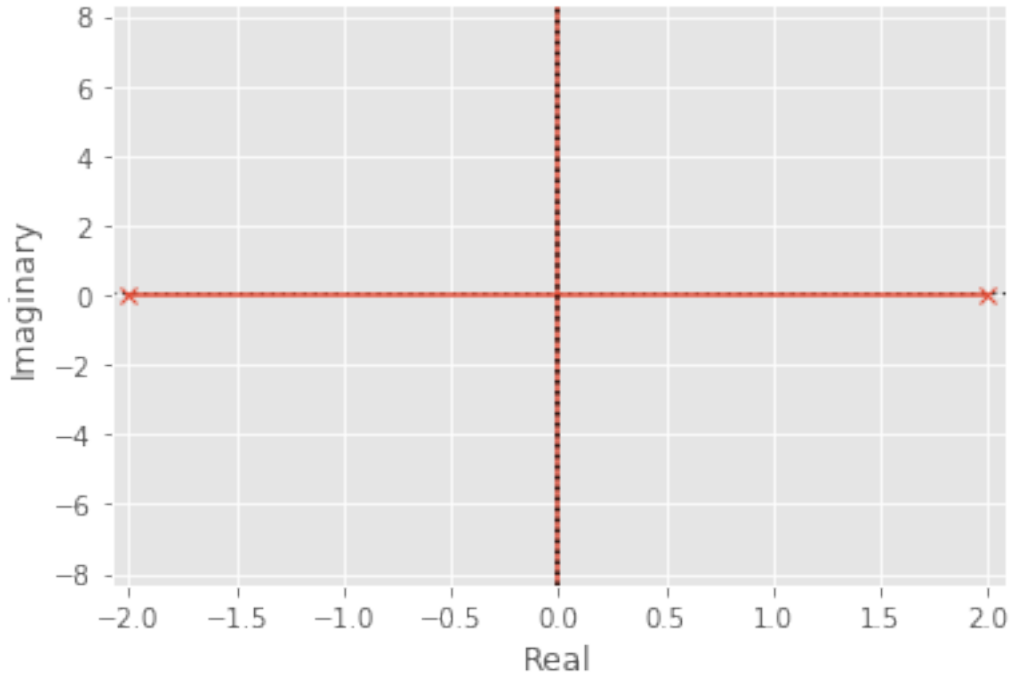


If we plot the root locus of the plant, we'll see how the roots of $1 + K_p P(s) = 0$ move as we tweak K_p.

Can we stabilize the system with constant gain feedback? Why or why not?

No, we will not be able to with constang gain feedback, because any gain K_p will still produce an unstable system as the points lie on the imaginary line or on the right half plane.

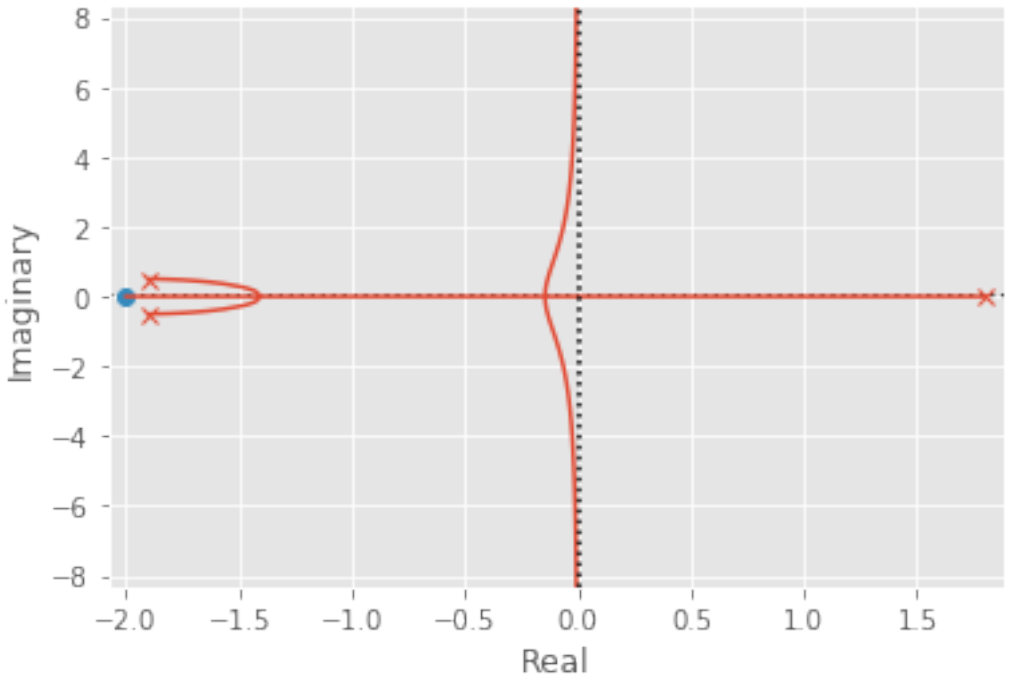
[124]:
```
_ = control.root_locus(plant)
```



We'll need to try something more sophisticated that vanilla PID to move all the closed-loop poles into the left-half plane. Let's try a lead controller (as shown in note 19):

$$H(s) = K\frac{s-\beta}{s-\alpha}$$

[125]:
```
alpha, beta = -2, -1
controller = control.tf([1, -beta], [1, -alpha])
r_locs, gains = control.root_locus(make_cl_tf(plant, controller))
```

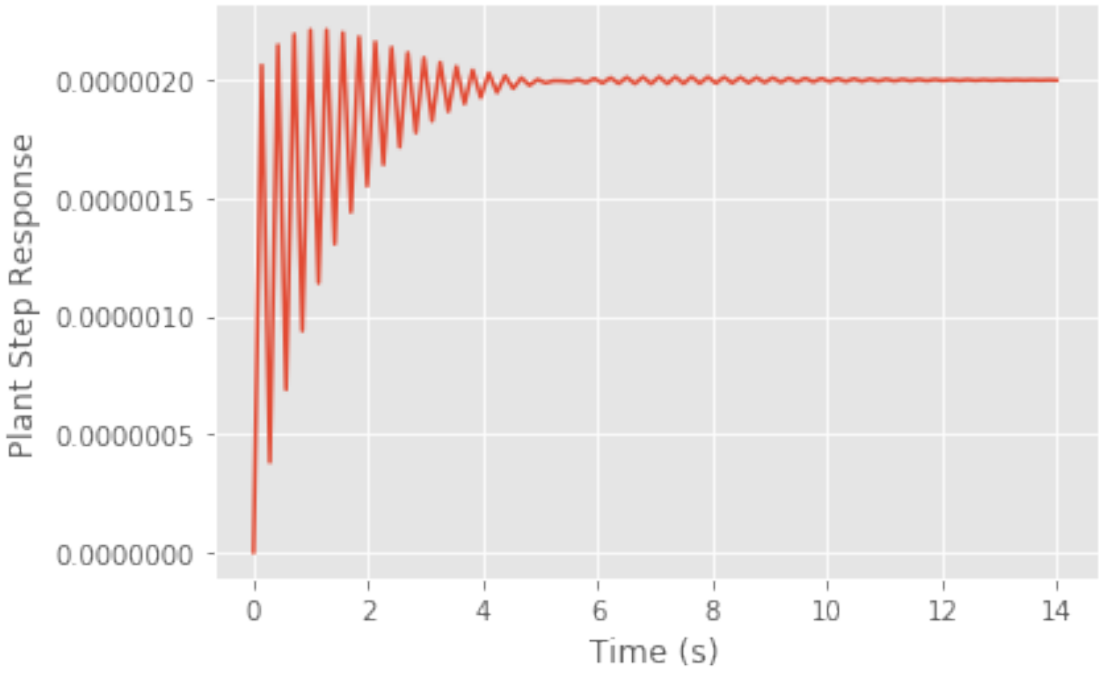


Your task is to search `r_locs` and `gains` for a suitable gain that will stabilize the system. To remind you what `control.root_locus` returns, `r_locs` is a 2D array whose rows are pole locations (since there are three closed-loop poles, every row of `r_locs` is three entries wide). `gains` is a 1D array of candidate values of K that have been sampled by the `control` package. The i^{th} gain in `gains` corresponds to the i^{th} row of closed-loop poles.

Since the goal here is just to stabilize the system, any K that shows the closed-loop system has a stable step response will suffice. In a real system, we might care about rise time, percent overshoot, and other quantities that would impose constraints on pole placement. Note that although some rows of `r_locs` might nominally have all strictly negative parts, the step response might still be unstable due to numerical imprecision. It's safer to pick a larger value of K for which all the poles will be further away from the $j\omega$-axis in the left-half plane.

[126]:
```
# TODO: your code here.
stable_gain = 10e5

stablized_sys = make_cl_tf(plant, stable_gain*controller)
plt.plot(*control.step_response(stablized_sys)[:2])
plt.xlabel('Time (s)')
plt.ylabel('Plant Step Response')
```

[126]:
```
Text(0, 0.5, 'Plant Step Response')
```



1.3 Q2: Inverted Pendulum (Optional)

Now, we'll apply PID to solve a classic problem in control theory: balancing an inverted pendulum mounted on a cart. Unfortunately, we don't have the hardware to show your control scheme in action, but here's a clip of a SpaceX landing performing similar stabilization.

The setup is shown in the schematic above. - The mass of the cart and pole are denoted M and m, respectively. - The length of the pole is ℓ. - The input to the system is a continuous force F. For simplicitly, we won't assume any constraints on F. - The output of the system is the angle θ the pole makes with the vertical, as depicted by the dashed line in the diagram above. - x is the cart's position relative to some fixed origin. - The pole's angle cannot be controlled directly. The only control we have is force, which is related to x and θ by differential equations.

As it turns out, the relationship between F and θ is nonlinear. However, for small θ, the system is well-approximated as an LTI system, meaning the plant has a transfer function and can be controlled with PID.

[127]:
```
# Feel free to ignore this block.

class CartPoleSimulation:
    """
    Simulate an inverted pole on a cart.

    The state of the system is a four-entry vector containing:
      * x: Cart's position.
```

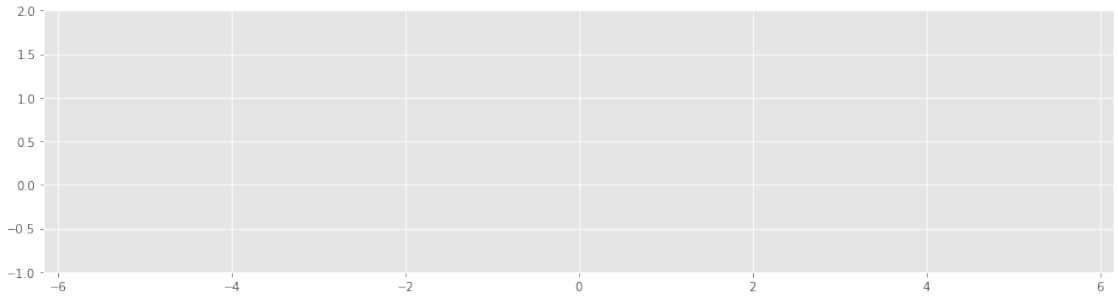
```
    * x_dot: Cart's velocity.
    * theta: Pole's angular position.
    * theta_dot: Pole's angular velocity.
  """
  cart_mass, pole_mass, pole_length, gravity = 1, 0.1, 1, 9.8
  # Just for visualization. These have no effect on the dynamics.
  cart_width, cart_height = 0.5, 0.25

  def __init__(self, initial_state=None, dt=0.02):
      if initial_state is None:
          initial_state = np.zeros(4)
      self.initial_state = self.state = initial_state
      self.dt = dt

  def reset(self):
      self.state = self.initial_state

  def step(self, force):
      """
      Simulate the evolution of the system over a small time step.

      Arguments:
          force (float): The force applied to the cart.

      Returns:
          The state vector after the time step is over.
          The internal state of the simulation is also mutated.
      """
      # These nonlinear dynamics represent the system in full fidelity.
      M, m, l = self.cart_mass, self.pole_mass, self.pole_length
      x, x_dot, theta, theta_dot = self.state
      A = np.array([
          [l**2*m, -l*m*np.cos(theta)],
          [-l*m*np.cos(theta), M + m],
      ])
      velocities = np.linalg.inv(A).dot(np.array([
          l*m*self.gravity*np.sin(theta),
          -l*m*theta_dot*np.sin(theta) + force,
      ]).reshape(2, 1))
      theta_ddot, x_ddot = velocities[0, 0], velocities[1, 0]
      self.state = np.array([
          x + x_dot*self.dt,
          x_dot + x_ddot*self.dt,
          theta + theta_dot*self.dt,
          theta_dot + theta_ddot*self.dt,
      ])
      return self.state
```

```
    def simulate(self, control_input, disturbance=None, steps=500):
        disturbance = disturbance or (lambda _: 0)
        trajectory = np.empty((steps, self.state.shape[0] + 2), dtype=np.
↪float64)
        self.reset()
        for t in range(steps):
            u = control_input(self.state)
            w = disturbance(self.state)
            trajectory[t, :4] = self.step(u + w)
            trajectory[t, 4], trajectory[t, 5] = u, w
        return trajectory

    def animate(self, trajectory):
        figure, ax = plt.subplots(figsize=(16, 4))
        pole = plt.Line2D((0, 0), (0, 0), lw=2)
        cart = plt.Rectangle((0, -self.cart_height),
                             self.cart_width, self.cart_height)
        control = plt.Line2D((0, 0), (0, 0), lw=2, color='#4dd056')
        disturbance = plt.Line2D((0, 0), (0, 0), lw=2, color='#e7724d')
        ax.add_line(pole)
        ax.add_line(control)
        ax.add_line(disturbance)
        ax.add_patch(cart)
        ax.axis('equal')
        ax.set_xlim(-10, 10)
        ax.set_ylim(-1, 2)

        def step_animation(state):
            x, _, theta, _, u, w = state
            pole.set_data([x, x - np.sin(theta)],
                          [0, np.cos(theta)])
            cart.set_x(x - self.cart_width/2)
            control.set_data([x, x + u], [-self.cart_height/2 + 0.05]*2)
            disturbance.set_data([x, x + w], [-self.cart_height/2 - 0.05]*2)
            return (pole, cart)

        return FuncAnimation(figure, step_animation, states, interval=1000*self.
↪dt, blit=True)

class Disturbance:
    def __init__(self):
        self.counter, self.force = 0, 0

    def push(self, state):
        if self.counter == 0:
            self.counter = np.random.randint(10, 20)
```

```
            # Alternate between adding and not adding force.
            if abs(self.force) < 1e-6:
                self.force = np.random.uniform(-0.3, 0.3)
            else:
                self.force = 0
        self.counter -= 1
        return self.force
```

[128]:
```
sim = CartPoleSimulation()
states = sim.simulate(lambda _: 0.5)
HTML(sim.animate(states).to_jshtml())
```

[128]: <IPython.core.display.HTML object>



You should see an animation of the cart-pole system above. The animation might take a minute to compute. * The blue box and red line represent the cart and pole, respectively. * The green line indicates the force applied by the controller. * An orange line would indicate a piecewise constant disturbance, which is another force that is added to the control input before it is fed into the plant.

As shown, the controller just applies a constant force of $+0.5$, which causes the cart to accelerate rightwards and the pole to fall over. Your task will be to implement a controller to keep the pole upright even when disturbances are applied. In particular, the main challenge will involve tuning the PID parameters for θ appropriately (you can ignore x). It's possible to get the pole to stay upright with just PD control. Remember that the reference (desired) signal for θ is identically zero, because that's when the pole is in the upright position.

The derivatives $\dot{x}$ and $\dot{\theta}$ are already automatically computed for you on every time step, and stacked into a state vector `state` $= (x, \dot{x}, \theta, \dot{\theta})$ that's passed into the `control_input` method of `Controller`.

[ ]:
```
class Controller:
    def __init__(self, gains, dt=0.02):
        self.gains, self.dt = np.array(gains), dt
        # In case you want to use integral control, you can
        # approximate the integrated error by adding
        # error*dt to the counter at every time step.
        #
```

```
        # However, empirically, adding an I term doesn't
        # seem to work that well because \theta is highly
        # sensitive to gravity.
        self.x_error = 0
        self.theta_error = 0

    def control_input(self, state):
        """
        Computes the force that should be applied to the cart.
        Called on each time step.

        Arguments:
            state (numpy.ndarray): An array of shape (4,) containing
                x, x_dot, theta, theta_dot at the current timestep.

        Returns:
            u(t) = F(t)
        """
        raise NotImplemented  # TODO: your code here.
```

[ ]:
```
sim.reset()
controller = Controller([0, 0, 0])  # TODO: your theta PID parameters here.
disturbance = Disturbance()
states = sim.simulate(controller.control_input, disturbance.push)
HTML(sim.animate(states).to_jshtml())
```

1.4 Conclusion

- Here, we've presented control for simple single-input single-output (SISO) systems. You can imagine extending this framework to multiple-input multiple-output (MIMO) systems to control multiple state variables at once (for example, controlling both x and θ in the demo above).
- Tuning PID will often depend on the kind of transient and steady-state behavior you're trying to achieve (e.g. overshoot, rise time, etc). EE 128 goes much deeper into how to perform this design, and how the frequency-based techniques you've learned in 120 relate to state-space-based techniques.
- There are many extensions we can consider:
  - Saturation: There are limits to what controls can be applied. For example, an op-amp that's applying a gain will saturate outside of its voltage rails. Here are some ways to deal with saturation and integral windup.
  - Sometimes, the control signal generated might involve switching an actuator on and off at a very high frequency, which might be undesirable because it can wear out a physical system. Not surprisingly, the solution might involve lowpass-filtering the control signal.
  - A control signal might incur considerable cost to apply, so we might want to model finding the control as an optimization problem. Linear Quadratic Regulator (LQR) is one popular technique for operating a system optimally whose cost is quadratic in the

states and inputs (some notes on the discrete-time case). It turns out LQR is easy to solve with dynamic programming.