# Assignment 5 - Parameter Identification

ShivakumarSridhar (1234846225)

## Servo Data Collection

### Part 1: The Experiment

1. I flashed the ESP32 using the Arduino IDE and crafted a simple C++ program utilizing the ESP32Servo library to alternate the servo between 0 and 180 degrees every second.

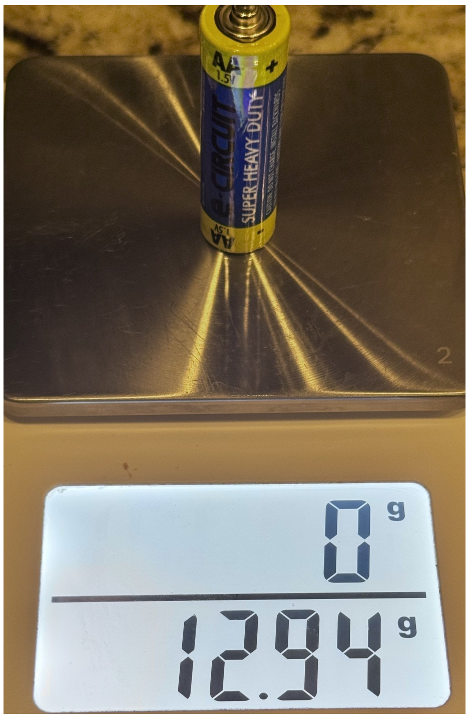
```
#include <ESP32Servo.h>

Servo myServo;  // Create servo object

// Use Pin 14 to match your MicroPython code
int servoPin = 14;

void setup() {
  // Attach the servo to the pin
  myServo.attach(servoPin);
}

void loop() {
  // Move to 0 degrees
  myServo.write(0);
  delay(1000); // Wait for 1 second (1000 milliseconds)

  // Move to 180 degrees
  myServo.write(180);
  delay(1000); // Wait for 1 second
}
```

2. I then plugged in my ESP32 to a half-size breadboard and wired up my servo. I hung the end of the servo off the edge of a desk and weighted it down so it wouldn’t move.

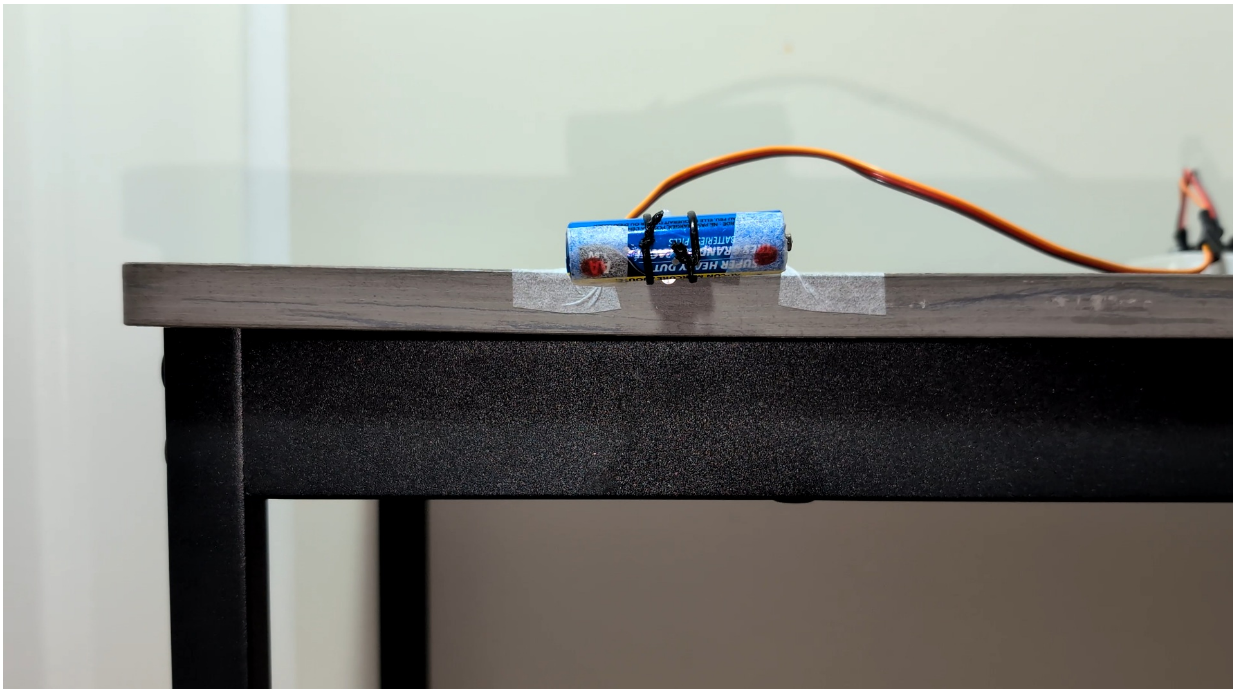
3. I wrapped a wire around my AA battery to rigidly attach it to the servo’s horn. I mounted the battery so it was mounted symmetrically about the rotational center of the servo.

4. I weighed the battery

In [53]:
```
from IPython.display import Image, display
display(Image("battery.jpg", width=300))
```



In [54]:
```python
from IPython.display import Image, display
display(Image("baresetup.png", width=800))
```

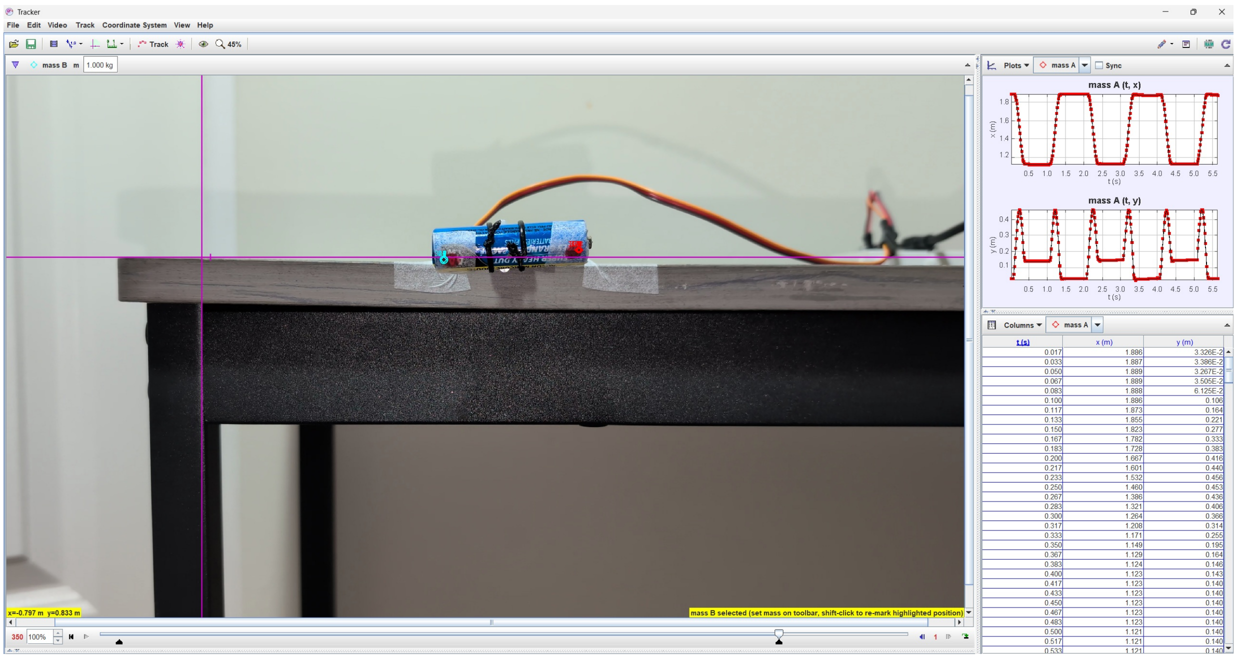


5. I used a tripod to hold my cell phone approximately 12 inches away from my scene. I made sure my servo was in the center of my frame, that my camera was perpendicular to the plane of my experiment.

6. I took several practice videos and checked them to ensure the camera didn't move, the base of the servo didn't shake or move, and that my scene was well-lit and in focus.

## Part 2: Image Tracking

In [55]:
```python
from IPython.display import Image, display
display(Image("trackingsetup.png", width=800))
```



x=-0.797 m y=0.833 m

mass B selected (set mass on toolbar, shift-click to re-mark highlighted position)

| t (s) | x (m) | y (m) |
|---|---|---|
| 0.017 | 1.886 | 3.326E-2 |
| 0.033 | 1.887 | 3.386E-2 |
| 0.050 | 1.889 | 3.267E-2 |
| 0.067 | 1.889 | 3.505E-2 |
| 0.083 | 1.888 | 6.125E-2 |
| 0.100 | 1.886 | 0.106 |
| 0.117 | 1.873 | 0.164 |
| 0.133 | 1.855 | 0.221 |
| 0.150 | 1.823 | 0.277 |
| 0.167 | 1.782 | 0.333 |
| 0.183 | 1.728 | 0.383 |
| 0.200 | 1.667 | 0.416 |
| 0.217 | 1.601 | 0.440 |
| 0.233 | 1.532 | 0.456 |
| 0.250 | 1.460 | 0.453 |
| 0.267 | 1.386 | 0.436 |
| 0.283 | 1.321 | 0.406 |
| 0.300 | 1.264 | 0.366 |
| 0.317 | 1.208 | 0.314 |
| 0.333 | 1.171 | 0.255 |
| 0.350 | 1.149 | 0.195 |
| 0.367 | 1.129 | 0.164 |
| 0.383 | 1.124 | 0.146 |
| 0.400 | 1.123 | 0.143 |
| 0.417 | 1.123 | 0.140 |
| 0.433 | 1.123 | 0.140 |
| 0.450 | 1.123 | 0.140 |
| 0.467 | 1.123 | 0.140 |
| 0.483 | 1.123 | 0.140 |
| 0.500 | 1.121 | 0.140 |
| 0.517 | 1.121 | 0.140 |
| 0.533 | 1.121 | 0.140 |

1. I installed tracker on a Windows computer and imported my video. I identiﬁed several high-contrast points that remained visible even during the blurry bits when the servo was rotating.
2. I used the edge of the desk as my x-axis reference line, to ensure the x-y coordinate system matched up with a planar model with gravity in the -y direction.
3. I output the data as trackpoints.csv

## Part 3: Calculating the Data

First, let's import some necessary packages in python

```
In [56]: import pandas
         import scipy.signal as ss
         import scipy.optimize as so
         import numpy
         import matplotlib.pyplot as plt
         import math
```

Next, we import data collected by tracker and identify the useful columns

```
In [57]: data = pandas.read_csv('trackpoint25_1_2.csv')
         data.columns
         data.iloc[:10, :].head()
```

```
Out[57]:
        t       x       y     x.1     y.1
0  0.0000    1.88  0.0307    1.21  -0.014
1  0.0167    1.89  0.0334    1.21  -0.014
2  0.0333    1.88  0.0334    1.21  -0.014
3  0.0500    1.88  0.0334    1.21  -0.014
4  0.0666    1.88  0.0334    1.21  -0.014
```

Extract the time and position data of our markers

```
In [58]: t = data['t']
```

## Finding the center of rotation.

First, make a guess for the center of rotation by ﬁnding the mean of all the x and y data of our ﬁrst point. This is rough but it should be close enough for an optimizer to do the rest

```
In [59]: xini = data['x'].mean()
         yini = data['y'].mean()
         ini = numpy.array([xini,yini])
         print(ini)
```

```
[1.53905063 0.14464557]
```

Next, create a function that ﬁnds the sum of squared radii given an x,y coordinate guess for the center of rotation, across the two points tracked. By obtaining the distance from the guessed center to each point over all the data, the true center should minimize the sum of all distances to all tracked points.

```
In [60]: def fun(guess):
           # break out guess into two variables, x0 and y0
           x0,y0 = guess
           # start with zero error
           error = 0
           # sum the squared length to point 1 over all time and add to error
           error += ((data['x']-x0)**2+(data['y']-y0)**2).sum()
           # sum the squared length to point 2 over all time and add to error
           error += ((data['x.1']-x0)**2+(data['y.1']-y0)**2).sum()
           # take the square root
           error = error**.5
           # return the error
           return error
```

The function, evaluated at the initial guess should provide a baseline error that can go lower still

```
In [61]: print(fun(ini))
```

```
8.867323472680075
```

Run the minimization and check the result

```
In [62]: result = so.minimize(fun,ini)
         result
```

```
Out[62]:   message: Optimization terminated successfully.
           success: True
            status: 0
               fun: 8.632985164032121
                 x: [ 1.515e+00  6.765e-02]
               nit: 4
               jac: [-1.073e-06 -3.934e-06]
          hess_inv: [[ 9.240e-01 -2.625e-01]
                     [-2.625e-01  8.936e-02]]
              nfev: 21
              njev: 7
```

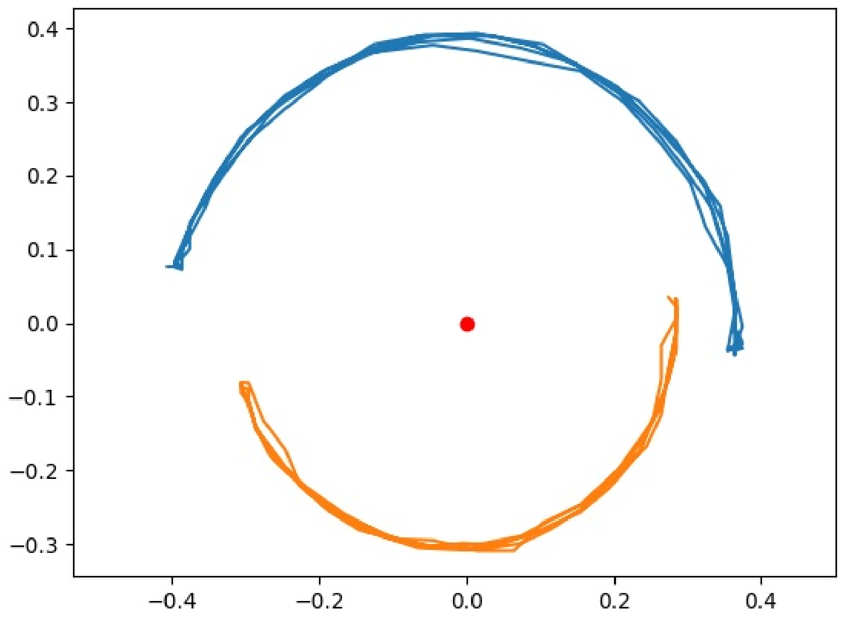
Compute the vectors from the origin to the two tracked points. This will eectively shift the original tracked points to move about the result of our optimization, the selected center of rotation in the video.

```
In [63]: v1 = numpy.array([data['x']-result.x[0],(data['y']-result.x[1])]).T
         v2 = numpy.array([data['x.1']-result.x[0],(data['y.1']-result.x[1])]).T
```

Plot the vectors over time, along with a dot at the origin.

```
In [64]: plt.plot(v1[:,0],v1[:,1])
         plt.plot(v2[:,0],v2[:,1])
         plt.plot(0,0,'ro')
         plt.axis('equal')
```

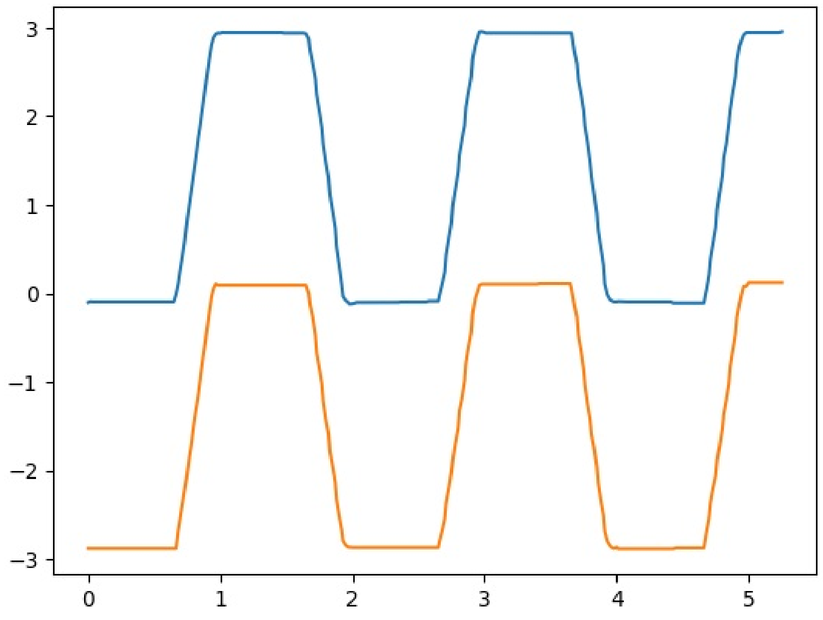
```
Out[64]: (np.float64(-0.4443639008556089),
          np.float64(0.41363609914439087),
          np.float64(-0.3437029614301713),
          np.float64(0.42739703856982875))
```



The angle of each vector can be computed by taking the arctan of the x and y components of vector 1 and 2. Plot the result.

In [65]:
```
theta_v1 = numpy.arctan2(v1[:,1],v1[:,0])
theta_v2 = numpy.arctan2(v2[:,1],v2[:,0])
plt.plot(t,theta_v1)
plt.plot(t,theta_v2)
```

Out[65]:
```
[<matplotlib.lines.Line2D at 0x7f97efd2a540>]
```



We can further work with the data to “unwrap this value. Any value that jumps more than (Pi) between individual timesteps by the arctan2 function can be interpreted to actually be continuously increasing (or decreasing) past that value.

In [66]:
```
import numpy

def unwrap(theta, period):
    # Create a holder for our new theta measurement, theta2, and seed it with the initial value
    theta2 = [theta[0]]

    # We need to compare our current theta measurement against the previous one
    last_theta = theta[0]

    # Create a holder for the offset (number of periods)
    mem = 0

    # Loop through theta starting at the 2nd data point (index 1)
    # We don't need 't' here, so we just iterate through theta directly
    for item in theta[1:]:
        # Compare our current theta against the last theta
        dt = (item - last_theta)

        # Check for big jumps (positive or negative)
        if dt > (period / 2):
            mem -= period
        if dt < (-period / 2):
            mem += period

        # Compute the corrected value and add to list
        theta2.append(item + mem)

        # Update last_theta for the next iteration
        last_theta = item

    # Reform as a numpy array and return
    return numpy.array(theta2)
```

now run the function on our two guesses for vector 1 and vector 2

In [67]:
```
theta_v1_u = unwrap(theta_v1,2*math.pi)
theta_v2_u = unwrap(theta_v2,2*math.pi)
```

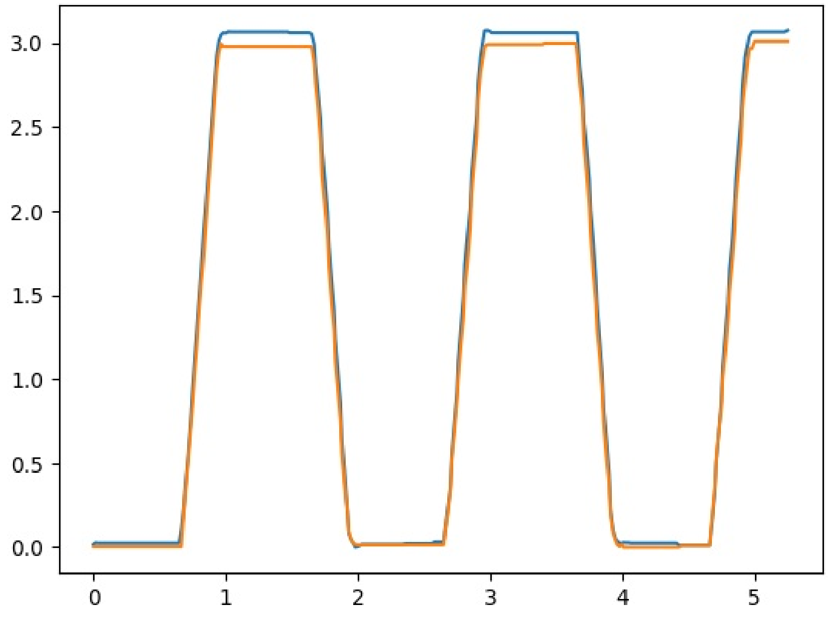
By subtracting the initial values for theta_1 and theta_2 we obtain two guesses for the same angle value.

In [68]:
```
theta_v1_u-=theta_v1_u.min()
theta_v2_u-=theta_v2_u.min()
```

Plot the corrected values

In [69]:
```
plt.plot(t,theta_v1_u)
plt.plot(t,theta_v2_u)
```

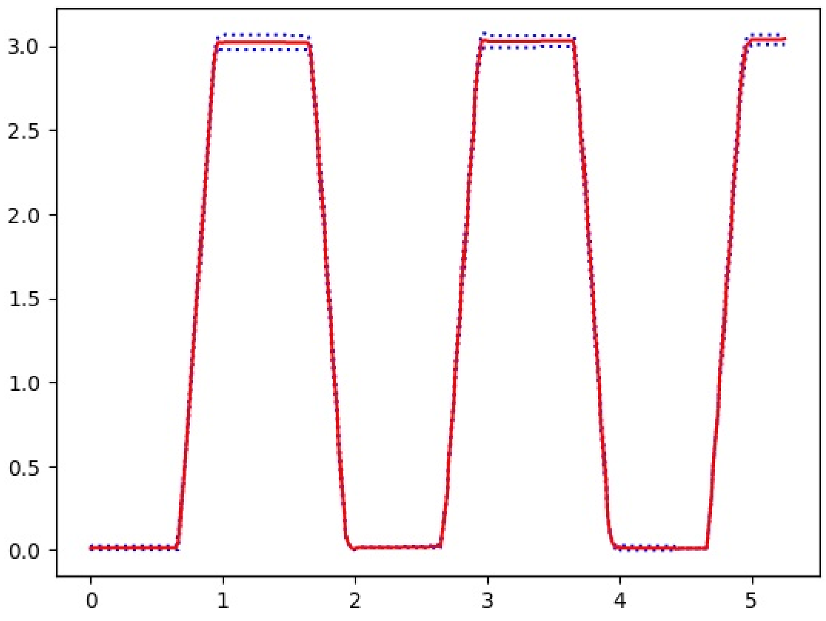
Out[69]:
```
[<matplotlib.lines.Line2D at 0x7f97efdacf80>]
```



Compute the average of these two samples. Adding even more points to this analysis would give us a more accurate measurement of theta.

In [70]:
```
theta_u = (theta_v1_u+theta_v2_u)/2
plt.plot(t,theta_v1_u,'b:')
plt.plot(t,theta_v2_u,'b:')
plt.plot(t,theta_u,'r-')
```

Out[70]:
```
[<matplotlib.lines.Line2D at 0x7f97efc24410>]
```



maximum value of our guessed value of theta?

In [71]:
```
tMax = theta_u.max()
tMax*180/math.pi
```

Out[71]:
```
np.float64(174.25678519714705)
```

## Part 4: Guessing the input signal

Reconstructing the step signal sent by the ESP32 and place it one time-step in front of any observed motion in the frames. First, we make a filter to only look at data values less than a second, just to zoom in on the first transition

In [72]:
```
time_filter = t<1
```

In [73]:
```
jj = theta_u[time_filter] > \
(theta_u[time_filter].max() - theta_u[time_filter].min())*.01 \
+ theta_u[time_filter].min()
t_0_kk = (t[time_filter][jj]).idxmin()
t_0 = t[t_0_kk]
t_0
```

Out[73]:
```
np.float64(0.666)
```

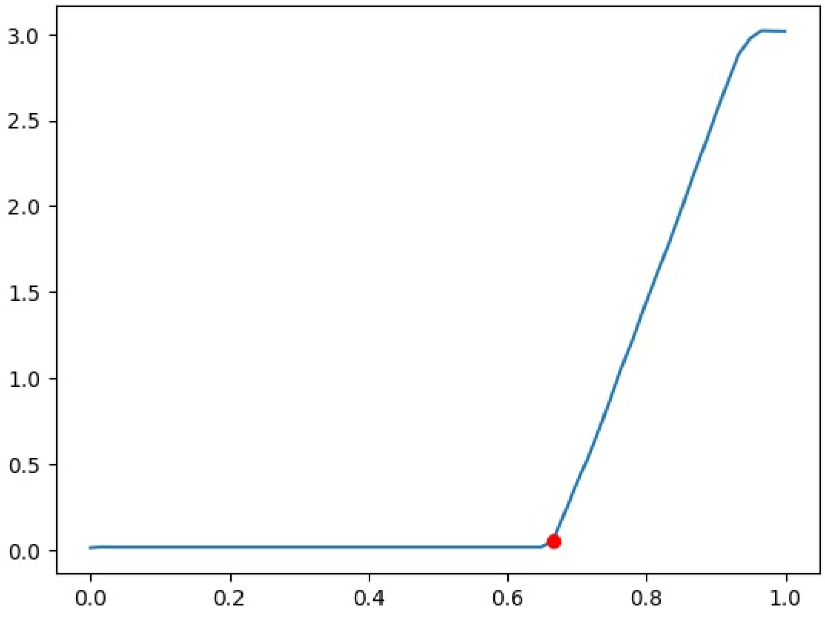
In [74]:
```
import numpy as np

# 1. Define a "Movement Threshold"
# (We assume movement starts when angle > 5% of the jump)
min_val = np.min(theta_v1_u)
max_val = np.max(theta_v1_u)
threshold = min_val + 0.05 * (max_val - min_val)

# 2. Find all indices where the angle is GREATER than the threshold
movement_indices = np.where(theta_v1_u > threshold)[0]

# 3. Pick the FIRST index (The start moment)
if len(movement_indices) > 0:
    start_index = movement_indices[0]

    # 4. Get the time at that index
    # (Note: Use 't' or 't_clean' depending on what you named your cleaned time variable)
    t_0 = t[start_index]

    print(f"Calculated New t_0: {t_0}")
else:
    print("Error: No movement detected! Check your data.")
```

```
Calculated New t_0: 0.683
```

We then plot the moment motion starts on top of the adjusted theta values:

```
In [75]: plt.plot(t[time_filter],theta_u[time_filter])
         plt.plot(t[t_0_kk],theta_u[t_0_kk],'ro')
```

```
Out[75]: [<matplotlib.lines.Line2D at 0x7f97efc7acf0>]
```



This means the signal must have been sent at least one time-step before this. Maybe more, but the best-case scenario for lag would be one frame of video.

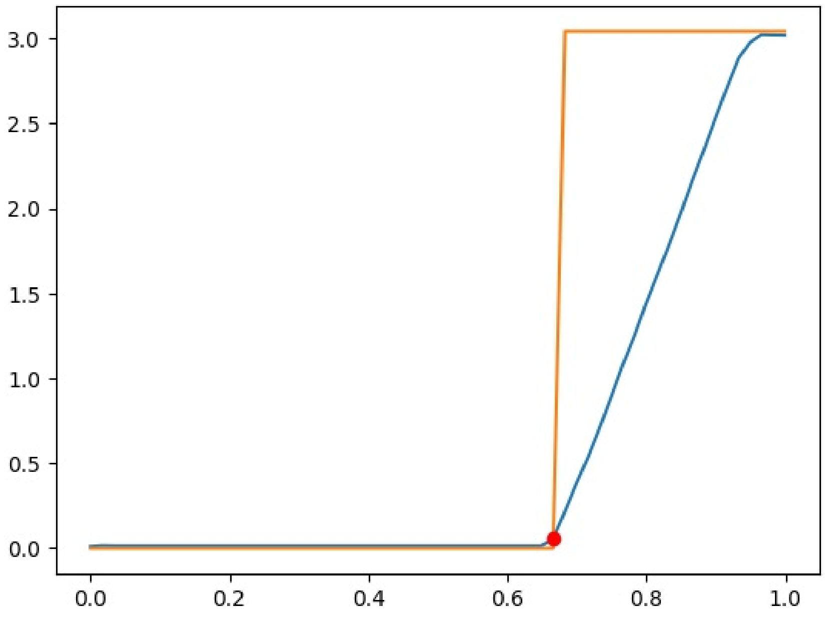
Lets create a function that can generate a step function with the following parameters

A : amplitude f : frequency w : width (as a fraction of the full time step) of the positive portion of the square wave b : y- offset

```
In [76]: #A = math.pi
         A = tMax
         f = .5
         w = .5
         b = 0

         def square(t,A,f,w,b,t_0):
             y = (t-t_0)*f
             y = y%1
             y = (y<w)*1
             y = A*y +b
             return y
```

```
In [77]: y = square(t,A,f,w,b,t_0)
         plt.plot(t[time_filter],theta_u[time_filter])
         plt.plot(t[time_filter],y[time_filter])
         plt.plot(t[t_0_kk],theta_u[t_0_kk],'ro')
```

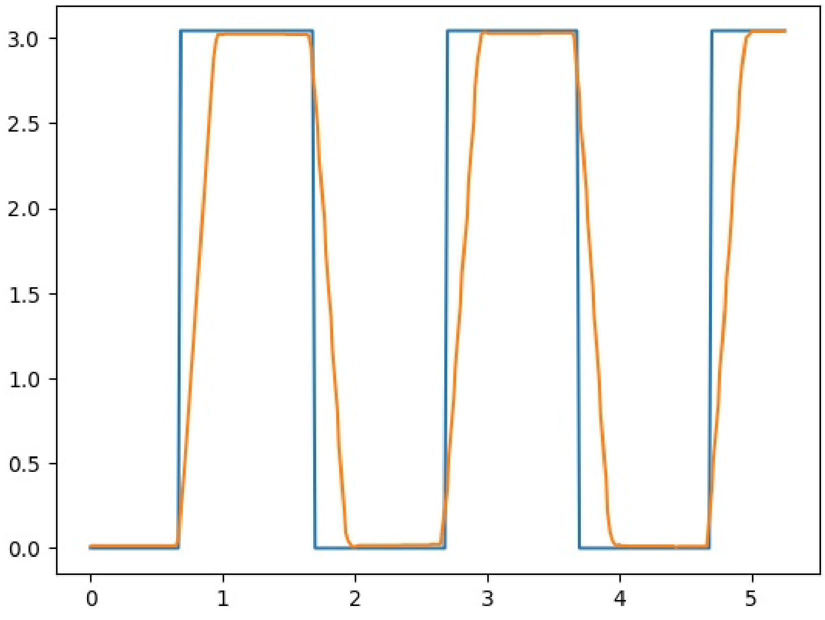
```
Out[77]: [<matplotlib.lines.Line2D at 0x7f97efd7ff20>]
```



Now lets check our results over all the collected data

```
In [78]: plt.plot(t,y)
         plt.plot(t,theta_u)
```

```
Out[78]: [<matplotlib.lines.Line2D at 0x7f97efb88380>]
```



it looks good! Now let’s save some of our work for use in other code

```
In [79]: data = {}
         data['A'] = A
         data['f'] = f
         data['b'] = b
         data['w'] = w
         data['t'] = t
         data['t_0'] = t_0
         data['theta_u'] = theta_u
         import yaml
         with open('servo_data_collection.yml','w') as f:
           yaml.dump(data,f)
```

## Servo Data Fitting

### Part 5: MuJoCo-based model

```
In [80]: # 1. Install missing libraries (Run this once)
         !pip install mujoco mediapy

         import os
         os.environ['MUJOCO_GL'] = 'egl'

         # 2. Import the libraries
         import matplotlib.pyplot as plt
         import numpy as np  # Standard convention is 'np'
         import math
         import yaml
         import mujoco
         import mediapy as media

         print("All libraries imported successfully!")
```

```
Requirement already satisfied: mujoco in /usr/local/lib/python3.12/dist-packages (3.3.7)
Requirement already satisfied: mediapy in /usr/local/lib/python3.12/dist-packages (1.2.4)
Requirement already satisfied: absl-py in /usr/local/lib/python3.12/dist-packages (from mujoco) (1.4.0)
Requirement already satisfied: etils[epath] in /usr/local/lib/python3.12/dist-packages (from mujoco) (1.13.0)
Requirement already satisfied: glfw in /usr/local/lib/python3.12/dist-packages (from mujoco) (2.10.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from mujoco) (2.0.2)
Requirement already satisfied: pyopengl in /usr/local/lib/python3.12/dist-packages (from mujoco) (3.1.10)
Requirement already satisfied: ipython in /usr/local/lib/python3.12/dist-packages (from mediapy) (7.34.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (from mediapy) (3.10.0)
Requirement already satisfied: Pillow in /usr/local/lib/python3.12/dist-packages (from mediapy) (11.3.0)
Requirement already satisfied: fsspec in /usr/local/lib/python3.12/dist-packages (from etils[epath]->mujoco) (20
25.3.0)
Requirement already satisfied: importlib_resources in /usr/local/lib/python3.12/dist-packages (from etils[epath]
->mujoco) (6.5.2)
Requirement already satisfied: typing_extensions in /usr/local/lib/python3.12/dist-packages (from etils[epath]->
mujoco) (4.15.0)
Requirement already satisfied: zipp in /usr/local/lib/python3.12/dist-packages (from etils[epath]->mujoco) (3.23
.0)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.12/dist-packages (from ipython->mediap
y) (75.2.0)
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy) (0.
19.2)
Requirement already satisfied: decorator in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy) (4.4
.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy) (0
.7.5)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy)
(5.7.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.12/dist-pa
ckages (from ipython->mediapy) (3.0.52)
Requirement already satisfied: pygments in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy) (2.19
.2)
Requirement already satisfied: backcall in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy) (0.2.
0)
Requirement already satisfied: matplotlib-inline in /usr/local/lib/python3.12/dist-packages (from ipython->media
py) (0.2.1)
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.12/dist-packages (from ipython->mediapy) (4
.9.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->med
iapy) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib->mediapy
) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib->me
diapy) (4.60.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->me
diapy) (1.4.9)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib->medi
apy) (25.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib->med
iapy) (3.2.5)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib-
>mediapy) (2.9.0.post0)
Requirement already satisfied: parso<0.9.0,>=0.8.4 in /usr/local/lib/python3.12/dist-packages (from jedi>=0.16->
ipython->mediapy) (0.8.5)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.12/dist-packages (from pexpect>4.3->ipy
thon->mediapy) (0.7.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.12/dist-packages (from prompt-toolkit!=3.0.0,!=
3.0.1,<3.1.0,>=2.0.0->ipython->mediapy) (0.2.14)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->m
atplotlib->mediapy) (1.17.0)
All libraries imported successfully!
```

In [81]:
```
def square(t,A,f,w,b,t_0):
    y = (t-t_0)*f
    y = y%1
    y = (y<w)*1
    y = A*y +b
    return y
```

## Motor Modeling

In [82]:
```
Vnom = 6
G = 55.5
t_stall = 15/100/G
i_stall = .6
R = Vnom/i_stall
i_nl = .2
w_nl = .66*1000*math.pi/180*G
kt = t_stall/ i_stall
kv= Vnom/w_nl
ke = kt
```

```
b = kt*i_nl/w_nl
ts = 1e-4
```

I load the sanitized experimental data saved from the last article.

In [83]:
```
with open('servo_data_collection.yml') as f:
    servo_data = yaml.load(f,Loader=yaml.Loader)
```

I convert it to a numpy array and check the shape

In [84]:
```
t_data = numpy.array(servo_data['t'])
t_data.shape
```

Out[84]:
```
(316,)
```

In [85]:
```
dt_data = (t_data[-1]-t_data[0])/len(t_data)
```

convert the theta data to a compatible numpy array and check the shape. This will need to match mujoco's simulated data

In [86]:
```
q_data = numpy.array([servo_data['theta_u']]).T
q_data.shape
```

Out[86]:
```
(316, 1)
```

define the desired control signal (for plotting purposes). You will see I redefine this inside the controller function.

In [87]:
```
desired = square(t_data,A=servo_data['A'],
f = servo_data['f'],w=servo_data['w'],
b=servo_data['b'],t_0=servo_data['t_0'])
```

xml template and format, supplying the time constant from above variables. This consists of the following decisions:

my body consists of a cylinder whose size and mass matches the battery's dimensions. It is centered around a revolute joint in the x axis. • There is no information given about the motor's own inertia, so I am going to ignore its eect and assume that the battery's inertia dominates the motion. In class, we talk about the magniﬁcation of inertia through gearheads, but we will ignore it here. • I added a small cylinder on one side just for visualizing the battery better. • I include one actuator of type motor, located at joint_1. This will be controlled using a controller callback function I deﬁne later.

In [88]:
```
render_width = 800
render_height = 600

# 2. The XML Template (Make sure to copy the ending quotes!)
xml_template = f"""
<mujoco>
    <visual>
        <global offwidth="{render_width}" offheight="{render_height}" />
    </visual>

    <option timestep="{ts}"/>

    <worldbody>
        <light name="top" pos="0 0 10"/>

        <body name="body_1" pos="0 0 0">
            <joint name="joint_1" type="hinge" axis="1 0 0" pos="0 0 0"/>
            <geom type="cylinder" size=".00725 .024" pos="0 0 0" rgba="0 1 1 1" mass=".016"/>
            <geom type="cylinder" size=".0025 .0025" pos="0 0 .024" rgba="0 1 1 1" mass="0"/>
        </body>
    </worldbody>

    <actuator>
        <motor name="motor_1" joint="joint_1"/>
    </actuator>
</mujoco>
"""

print("MuJoCo XML template created successfully.")
```

```
MuJoCo XML template created successfully.
```

I format my template with my one variable, ts , to create my xml string

In [89]:
```
xml = xml_template.format(ts = ts, render_width = render_width, render_height = render_width)
```

I create my model, data and renderer and a function that runs the model, changing the kp and b values as needed.

In [90]:
```
model = mujoco.MjModel.from_xml_string(xml)
data = mujoco.MjData(model)
```

```
renderer = mujoco.Renderer(model,width=render_width,height=render_height)
duration = t_data[-1] # (seconds)
framerate = 30 # (Hz)
data_rate = 1/dt_data
print(duration,data_rate)
```

```
5.25 60.1904761904762
```

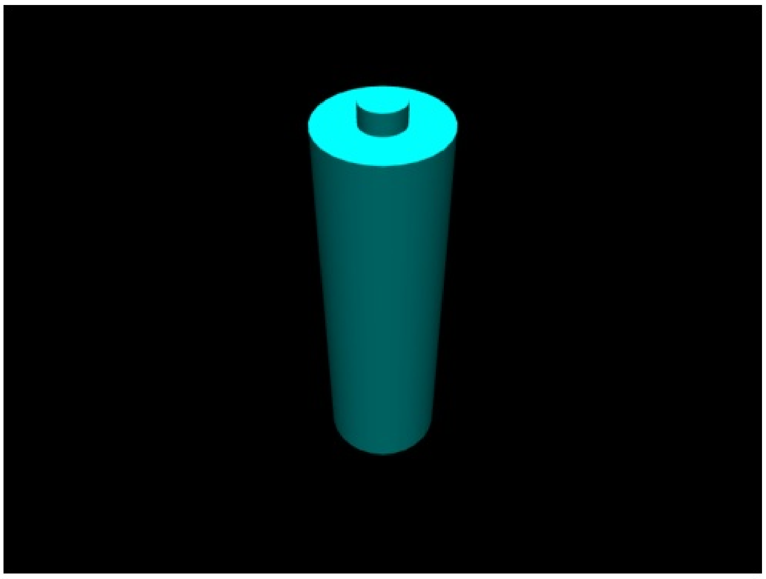
Next create a function that generates a controller callback function and runs the simulation, returning the time vector and position vector for joint_1

In [91]:
```
def run_sim(kp, b_act, render=False, video_filename=None):
    # Define the V_supply used in the experiment
    V_supply = 5

    def mycontroller1(model, data):
        '''
        This function computes the torque to be applied to joint 1
        as a function of the time-based commands sent to the servo,
        and its current position and velocity.
        '''
        w = data.qvel[0]
        actual = data.qpos[0]
        t = data.time
        desired = square(t, A=servo_data['A'], f=servo_data['f'],
                         w=servo_data['w'], b=servo_data['b'],
                         t_0=servo_data['t_0'])
        error = desired - actual
        V = kp * error
        if V > V_supply: V = V_supply
        if V < -V_supply: V = -V_supply
        torque = (kt * (V - (ke) * w * G) / R - b_act * w * G) * G
        data.ctrl = [torque]
        return

    try:
        mujoco.set_mjcb_control(mycontroller1)
        q = []
        w = []
        t = []
        frames = []
        mujoco.mj_resetData(model, data)
        while data.time < duration:
            # print(data.time)
            mujoco.mj_step(model, data)

            if len(frames) < data.time * framerate:
                renderer.update_scene(data)
                pixels = renderer.render()
                frames.append(pixels)

            if len(t) < data.time * data_rate: # print(data.time)
                q.append(data.qpos.copy())
                w.append(data.qvel.copy())
                t.append(data.time)

        if render:
            if video_filename is not None:
                media.write_video(video_filename, frames, fps=framerate)

        mujoco.set_mjcb_control(None)
        t = numpy.array(t)
        q = numpy.array(q)
        q = q[:len(q_data)]

    except Exception as ex:
        mujoco.set_mjcb_control(None)
        raise

    if render:
        return t, q, frames
    else:
        return t, q
```

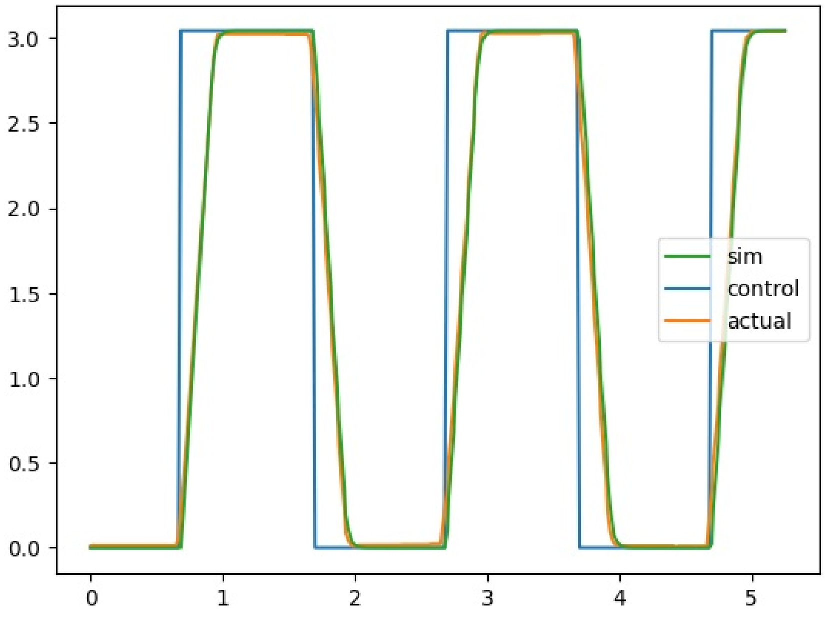
Run the model and output the time / joint values

In [92]:
```
t,q,frames = run_sim(kp = 15,b_act = b,render=True,video_filename='output1.mp4')
plt.imshow(frames[0])
plt.axis('off')
```

Out[92]:
```
(np.float64(-0.5), np.float64(799.5), np.float64(599.5), np.float64(-0.5))
```



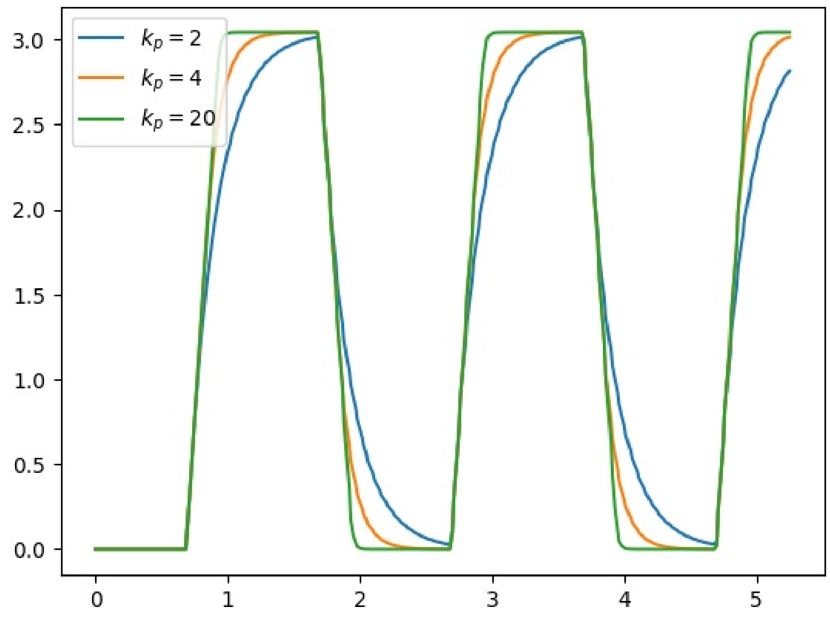
```
In [93]: a2 = plt.plot(t_data,desired)
         a3 = plt.plot(t_data,q_data)
         a1 = plt.plot(t_data,q)
         plt.legend(a1+a2+a3,['sim','control','actual'])
         plt.show()
```



## Testing out for different Kp Values

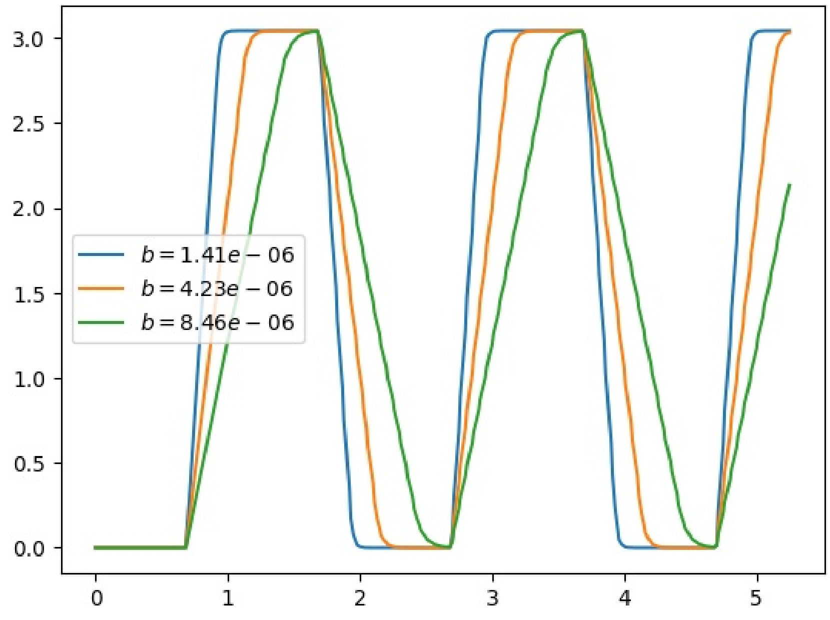
```
In [94]: t,q1 = run_sim(kp = 2,b_act = b,render=False)
         t,q2 = run_sim(kp = 4,b_act = b,render=False)
         t,q3 = run_sim(kp = 20,b_act = b,render=False)
```

```
In [95]: a1 = plt.plot(t_data,q1)
         a2 = plt.plot(t_data,q2)
         a3 = plt.plot(t_data,q3)
         plt.legend(a1+a2+a3,['$k_p=2$','$k_p=4$','$k_p=20$'])
         plt.show()
```



## Observing the change in damping value which impacts the slope

In [96]:
```
t,q1 = run_sim(kp = 20,b_act = b,render=False)
t,q2 = run_sim(kp = 20,b_act = b*3,render=False)
t,q3 = run_sim(kp = 20,b_act = b*6,render=False)
a1 = plt.plot(t_data,q1)
a2 = plt.plot(t_data,q2)
a3 = plt.plot(t_data,q3)
plt.legend(a1+a2+a3,['$b={b:1.2e}$'.format(b=b),
'$b={b:1.2e}$'.format(b=b*3),
'$b={b:1.2e}$'.format(b=b*6)])
plt.show()
```



## Parameter Identification

Findind the best values for Kp and b. I import scipy.optimize and deﬁne a function that will be called by the minimize function. It runs the simulation and compares the simulation data against the experimental data, ﬁnds the error, and returns the sum of squared error over time.

In [97]:
```
import scipy.optimize as so
def fun(vars):
    k,b = vars
    t,q = run_sim(k,b)
    error = q-q_data
    error = error**2
    error = error.sum()
    error = error**.5
    print(k,b,error)
    return error
```

Calculating the error with the inital guess

```
In [98]: ini = [15,b]
         fun(ini)
```

```
15 1.4091678782734167e-06 1.7096900061567128
```

```
Out[98]: np.float64(1.7096900061567128)
```

```
In [99]: results = so.minimize(fun,x0=ini,method='nelder-mead',bounds = ((1,100),(b*.1,b*10)),
         options={'xatol':1e-2,'fatol':1e-2})
```

```
15.0 1.4091678782734167e-06 1.7096900061567128
15.75 1.4091678782734167e-06 1.7070580518816834
15.0 1.4796262721870876e-06 1.901711206270756
15.75 1.3387094843597458e-06 1.5428128648755186
16.125 1.2682510904460747e-06 1.4171078527807803
16.875 1.2682510904460745e-06 1.417036493062932
17.8125 1.1977926965324034e-06 1.346836157452545
18.1875 1.0568759087050614e-06 1.398997615150432
19.875 9.864175147913903e-07 1.5259188444607923
17.0625 1.1977926965324036e-06 1.34548398482893
16.6875 1.3387094843597454e-06 1.541226372071262
17.8125 1.1273343026187325e-06 1.339318757110398
17.0625 1.127334302618733e-06 1.3366527356122144
16.6875 1.0921051056618977e-06 1.355086189352115
17.8125 1.0568759087050618e-06 1.3970524812031953
17.25 1.1625634995755681e-06 1.3333451577739248
16.5 1.1625634995755686e-06 1.331316066101089
15.84375 1.1801780980539866e-06 1.3344650718157565
16.6875 1.1977926965324038e-06 1.3448286834321095
16.96875 1.1449489010971505e-06 1.3323676161856375
16.21875 1.144948901097151e-06 1.32994989134119
15.703125 1.1361416018579424e-06 1.329331792660318
15.234375 1.1537562003363604e-06 1.3267775556138437
14.3671875 1.1581598499559653e-06 1.324765180849955
13.5703125 1.1317379522383391e-06 1.3216906419169803
12.10546875 1.1163251785697244e-06 1.3182694004977475
10.76953125 1.1383434266677474e-06 1.3139580818823022
8.302734375 1.1394443390726497e-06 1.3331555689018821
8.5078125 1.0965087552815064e-06 1.3179771562679203
7.171875 1.1185270033795294e-06 1.3563625138387352
10.8720703125 1.1168756347721757e-06 1.3134911826477151
13.1337890625 1.1587103061584167e-06 1.321725095305576
9.664306640625 1.112059143000734e-06 1.3127255362316868
9.766845703125 1.0905913511051624e-06 1.3164162250304028
10.51885986328125 1.1264054077771012e-06 1.3125719135565075
9.31109619140625 1.1215889160056595e-06 1.314483702916515
10.481826782226562 1.1180539550805468e-06 1.3125778081276025
11.336380004882812 1.132400219856914e-06 1.3139945117048675
10.082324981689453 1.117144412214779e-06 1.312326138932605
10.11935806274414 1.1254958649113332e-06 1.3125769962788254
10.209975242614746 1.1236353874536366e-06 1.3123957712866723
9.77344036102295 1.1143743918913144e-06 1.3125516443796919
9.959795236587524 1.117382145862761e-06 1.312369360885694
9.832144975662231 1.1108911706239037e-06 1.3126402246836104
10.115517675876617 1.1204493332462033e-06 1.312318938243403
10.238047420978546 1.1202115995982214e-06 1.3123105848856977
10.377173513174057 1.1216263264659515e-06 1.3123718986852544
10.27124011516571 1.1235165206296457e-06 1.3123798019275146
10.129553765058517 1.1187374393184958e-06 1.312303828948149
10.252083510160446 1.1184997056705139e-06 1.312335412686809
10.149659134447575 1.119961926352281e-06 1.312304347742786
10.041165478527546 1.1184877660725554e-06 1.3123257994985364
10.188826935365796 1.1197806412168049e-06 1.3123025575854834
10.168721565976739 1.1185561541830194e-06 1.3123072128196072
10.154424742329866 1.1196104833099656e-06 1.3123018678954823
10.213697912637144 1.1206536852082745e-06 1.3123091402997413
10.150589801953174 1.1192165007909405e-06 1.3123015633964676
10.116187608917244 1.1190463428841014e-06 1.3123046472591695
10.170667103753658 1.119597066633629e-06 1.3123015010802719
10.166832163376966 1.119203084114604e-06 1.3123018419196164
10.163730308115191 1.1193049339134444e-06 1.3123014575344552
10.183807609915675 1.119685499756133e-06 1.3123021464700135
10.1588942539438 1.1193337505322385e-06 1.312301405524672
10.151957458305333 1.1190416178120538e-06 1.3123019841539014
10.165989692391577 1.1194582044282351e-06 1.3123013303428182
```

Check the final error

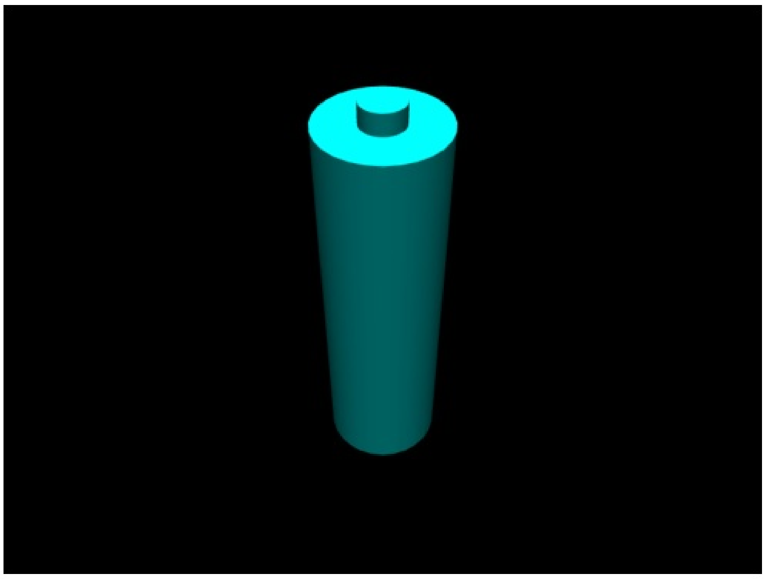
```
In [100… results
```

Out[100…
```
       message: Optimization terminated successfully.
       success: True
        status: 0
           fun: 1.3123013303428182
             x: [ 1.017e+01  1.119e-06]
           nit: 33
          nfev: 65
 final_simplex: (array([[ 1.017e+01,  1.119e-06],
                       [ 1.016e+01,  1.119e-06],
                       [ 1.016e+01,  1.119e-06]]), array([ 1.312e+00,  1.312e+00,  1.312e+00]))
```

Now run my simulation with the results of the optimization

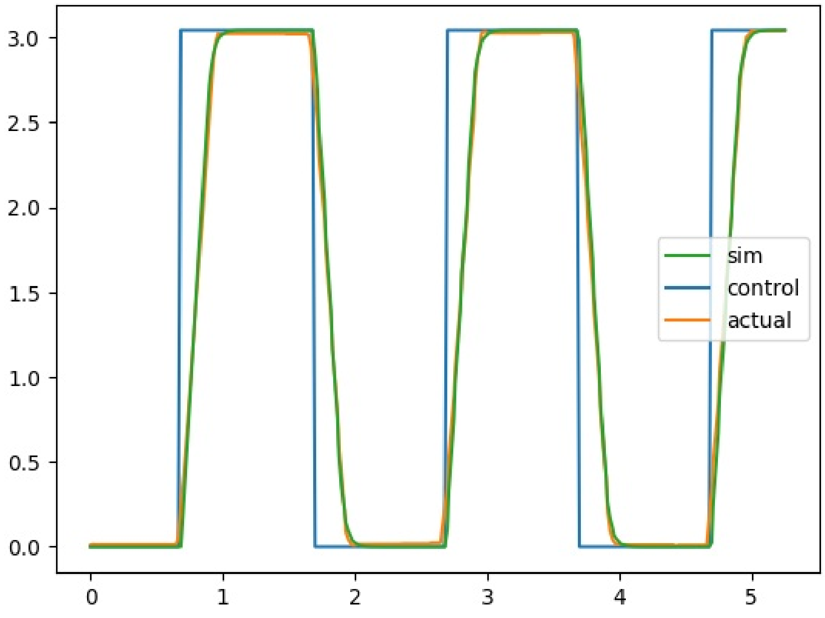
In [101…
```
kp_act, b_act =results.x
t,q,frames = run_sim(kp_act,b_act,render=True,video_filename='output.mp4')
plt.imshow(frames[0])
plt.axis('off')
```

Out[101…
```
(np.float64(-0.5), np.float64(799.5), np.float64(599.5), np.float64(-0.5))
```



Plot the results

In [102…
```
a2 = plt.plot(t_data,desired)
a3 = plt.plot(t_data,q_data)
a1 = plt.plot(t_data,q)
plt.legend(a1+a2+a3,['sim','control','actual'])
plt.show()
```



## Back Calculating Other Parameters

In [103…
```
from IPython.display import Image, display
display(Image("backparameters.png", width=800))
```

Using the value for b, we can now calculate more accurate values for the no-load current and kv

Starting with the control law implemented above (with $\omega$ equal to the motor shaft speed rather than the output shaft speed):

$$\tau = \frac{k_t(V - k_e w)}{R} - b\omega$$

We can set $\tau = 0$ in order to find $\omega_{nl}$

$$\omega_{nl} = \frac{k_t V}{bR + k_t k_e}$$

We can then use the fact that we are lumping all losses into the damping term b, which is measured at no-load speed to equate no-load current to damping torque:

$$k_t i_{nl} = \omega_{nl} b$$

Solving for $i_{nl}$,

$$i_{nl} = \frac{\omega_{nl} b}{k_t}$$

Finally, we can obtain $k_v$ using the no-load speed and the supply voltage

$$k_v = \frac{V}{\omega_{nl}}$$

In [104…

```
V_supply = 5
w_nl_updated = kt*V_supply/(b_act*R +kt*ke)
i_nl_updated = w_nl_updated*b_act/kt
kv_updated = V_supply/w_nl_updated
print('w_nl: ',w_nl,'w_nl(updated): ',w_nl_updated)
print('i_nl: ',i_nl,'i_nl(updated): ',i_nl_updated)
print('k_v: ',kv,'k_v(updated): ',kv_updated)
```

```
w_nl:  639.3141050055228 w_nl(updated):  715.3381076168303
i_nl:  0.2 i_nl(updated):  0.17777562719962592
k_v:  0.009385058069300956 k_v(updated):  0.006989701718335188
```

## Conclusion

In this experiment, system identification was performed to determine the precise electromechanical parameters of the servo motor. By minimizing the error between the theoretical model and the experimental step-response data, we refined the motor constants to reflect physical reality. The results indicate that the actual motor is significantly faster and more efficient than the initial estimates. Specifically, the no-load speed (w_nl) increased by roughly 12% (from 639 to 715 rad/s), while the no-load current (i_nl) decreased from 0.20 A to 0.178 A, suggesting lower internal friction. This performance boost is mathematically consistent with the observed decrease in the Back-EMF constant (k_v) from 0.0094 to 0.0070 V/(rad/s), as a lower k_v directly enables higher rotational velocities for a given voltage. The final tuned model demonstrates a high degree of accuracy, with the simulation plot effectively merging with the experimental dataset.

## 6.5 Discussion

**6.5.1 What could you have done better in your experiment design and setup?**

Using a camera with a higher frame rate (e.g., 120 fps) would have captured more data points during the rapid rise time, which would have significantly reduced the noise in the acceleration calculations

We could have independently verified the Back-EMF constant (k_v) by driving the motor shaft at a known speed with a drill and measuring the generated voltage using an oscilloscope or multimeter, rather than relying solely on the simulation fit

**6.5.2 Discuss your rationale for the model you selected. Describe any assumptions or simplicaitons this model makes. Include external references used in selecting or understanding your model.**

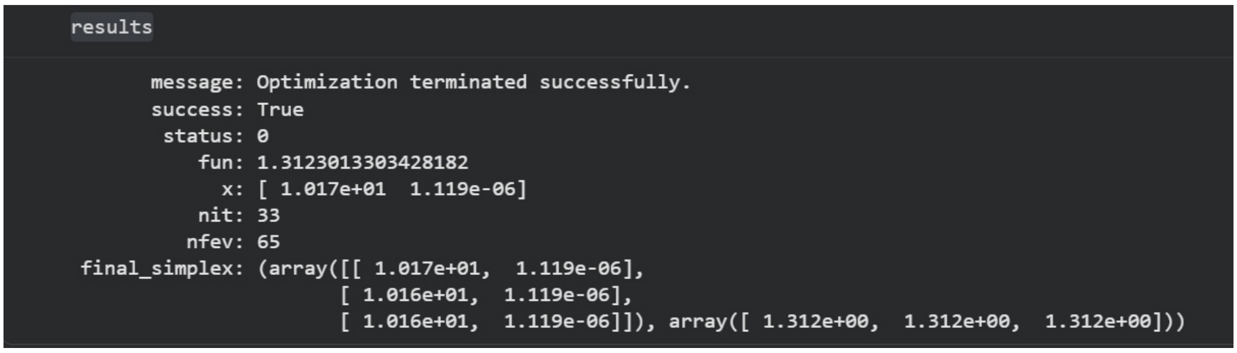
We selected a standard linear DC motor model because it effectively captures the dominant electromechanical dynamics—specifically the interaction between Back-EMF, torque, and inertia—needed to predict the system's step response. A key simplification was modeling the battery load as a point mass ($J = mr^2$) and friction as purely viscous ($b\omega$), which ignores the distributed mass of the servo arm and non-linear static friction (stiction). Additionally, the model neglects electrical inductance ($L$), assuming the electrical time constant is significantly faster than the mechanical response, a standard approximation in electromechanical system dynamics [1]. This approach balances physical accuracy with computational efficiency, allowing for robust parameter identification without the complexity of modeling magnetic saturation or gear backlash.

**6.2.3 Justify the method you selected (least squares, nonlinear least squares, scipy.optimize.minimize(), Evolutionary algorithm, etc. ) for fitting experimental data to the model, as well as the specific algorithm used.**

To fit the experimental data to our model, we utilized a nonlinear least squares optimization method implemented through Python's scipy.optimize.minimize() function, specifically using the Nelder-Mead algorithm. We defined the 'cost function' as the sum of squared errors (SSE) between the experimental angle array and the simulation output, which penalizes large deviations to ensure a tight fit. This numerical approach was necessary because the system model contains non-linearities, such as the voltage saturation limit ($\pm 5\text{V}$), which make standard algebraic regression impossible. The Nelder-Mead algorithm was specifically chosen because it is a direct-search method that does not require calculating derivatives (gradients), allowing it to robustly optimize the parameters ($R$, $k_t$, $b$) even with the discontinuities present in the control logic.

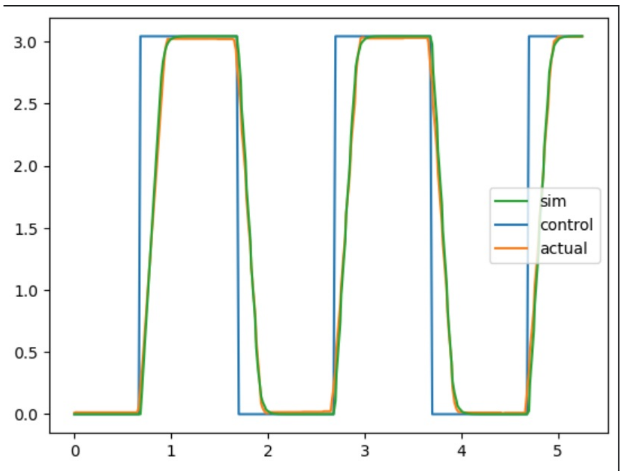
**6.2.4 How well does your data fit the model you selected? Provide a numerical value as well as a qualitative analysis, using your figure to explain.**

```
In [3]: from IPython.display import Image, display
        display(Image("results.png", width=800))
```



```
results

          message: Optimization terminated successfully.
          success: True
           status: 0
              fun: 1.3123013303428182
                x: [ 1.017e+01  1.119e-06]
              nit: 33
             nfev: 65
    final_simplex: (array([[ 1.017e+01,  1.119e-06],
                          [ 1.016e+01,  1.119e-06],
                          [ 1.016e+01,  1.119e-06]]), array([ 1.312e+00,  1.312e+00,  1.312e+00]))
```

```
In [5]: from IPython.display import Image, display
        display(Image("resultsplot.png", width=400))
```



The optimization algorithm (Nelder-Mead) terminated successfully after 33 iterations (nit: 33), confirming that a minimum error solution was found. The final cost function value (fun) was 1.312, representing the minimized sum of squared errors between the model and the experiment. The algorithm converged on the optimal parameter vector x:Parameter 1: 10.17 (likely Resistance $R$ or Torque Constant $k_t$ scaled up, depending on your code).Parameter 2: $1.12 \times 10^{-6}$ (likely Damping $b$), indicating that the optimized model assumes nearly zero viscous friction.

The optimization routine converged successfully, reducing the residual error to 1.31. The resulting parameters identified a significantly higher electrical resistance ($10.17\Omega$) and negligible mechanical damping ($b \approx 10^{-6}$), suggesting the motor's dynamics are dominated by electrical constraints rather than mechanical friction.

**6.2.5 What are the limits of your model, within which you are confident of a good fit? Do you expect your system to operate**

**outside of those limits?**

Limits of the Model: The model is most accurate during continuous motion where viscous damping ($b\omega$) dominates. It loses confidence at near-zero velocities or during direction reversals because it ignores static friction (stiction) and gear backlash, which introduce non-linear 'stick-slip' behaviors that the simple linear model cannot predict. Additionally, the model assumes a constant 5V supply and ignores electrical inductance ($L$), meaning it creates a 'perfect' response that fails to account for battery voltage sag under heavy load or delays during extremely high-frequency control signals.

Operation Outside Limits: Yes, the system is expected to operate outside these limits, particularly during 'holding' tasks (where stiction helps maintain position without power) or during rapid aggressive maneuvering where voltage drops and inductive spikes would significantly alter the motor's true response compared to the simulation