# MAE 106 Controller Design Assignment Instructions

There is a video introduction to this assignment from Class 20 on the Week 7 page.

This assignment involves the design of a PID feedback controller to stabilize a simulated cart and pendulum system. Each student will submit this assignment individually.

Start by copying these two Python Jupyter Notebooks to your Google Drive:

- https://drive.google.com/file/d/1cTNocjVn7TddnrIovd_oKTLHP2ugmUvL/view?usp=sharing
- https://colab.research.google.com/drive/1qDmukEEEsQUqQ29bEKz5cO_VJzS0Rfxp?usp=sharing

If your browser gives you the option, **open these files in Google Colaboratory**. Open these .ipynb files from your Google Drive **in the browser (Google Chrome recommended).** This will open them using Google Colaboratory, and you can do all of your edits and running code directly in the browser without needing any additional software.

The first notebook is an introduction to linearization, specifically linearizing the nonlinear inverted pendulum on a cart system we will design a controller for. There is nothing to turn in after working through this first notebook.

The second notebook is where you will design and simulate a PID controller for the inverted pendulum on a cart system. Please follow the instructions in the notebook and submit a report with your answers, showing all work. You can use this report template, and then upload your completed document as a pdf.

Post any questions to Ed Discussion!

---

[Below in this PDF are printed versions of the interactive Python Jupyter Notebooks linked above that the students work through: First the introduction to linearization notebook and then the PID controller design notebook.]

This initial code installs and imports necessary packages to run the program.

```
GOOGLE_COLAB = True
```

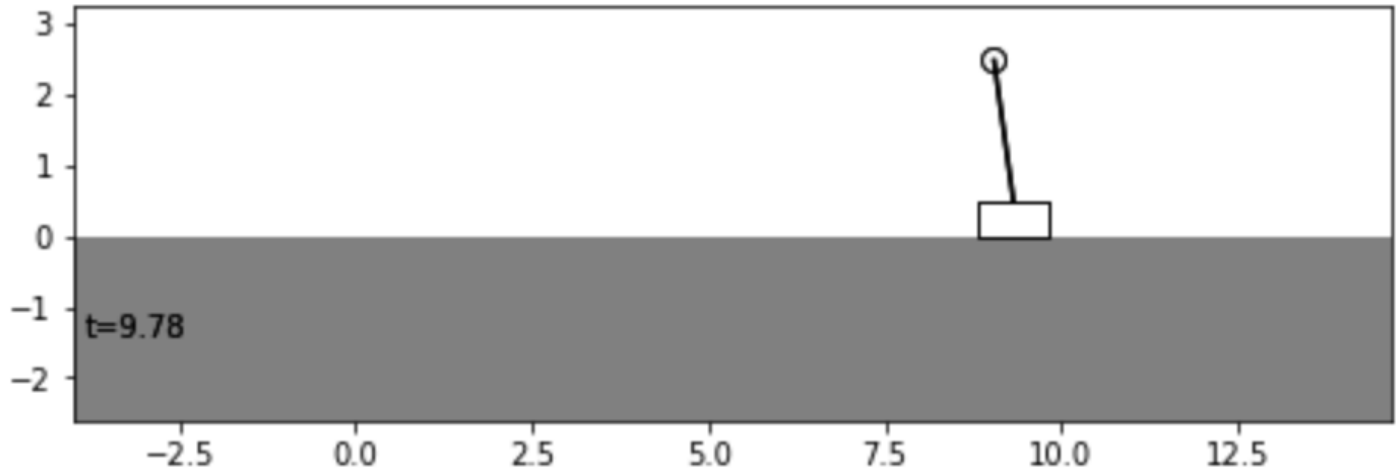
```
if not GOOGLE_COLAB:
    %cd ../
else:
    !pip install git+https://github.com/rland93/pendsim.git
```

```
from pendsim import sim, controller, viz, utils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from IPython.display import HTML
```

# Linearization Notebook

This notebook explores linearization: what does it mean to linearize a nonlinear system?

For this purpose, we explore an inverted pendulum on a cart system:




Imagine the rectangle is a cart that can move horizontally, and the pendulum is free to rotate about the point that it is attached to the cart. For simplicity, let's neglect friction and inertia. This system can be modeled using nonlinear equations of motion.

The system has the following parameters:

- $M$ = mass of the cart
- $m$ = mass on the end of the pendulum
- $l$ = length of the pendulum

- $g$ = gravity

The system's states that vary with time are:

- $x$ = cart position
- $\dot{x}$ = cart velocity
- $\theta$ = pendulum angle (where $\theta = 0$ corresponds to the upright position)
- $\dot{\theta}$ = pendulum angular velocity

And some horizontal force or "push" of the cart is given by $u$. Then the equations of motion for this system are:

$$(M + m)\ddot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2 \sin\theta = u$$
$$ml^2\ddot{\theta} + mgl\sin\theta = -ml\ddot{x}\cos\theta$$

This is a system of nonlinear 2nd-order differential equations in terms of $x$ and $\theta$. It is highly nonlinear: we have several sines and cosines and some quadratic terms.

We can easily produce, however, a simplified, linear approximation of the system that behaves similar to the nonlinear system in a neighborhood around a given point.

To do so, we will use techniques familiar to us from calculus.

Let's take a nonlinear function, $f(x) = sin(x)$, as an example. This looks squiggly:

```
x = np.linspace(0,8,500)
y = np.sin(x)
plt.plot(x, y)
```

We know from calculus that we can take the derivative of this function and write down its Taylor Series around any point $a$ that we want. Let's choose the point $a = 1$.

To find the Taylor Series about $a = 1$, we evaluate the function's derivative at the point of interest and use that to find the series terms:

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 \ldots$$

Now, this series continues forever, and it's also nonlinear. As we continue to add terms to the summation, it approaches our original function $f(x) = sin(x)$.

```
# helper function for drawing taylor series
def taylor_plot(a, x):
    a = 1
    y = np.sin(x)
    y1 = np.repeat(np.sin(a), y.shape)
    y2 = y1 + np.cos(a) * (x-a)
```

```
    y3 = y2 + np.sin(a) * (1/2.) * (x-a)**2
    y4 = y3 + np.cos(a) * (1/6.) * (x-a)**3
    y5 = y4 + np.sin(a) * (1/24.) * (x-a)**2
    plt.axvline(a, linestyle=":", color="#555")
    plt.plot(x, y, 'k')
    plt.plot(x, y1, color="#333")
    plt.plot(x, y2, color="#666")
    plt.plot(x, y3, color="#888")
    plt.plot(x, y4, color="#999")
    plt.plot(x, y4, color="#aaa")
```

We can look at this with our function $y = sin(x)$:

```
x = np.linspace(0,8,500)
taylor_plot(1, x)
```

We can see that our Taylor Series approximation diverges a lot from the nonlinear function as we get farther away from $a$; let's look a little bit closer:

```
x = np.linspace(0, 2, 500)
taylor_plot(1, x)
```

Here, we notice a few things. First, that the 0th order Taylor Series approximation isn't much use; it's just the function output at the point $a$. Second, as we add more terms, we achieve a diminishing return as far as how well the function gets approximated. The 2nd, 3rd, and 4th order terms are all very close to the 1st order term as far as accuracy, in a region close enough to the point. So let's look at just the first-order approximation:

```
a = 1
# the actual function
y = np.sin(x)
# 1st order taylor approx.
y1 = np.sin(a) + np.cos(a) * (x - a)

plt.plot(x, y, 'k')
plt.plot(x, y1, '#888')
```

This is the power of the linear approximation. For a function $f(x)$, if we can find its derivative, we can produce a "pretty good" approximation for it around any point we want. Most importantly, that first-order approximation is a *linear* approximation. We notice our `y1` function above

$$y_1 = \sin(a) + \cos(a)(x - a)$$

has the same form as the familiar line equation

$$y_1 = c_1 x + c_2$$

We can extend this line of thinking to the original pendulum/cart system; the linearization method here works for *any* function, no matter how complex, so long as it is differentiable. It's a little bit of effort to take the derivative of the vector-valued system of ODEs we have above, so let's review it briefly here.

We represent the pendulum state as a vector: $\mathbf{x} = [x, \dot{x}, \theta, \dot{\theta}]$. Here, bold $\mathbf{x}$ represents the entire system, not just the cart position. With some rearranging, the original equations of motion can be written in terms of $\ddot{x}$ and $\ddot{\theta}$. The equations of motion can be written in vector form:

$$[\dot{x}, \ddot{x}, \dot{\theta}, \ddot{\theta}] = f([x, \dot{x}, \theta, \dot{\theta}])$$

$$\dot{\mathbf{x}} = f(\mathbf{x})$$

In this form, the derivative of the function $f$ is the Jacobian matrix of partial derivatives:

$$\frac{\partial \mathbf{x}}{\partial t} f(\mathbf{x})$$

Then, the first-order Taylor Series approximation of the function is equal to

$$f_{linear}(x) = f(a) + \frac{\partial \mathbf{x}}{\partial t} f(a)(x - a)$$

This approximation is often written in "state-space" form as

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

Now we can take steps towards our goal of finding a linear approximation for our nonlinear system. For a given state $\mathbf{x} = a$, we want to find matrices $A$ and $B$ by computing the partial derivatives $\frac{\partial \mathbf{x}}{\partial t} f(a)$ and rearranging the terms of the first-order Taylor Series approximation.

Which state will we linearize about?

In general, we might choose and equilibrium point (or the set point of a controller) for the linearization. Then, a well-designed controller will keep the state within the region surrounding the setpoint, and then, the linearized system model will accurately represent of the system state.

We choose $a = [0, 0, 0, 0]$, which corresponds to the upright position and zero velocity.

After finding the derivative of our vector-valued function and rearranging some terms, we arrive at the linearized model of our original equation.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g(M+m)}{Ml} & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ \frac{1}{Ml} \end{bmatrix}$$

How good is the linear approximation? In order to see, we can compare the true state with the state that is predicted by the linear model. The following code gives us the linear prediction.

```
class Linearization_Measurement(controller.Controller):
    def __init__(self, pend, dt) -> None:
        self.pend = pend
        self.A, self.B = self.get_linear_sys(pend.jacA, pend.jacB, dt)
        self.x_1 = np.array([0,0,0.01,0])

    def policy(self, state: np.ndarray, dt: float):
        # get error from previous state
        self.err = state - self.x_1


        # predict next state
        self.x_1 = self.A @ state + self.B @ np.array([0])
        action = 0

        # store variables
        data = {}
        data.update(utils.array_to_kv("x_1", controller.LABELS, self.x_1))
        data.update(utils.array_to_kv("pred_err", controller.LABELS, self.err))
        return action, data
```

Now, let's set up a simulation of the nonlinear system to investigate how well the linear model predicts the state.

```
dt, t_final = 0.01, 10
# start slightly tilted so that the pendulum will fall over
pend = sim.Pendulum(2, 2, 2, initial_state=np.array([0,0,0.01,0]))
cont = Linearization_Measurement(pend, dt)
simu = sim.Simulation(dt, t_final, lambda t: 0)
results = simu.simulate(pend, cont)
```

We can see how good our linearization is by looking at the difference between what was *predicted* to happen by our linearized model and what *actually* happened in the simulation.

We should notice two things. First, that at the linearization point, the difference should be very small, maybe zero. This makes sense from our plots of $sin(x)$ above. At the point $a$, the linearization produces exactly the function output.

Second, we should notice that near the point $a$, the linearization produces a pretty good approximation of the underlying function. Remember, we might use this technique for functions that are far more complex than $sin(x)$ -- like our pendulum dynamics! And, given that we are only doing a first-order approximation, far away from the point around which we linearized our model, the linear function might not be so great an approximation.

Now we can look at the error between what the linear model predicted and what the real state was. Let's do that graphically.

```
fig, ax = plt.subplots(nrows=2, sharex=True)
ax[0].plot(results[("pred_err", "t")].abs())
ax[0].set_title("Prediction error over time")
ax[0].set_ylabel("Error (rad)")
ax[1].plot(results[("state", "t")])
ax[1].set_title("Theta over time")
ax[1].set_ylabel("Theta (rad)")
ax[1].set_xlabel("Time (s)")
```

From the graph, we can see pretty clearly that our approximation is good near the linearization point and is less accurate as we move away from it. The linearization was performed where $\theta = 0$. In our simulation, this occurred at $t = 0$s and at approximately $t = 10$s.

On top, we have the difference between the linear model and the actual system. On the bottom, we have the system state.

Just like our Taylor Series for $\sin(x)$ above -- as we get further away from the linearization point, our 1st-order model becomes less accurate.

Finally, let's look at one more plot. Here, we plot $\theta$ on the horizontal axis and the prediction error on the vertical axis.

```
fig, ax = plt.subplots()
ax.plot(results[("state", "t")].shift(1), results[("pred_err", "t")].abs())
ax.set_title(r"Prediction error ($\theta$) as a function of system state ($\theta$)")
ax.set_xlabel(r"$\theta$ (rad)")
ax.set_ylabel(r"Absolute Error (rad)")
```

This graph shows the same result. Near the linearization point of $\theta = 0$, the error is $0$. As we get farther away from $\theta = 0$, our approximation is less accurate.

So, if we design a controller that keeps the value of $\theta$ near the desired point $0$, our linearized system model will be a good approximation of the nonlinear system!

This initial code installs and imports necessary packages to run the program.

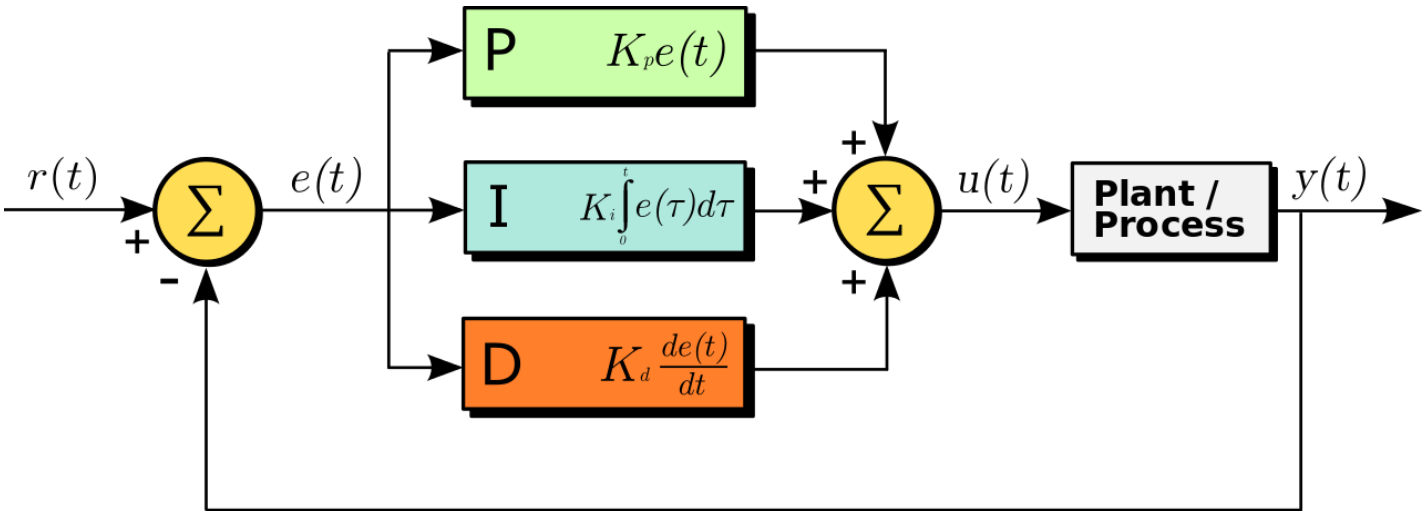
```
GOOGLE_COLAB = True
```

```
if not GOOGLE_COLAB:
    %cd ../
else:
    !pip install git+https://github.com/rland93/pendsim.git
```

```
from pendsim import sim, controller, viz
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML
```

# PID Notebook

PID, or proportional-integral-derivative, control is a model-free feedback policy which uses an error signal and tunable gains to compute a control action that produces a desired system response.




The block diagram of the PID system in the time domain is shown above. Given the error signal $e(t)$, we tune the three gains, $K_p$, $K_i$ and $K_d$, to generate the control signal, $u(t)$. The control signal is an input to the physical system (the *plant / process*), which finally produces some output $y(t)$. (Graphic is sourced from Wikipedia).

For more information about PID control, see Feedback Systems: An Introduction for Scientists and Engineers, Ch 10, or for a less technical introduction, Wikipedia.

---

# Part 1: System Analysis

Consider the inverted pendulum on a cart system. The cart can move horizontally, and the pendulum is free to rotate about the point that it is attached to the cart. For simplicity, let's neglect friction and inertia.

The system has the following parameters:

- $M$ = mass of the cart
- $m$ = mass on the end of the pendulum
- $l$ = length of the pendulum
- $g$ = gravity

The system's states that vary with time are:

- $x$ = cart position
- $\dot{x}$ = cart velocity
- $\theta$ = pendulum angle (where $\theta = 0$ corresponds to the upright position)
- $\dot{\theta}$ = pendulum angular velocity

And some horizontal force or "push" of the cart is given by $u$.

The linearized system can be modeled in the time domain as

$$\dot{\mathbf{x}} = A\mathbf{x} + Bu$$

Written out, this is

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g(M+m)}{Ml} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ \frac{1}{Ml} \end{bmatrix} u$$

Assuming we can measure the angle $\theta$, our goal is to design a proportional-derivative (PD) controller that can stabilize an inverted pendulum on a cart in the upright position (i.e., $\theta = 0$) when it is released from rest at a certain angle different from the upright position (i.e., $\theta \neq 0$).

Therefore, we will consider a control law of the form

$$u(t) = k_p e(t) + k_d \dot{e}(t)$$

where $e(t) = \theta_d(t) - \theta(t)$ is the error signal that is equal to the difference between the desired angle ($\theta_d = 0$ in the upright position) and the measured angle $\theta$.

In the code below, define your system. Choose values for the mass of the cart (choose something between 0.1kg and 10kg), the mass on the end of the pendulum (choose something between 0.1kg and 10kg), and the length of the pendulum (choose something between 0.1 and 3 meters).

## Include the following in your report:

- Write down the closed-loop transfer function relating the output $\theta$ to the input $\theta_d$.
- Write down the poles of the closed-loop system as a function of the control gains and model parameters.
- What is the condition on $k_p$ such that the poles are complex (i.e., for what values of $k_p$ will the poles have non-zero imaginary parts)?
- Write down the parameter values you chose.

```
pend = sim.Pendulum(
    1.0, # Pendulum base mass [kg] - CHANGE THIS VALUE
    1.0, # Pendulum ball mass [kg] - CHANGE THIS VALUE
    1.0, # Pendulum length [m] - CHANGE THIS VALUE

    # state = [x [m], xdot [m/s], theta [rad], thetadot [rad/s]]
    initial_state = np.array([0.0, 0.0, 0.1, 0.0])
    # MAY CHANGE THIRD VALUE IN THIS ARRAY TO A DIFFERENT SMALL VALUE.
    # YOU WILL CHANGE THIS VALUE AGAIN AFTER TUNING A PD CONTROLLER.
)

dt, t_final = 0.01, 10 # time step and simulation time [s]
# define the force that is applied to the cart [N] - don't worry about this function
def force_func(t):
    return 0
simu10 = sim.Simulation(dt, t_final, force_func)
```

---

# Part 2: Controller Design

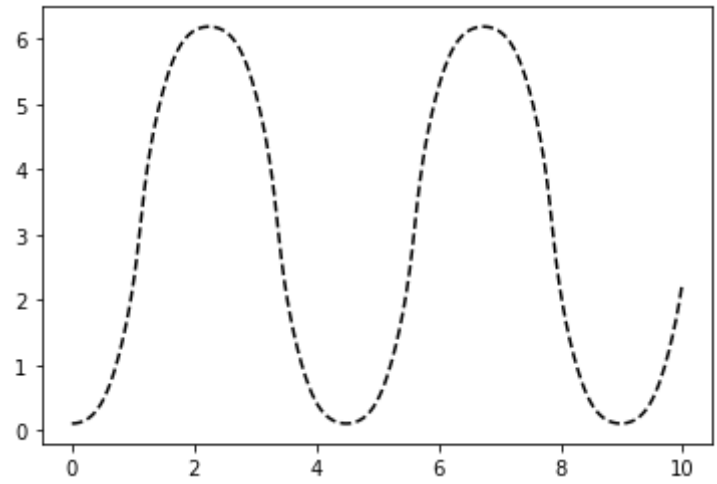
A PID controller has three gains, that we call `kp`, `ki`, `kd` here. These are also called the proportional, integral, and derivative gains, respectively.

To start, set all the gains equal to zero. This zero-gain controller fails to stabilize the pendulum system because all of the coefficients in front of the actuation are 0! So, with all gains equal to 0, the controller takes no action at all. Therefore, $\theta$ increases, and the pendulum simply falls over. The plot shows $\theta$ changing from 0 to $2\pi$ radians as the pendulum swings around and around since there is no friction or inertia.

```
kp, ki, kd = 0.0, 0.0, 0.0 # control gain values
cont = controller.PID((kp, ki, kd))
results = simu10.simulate(pend, cont)
fig1, ax1 = plt.subplots()
ax1.plot(results[('state','t')], 'k--', label='theta')

plt.show()
```

```
100%|██████████| 1000/1000 [00:01<00:00, 926.39it/s]
```




And for a visual confirmation of our suspicion, we visualize the virtual experiment:

```
visu = viz.Visualizer(results, pend, dt)
ani = visu.animate()
HTML(ani.to_html5_video())
```

To tune the PID controller, start by changing the proportional gain, called `kp`. Start at zero and slowly increase it to see what happens. Let's create several controllers, increasing the gain $k_p$ for each controller. Feel free to change this in the code below. Our ultimate goal is to stabilize the pendulum system in the upright position, so that means we want $\theta$ to settle at 0 radians.

```
# starting gain
kp = 0.0
# number of times to increase the gain
n = 32
# amount to increase by
increase_by = 1.5 # YOU CAN CHANGE THIS
# empty lists
conts = []
pends = [pend] * n
gains = []
for _ in range(n):
    # increase the gain
    kp += increase_by
    # set ki, kd to 0
    pid = kp, 0.0, 0.0
    conts.append(controller.PID(pid))
    gains.append(kp)
# simulate each controller
all_results = simu10.simulate_multiple(pends, conts)
```

Now, let's simulate the system and plot the values of $\theta$ for each gain.

```
nrows, ncols = 8, 4
fig1, ax1 = plt.subplots(nrows=nrows, ncols=ncols, sharex=True, sharey=True, figsize=(
axn, ax_idxs = 0, {}
# index helper map for plots
for i in range(nrows):
    for j in range(ncols):
        ax_idxs[axn] = (i, j)
        axn += 1
# create figures and set the title as the gain
for g, (idx, res), (axi, axj) in zip(gains, all_results.groupby(level=0), ax_idxs.valu
    res.index = res.index.droplevel(0)
    ax1[axi, axj].plot(res[('state', 't')])
    ax1[axi, axj].set_title('gain=' + str(g))
# label figures
for i in range(nrows):
    ax1[i, 0].set_ylabel('theta (rad)')
for j in range(ncols):
    ax1[-1, j].set_xlabel('time (s)')
plt.show()
```

We can see that the controller keeps the pendulum's angle close to zero with a large enough $k_p$. Set $k_p$ to this value in the code below.

These results should make intuitive sense with a bit of control systems knowledge: increasing the proportional gain makes the system respond faster and also increases the frequency of oscillation.

The pendulum is (somewhat) stable around $\theta = 0$, but we still have a pesky oscillation. Can we remove it?

There is one more thing going on here. It appears that the system is oscillating but also that the oscillations are increasing in magnitude. Eventually, this instability will compound enough to topple the pendulum, despite having a proportional controller in place with a suitable gain. To see this, we can increase the simulation time, this time to 40 seconds instead of 10.

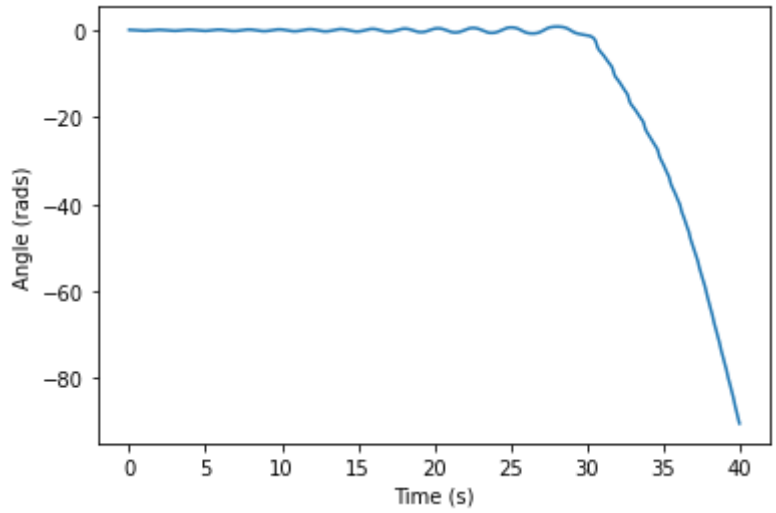
```
kp = 30.0 # CHANGE THIS TO A REASONABLE VALUE YOU FOUND WHEN TUNING ABOVE
cont = controller.PID( (kp, 0, 0) )
simu30 = sim.Simulation(dt, 40, force_func)
res_proportional = simu30.simulate(pend, cont)
```

```
100%|██████████| 4000/4000 [00:01<00:00, 2890.12it/s]
```

And we plot our longer-term simulation:

```
_, ax = plt.subplots()
ax.plot(res_proportional[('state', 't')])
```

```
ax.set_ylabel("Angle (rads)")
ax.set_xlabel("Time (s)")
plt.show()
```




We see that eventually the oscillation grows large enough that the inverted pendulum falls over. The controller pushes a little bit too hard right, and the pendulum overshoots a bit to the left; then, the controller pushes left, and the pendulum overshoots a bit (more!) to the right, and the process continues until the pendulum tips over.

This looks like a response associated with a complex pole with positive real part: a steady oscillation bounded by an envelope of an increasing exponential -- that is, until the pendulum falls over. This is due to the non-zero initial angle.

We need damping! A derivative term seeks to drive the rate of change of the error closer to zero over time. Practically, this means that it can counteract the magnitude of the steady oscillation we see (i.e., add damping to the system), and as a result, stabilize the system.

Below, we follow the same experimental process for tuning: slowly increase $k_d$ and see the effects. In real-life systems, where failure can have expensive or dangerous consequences, tuning is a very delicate process, typically informed heavily by process knowledge. In this simulated world, we have no such concerns.

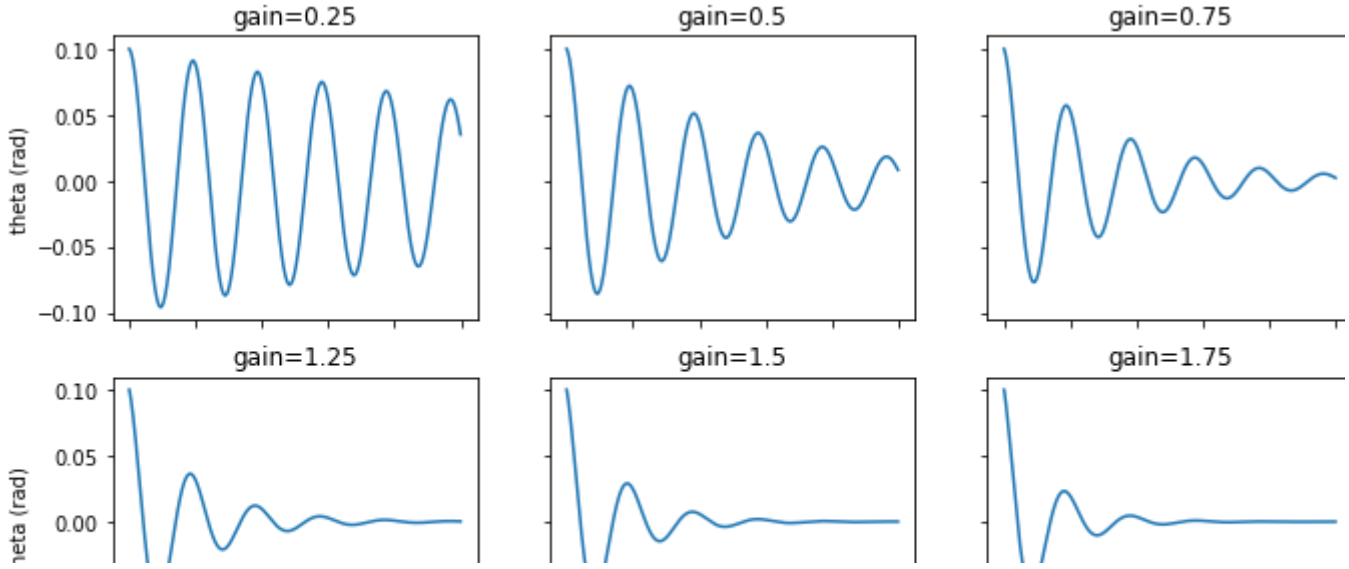
```
kp = 30.0 # CHANGE THIS TO A REASONABLE VALUE YOU FOUND WHEN TUNING ABOVE
kd = 0.0
n = 16
increase_by = 0.25 # YOU CAN CHANGE THIS
conts = []
pends = [pend] * n
gains = []
for _ in range(n):
```

```
        # increase the gain
        kd += increase_by
        # set ki, kd to 0
        pid = kp, 0.0, kd
        conts.append(controller.PID(pid))
        gains.append(kd)
    # simulate each controller
    all_results = simu10.simulate_multiple(pends, conts)

    nrows, ncols = 4, 4
    fig1, ax1 = plt.subplots(nrows=nrows, ncols=ncols, sharex=True, sharey=True, figsize=(
    axn, ax_idxs = 0, {}
    for i in range(nrows):
        for j in range(ncols):
            ax_idxs[axn] = (i, j)
            axn += 1
    for g, (idx, res), (axi, axj) in zip(gains, all_results.groupby(level=0), ax_idxs.valu
        res.index = res.index.droplevel(0)
        ax1[axi, axj].plot(res[('state', 't')])
        ax1[axi, axj].set_title('gain=' + str(g))
    # label plots
    for i in range(nrows):
        ax1[i, 0].set_ylabel('theta (rad)')
    for j in range(ncols):
        ax1[-1, j].set_xlabel('time (s)')

    plt.show()
```

```
100%|██████████| 1000/1000 [00:00<00:00, 1583.46it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1572.35it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1684.75it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1586.66it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1724.68it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1555.73it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1694.20it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1611.31it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1725.43it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1527.42it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1677.85it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1558.19it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1586.41it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1691.98it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1394.83it/s]
100%|██████████| 1000/1000 [00:00<00:00, 1213.46it/s]
```

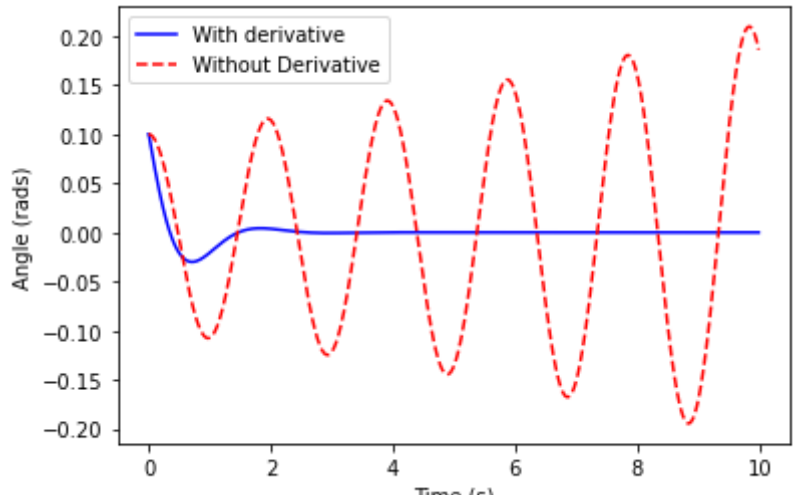



Now, we look again at $\theta$. It looks significantly better! The addition of a derivative term drives the oscillations towards zero. To see the effect, we plot the two controllers next to one another, one with the $k_d$ term and one without.

```
res_p_10 = simu10.simulate(pend, controller.PID((30, 0.0, 0.0)))
# Arguments in controller.PID are PID gains (kp, ki, kd)
# CHANGE THESE GAINS TO THOSE THAT YOU EXPERIMENTALLY FOUND TO WORK WELL.
# CHANGE ONLY kp ABOVE, BUT CHANGE BOTH kp AND kd BELOW.
res_pd_10 = simu10.simulate(pend, controller.PID((30, 0.0, 3.5)))

fig, ax = plt.subplots()
ax.plot(res_pd_10[('state', 't')], 'b-', label='With derivative')
ax.plot(res_p_10[('state', 't')], 'r--', label='Without Derivative')
ax.set_ylabel('Angle (rads)')
ax.set_xlabel('Time (s)')
ax.legend()
plt.show()
```

```
100%|██████████| 1000/1000 [00:00<00:00, 2973.28it/s]
100%|██████████| 1000/1000 [00:00<00:00, 2901.93it/s]
```




The difference is stark!

Finally, we can look at a visualization of how this controller performed.

```
visu = viz.Visualizer(res_pd_10, pend, dt)
ani = visu.animate(blit=True)
HTML(ani.to_html5_video())
```

Since this simulation is relatively accurate, we expect that a pendulum with the same attributes (length, mass, etc.), if controlled by a controller with the same gain (as expressed in Newtons of force applied), would have similar stability characteristics -- so long as we could accurately measure the state!

Tuning a controller in simulation is common practice before implementing a controller in a physical system, where the consequences of bad tuning can be disasterous. If there were serious consequences to knocking over our pendulum, we would want to use the gains we have discovered here as a starting point.

## Include the following in your report:

- Write down the final $k_p$ and $k_d$ values you chose that stabilize the system.
- Given these gain values and the parameter values you chose, calculate the values of the poles of your closed-loop system and plot them on the complex plane. Drawing the figure by hand is fine.
- Is this system stable, marginally stable, or unstable? Is the system underdamped, critically damped, or overdamped?
- Include the figure generated above that shows the comparison between responses with a P controller versus a PD controller.

- Would adding an integral term to the control law (i.e., make it a PID controller rather than a PD controller) improve the performance of this closed-loop system? Describe why or why not. Feel free to experiment with $k_i$ values in the code.
- Finally, change the initial condition for $\theta$ in Part 1 above. What is the largest initial angle for which this same PD controller you already tuned can stabilize the system? (You only need to update the initial condition in the code that defines 'pend' in Part 1, rerun that code, and rerun the code at the end of Part 2 that compares the P and PD controllers.)
- Include the figure generated above comparing the P and PD controllers for this new initial condition.