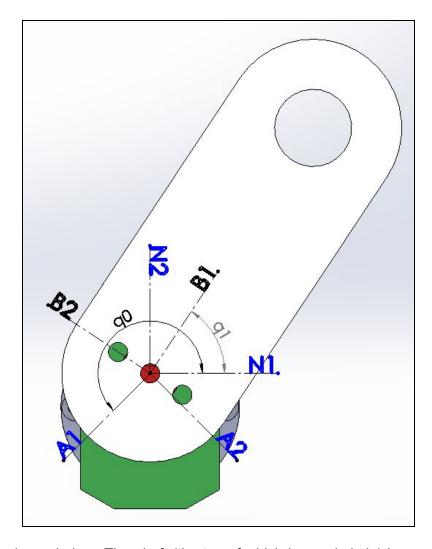
# Intro to Robotics Exercise 3

Samuel Law & Rhys Miller

## System Diagram



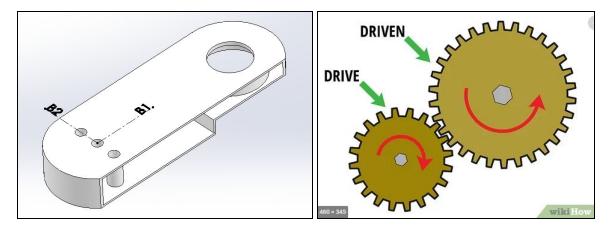
The system is shown below. The shaft (the top of which is a red circle) is considered body A, and the link is body B. The shaft's A.3 axis is in the same direction as the N.3 and B.3 axis, however, the A.1 axis is at an angle q0 from the N.1 axis, not an angle q1. The COM of body A is assumed to be on its polar axis. The COM of body B is assumed to be at the dot next to the text B1. which is a distance LB away from point N, which is located at the origin.

The constraints equation is  $\Delta q_0 = -q_1 gr$  where the q's are the generalized coordinates and gr is the gear ratio between the motor shaft and the motor output spindle. The constraint equation carries to both the velocity and acceleration terms.

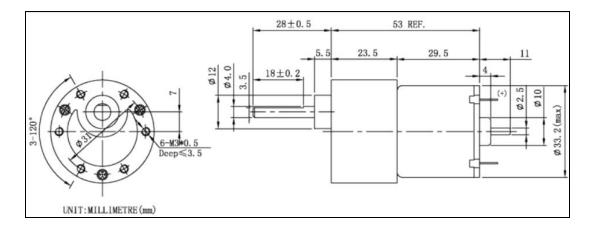
Thus the system has two generalized coordinates, q0 and q1, and a single gear constraint equation, thus the system has a single degree of freedom

The gear ratio used in the gear constraint was printed on the side of the motor.

### **Mass Properties**



The mass for the link, body B, was computed using ABS mass properties from SolidWorks applied to the CAD model. The model was hollowed out, and the shell thickness set to 0.90 mm, the same as was used to print the physical model. The internally printed structure was neglected as a sparse infill was used.



The mass of the axle that spins in body A was calculated using the drawing shown above. The axle was assumed to be 10.0 mm in diameter and 53 mm in length. A CAD model was generated, and the mass estimated with steel mass properties applied to the CAD model. The gears were assumed to interact in a manner similar to the ones shown below

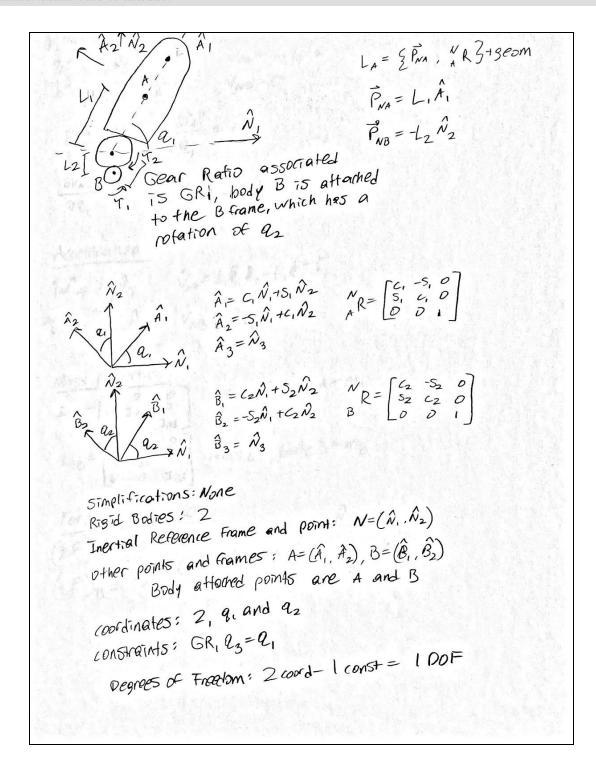
The moments of inertia for both bodies were evaluated in the same manner as the mass properties.

Body	Mass [kg]	lzz [kg m²]
Body A	26.45 x 10 <sup>-3</sup>	3.30 x 10 <sup>-7</sup>
Body B	7.28 x 10 <sup>-3</sup>	6.39 x 10 <sup>-6</sup>

### **Equations of Motion**

Per the assignment, the equations of motion were derived by hand. This is inconsistent with prior exercises, and points should not be taken off for not showing the equations of motion derived computationally. A direct quotation from the instructions is shown below.

Write up this lab as a homework problem including the development of the equations of motion by hand. Make sure to include:



$$\frac{\text{Velocity}}{\text{NW}^{A}} = \dot{q}_{1} \hat{N}_{3}$$

$$\text{NW}^{B} = \dot{q}_{3} \hat{N}_{3} = \frac{\dot{q}_{1}}{GR_{1}} \hat{N}_{3}$$

$$\text{NW}^{B} = D$$

$$\frac{\partial V_{NM}}{\partial \dot{q}_{1}} = L_{1} \hat{A}_{2}$$

$$\frac{\partial V_{M}}{\partial \dot{q}_{1}} = \lambda_{1} \hat{A}_{2}$$

$$\frac{\partial V_{M}}{\partial \dot{q}_{1}} = \lambda_{2} \hat{q}_{1}$$

$$\frac{\partial V_{NM}}{\partial \dot{q}_1} = L_1 \hat{A}_2 \quad \frac{\partial \hat{W}^A}{\partial \dot{q}_1} = \hat{N}_3$$

$$\frac{\partial \dot{v}}{\partial \dot{e}_{1}} = 0 \qquad \frac{\partial \dot{w}}{\partial \dot{e}_{1}} = \frac{1}{6R_{1}} \dot{N}_{3} \qquad \frac{\partial \dot{w}}{\partial \dot{e}_{2}} = 0$$

$$\frac{Acceleration}{\text{Ni}^{A} = \dot{e}_{1} \hat{N}_{3}} \quad \dot{V}_{NA} = L_{1} \ddot{e}_{1} \hat{A}_{2} - L_{1} \dot{e}_{1}^{2} \hat{A}_{1}$$

$$\text{Ni}^{B} = \frac{\dot{e}_{1}}{GR_{1}} \hat{N}_{3} \quad \dot{V}_{NB} = 0$$

$$v_{ij}^{B} = \frac{\dot{q}_{1}}{GR_{1}} \hat{N}_{3} \qquad \dot{V}_{MB} = C$$

$$\frac{Mass properties}{I_{AA} = \begin{bmatrix} I_{A} & 0 & 0 \\ 0 & I_{A2} & 0 \\ 0 & 0 & I_{A3} \end{bmatrix}} \quad mass of pody A = M_A$$

Mass properties

$$I_{AA} = \begin{bmatrix} I_{A}, & 0 & 0 \\ 0 & I_{A2} & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \quad \text{mass of body } B = M_{B}$$

$$I_{BB} = \begin{bmatrix} I_{B1} & 0 & 0 \\ 0 & I_{B2} & 0 \\ 0 & 0 & I_{B3} \end{bmatrix} \quad \text{mass of body } B = M_{B}$$

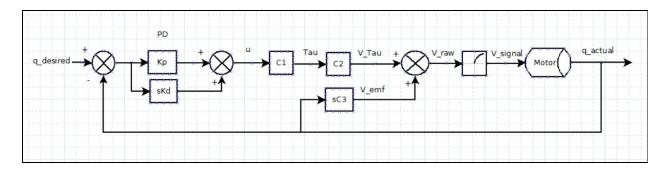
Forces and moments
$$\underbrace{\{\Xi F\}_{A} = -g m_{A} \hat{N}_{3}}_{EF} \underbrace{\{\Xi M\}_{AA} = -7 \Upsilon_{2} \hat{N}_{3}}_{EF} \underbrace{\{\Xi M\}_{BB} = -\Upsilon_{1} \hat{N}_{3}}_{F}$$

$$\underbrace{\{\Xi F\}_{B} = -g m_{B} \hat{N}_{3}}_{EF} \underbrace{\{\Xi M\}_{BB} = -\Upsilon_{1} \hat{N}_{3}}_{F}$$

$$(2F)_{R} = -9m_{B}N_{3}$$
  $(2M)_{BB} = -7N_{3}$ 

$$\frac{\partial H_{AA}}{\partial t} = I_{AA} \stackrel{VV}{V}^{A} + \stackrel{VV}{V}^{A} \times I_{AA} \stackrel{VV}{V}^{A} = \begin{bmatrix} I_{A1} & 0 & 0 & 0 \\ 0 & I_{A2} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I_{A3} \end{bmatrix} \begin{bmatrix} I_{B_{1}} & 0 & 0 \\ 0 & 0 & I$$

### **Block Diagram & Discussion**



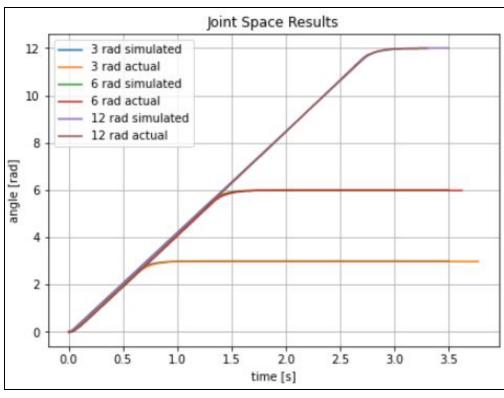
All simulations were performed using an RK45 integrator. The Kp value used was calculated based on the CPR of the encoder, which was set to 1000. The equation used is shown below, where Vmin is the minimum voltage required for the motor to begin moving. Vmin was determined via experimentation. For the joint space model, Kv was simply set so as to be critically damped. The saturation block between V\_raw and V\_signal indicates that the voltage signal was constrained between 0 and 12 volts.

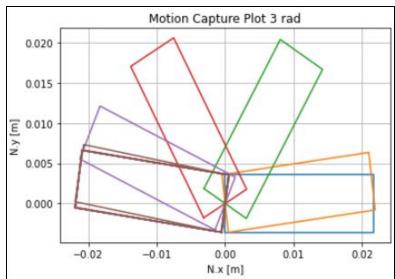
$$V_{min} = K_p(\frac{2\pi}{CPR})$$

The only change in the block diagram for the PD controller was that the C1 block was removed, thus eliminating any kind of compensation for mass properties. This meant that the controller had to be tuned in order to work correctly.

# Joint Space Simulation & Experiment Results

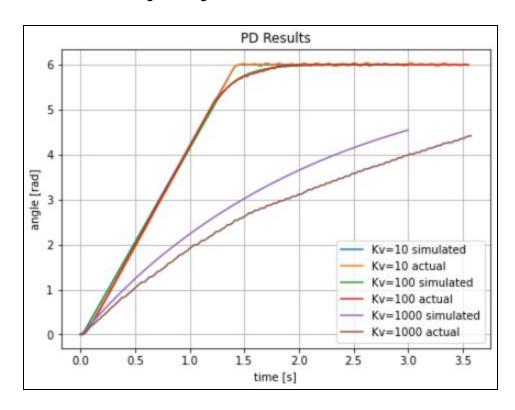
The joint space controller simulation closely matched the experimental results. That being said a slight DC gain of 4.5 had to be applied to the controller signal in order to overcome the static friction in the system and get the motor to not stop short.





# PD Simulation & Experiment Results

The PD controller did not behave as simulated. Due to the massive increase in the strength of the signal, as C1 was a very small number and now was absent from the controller, the controller required a DC gain of 0.5 to function in the realm of normal. There was also substantial vibration. A PID controller is recommended as it would allow for lower Kp and Kv values, while not undershooting the target.



# Team Photo (note, only one person due to COVID)



# References

[gears] https://www.wikihow.com/Determine-Gear-Ratio

```
TITLE: Intro to Robotics Exercise 3
              AUTHORS: Rhys Miller & Samuel Law
     In [1]: | %reset -f
     In [2]: %matplotlib inline
     In [3]: import sympy as sm
              import sympy.physics.mechanics as me
              import matplotlib.pyplot as plt
              from IPython.display import Latex, display
              from numpy import linspace, array, pi, sqrt, polyfit, vstack, sign
              from scipy.integrate import RK45
     In [4]: def show(eq_lhs: str, eq_rhs: str):
                  eq = f"${eq_lhs} = {eq_rhs}$"
                  return display(Latex(eq))
              Kinematics
     In [5]: # create the inertial frame
              N = me.ReferenceFrame("N")
              pN = me.Point("pN")
              pN.set_vel(N, 0)
     In [6]: # define the constant g
              g, gear_ratio = sm.symbols("g, gear_ratio")
     In [7]: # declare independent variables
              q1, u1 = me.dynamicsymbols("q1, u1")
              q1d, u1d = me.dynamicsymbols("q1, u1", 1)
              q1dd = me.dynamicsymbols("q1", 2)
     In [8]: # declare dependent variables
              q0 = -q1*gear_ratio
              q0d = -q1d*gear_ratio
              q0dd = -q1dd*gear_ratio
              u0 = -u1*gear_ratio
     In [9]: # create the bodies
              bodyA = me.Body(
                  name = "bodyA",
                  frame = N.orientnew("A", "axis", (q0, N.z)),
              bodyB = me.Body(
                  name = "bodyB",
                  frame = N.orientnew("B", "axis", (q1, N.z))
    In [10]: # define the motor shaft diameter
              bodyA.D = sm.symbols("D_A")
              # define the link length to bodyB COM
              bodyB.L = sm.symbols("L_B")
    In [11]: # define positions of the COMs
              bodyA.masscenter = pN.locatenew("pA", 0)
              bodyB.masscenter = pN.locatenew("pB", bodyB.L*bodyB.frame.x)
    In [12]: # define the velocities
              bodyA.masscenter.set_vel(N, bodyA.masscenter.pos_from(pN).dt(N))
              bodyB.masscenter.set_vel(N, bodyB.masscenter.pos_from(pN).dt(N))
    In [13]: # define accelerations
              bodyA.masscenter.set_acc(N, bodyA.masscenter.vel(N).dt(N))
              bodyB.masscenter.set_acc(N, bodyB.masscenter.vel(N).dt(N))
              Dynamics
    In [14]: # apply kanes method
              KM = me.KanesMethod(
                  frame = N,
                  q_{ind} = [q1],
                  u_ind = [u1],
                  kd_{eqs} = [q1d - u1]
    In [15]: # create a list of the bodies
              bodies = [bodyA, bodyB]
    In [16]: # define the applied loads
              tau1 = sm.symbols("tau_1")
              loads = [
                  # point/frame, force.moment
                   (bodyA.masscenter, -g*bodyA.mass*N.z),
                  (bodyB.masscenter, -g*bodyB.mass*N.z),
                  (bodyB.frame, tau1*bodyA.frame.z)
    In [17]: # apply the loads and bodies to the KM object
              KM.kanes_equations(bodies, loads);
    In [18]: # extract the mass and force matrix
              MM = KM.mass_matrix
              FM = KM.forcing
    In [19]: # seperate the force out into driving and non-driving
              A, b = sm.linear_eq_to_matrix(FM, tau1)
              TM = A.inv()*(MM*sm.Matrix([q1dd]) + b)
              show("TM", me.vlatex(TM))
              TM = \left[ \left( L_B^2 body B_{mass} + body A_{izz} gear_{ratio}^2 + body B_{izz} \right) q_1 \right]
    In [20]: # create a dict to hold numeric values
              vals = {
                  # lengths
                  bodyB.L: 0.02162775, # m
                  bodyB.mass: 0.00728203, # kg
                  "bodyB_izz": 6.39e-06, # kg*m^2
                  "bodyA_izz": 3.3e-07, # kg*m^2
                  gear_ratio: 144,
              Joint Space Simulation
              The following block diagram requires several assumptions to model accurately
               1. The signal is constrained to between 0 and 5 volts
                2. The motor and q actual have a linear relationship
                3. Due to nonlinear behavior that limits the voltage 12V,
                 modeling in time domain is easier.
                4. The delay between the control signal and the motor signal is neglegable
    In [21]: # create variables for desired output
              q1_desired, q1d_desired = sm.symbols("q1_desired, q1d_desired")
              # creatue variables for current output
              q1_current, q1d_current = sm.symbols("q1_current, q1d_current")
    In [22]: # create the controller signal
              Kp, Kv = sm.symbols("Kp, Kv")
              u = (Kv*(q1d\_desired - q1d\_current)) + (Kp*(q1\_desired - q1\_current))
    Out[22]:
              Kp(-q_{1current} + q_{1desired}) + Kv(-q_{1desired} + q_{1desired})
    In [23]: # create the constant C1
              C1, * = TM.subs(vals)/q1dd # kg*(m^2)
              C1
    Out[23]: 0.00685267623922198
    In [24]: # create the constant C2
              R = 7.5
                          # ohms
              kt = 0.891514 # Nm/amp, from manufacture's page
              C2 = R/kt
              C2
    Out[24]: 8.412655325659497
              SIMULATION CONTROLS
    In [25]: # calculate max Kp value
              V_{min} = 3
                                           # voltage required for motor to actually move based on experimentation
              CPR = 1000
                                           # counts per revolution
              rad_min = 2*pi/1000 # smallest detectable error
              Kp_max = V_min/(rad_min) # the 2 allows it to be +- one count from desired position
              Kp_max
    Out[25]: 477.464829275686
    In [26]: # create all the callables
              simulation_parameters = {
                  q1_desired: 3,
                  q1d_desired: 0,
                  Kp: int(Kp_max),
                  Kv: sqrt(477*8),
    In [27]: # create controller callable
              controller_signal = sm.lambdify([q1_current, q1d_current], u.subs(simulation_parameters), modules="numpy")
              controller_signal(3, 0)
    Out[27]: 0.0
    In [29]: # create motor torque callable to drive q1dd
              rpms = (0, 40)
              taus = (0.980665, 0.4413)
              slope, intercept = polyfit(rpms, taus, 1)
              def tau_motor(x):
                  if rpms[0] <= x <= rpms[1]:
                       return (slope*x) + intercept
                       return 0
              x1 = q1, x1d = q1d
              x2 = q1d, x2d = q1dd
    In [30]: # create the RK45 simulation
              def model(t, x):
                  # unpack the vector
                  q1, q1d = x
                  # calculate the controller signal
                  u = controller_signal(q1, q1d)
                  # get globals
                  global C1
                  # calculate tau
                  tau_desired = abs(u*C1)
                  # calculate RPM
                  RPM = abs(q1d*30/pi)
                  # check if desired torque is in the opposite direction of RPM
                  if sign(u) != sign(q1d):
                       tau_max = tau_motor(0)
                  else:
                       tau_max = tau_motor(RPM)
                  # determine value of tau
                  tau = tau_desired if tau_desired < tau_max else tau_max</pre>
                  # calculate the acceleration due to torque
                  q1dd = tau*sign(u)/C1
                  # return the xdots
                  x1d = q1d
                  x2d = q1dd
                  return (x1d, x2d)
              # create the initial conditions
              simulator = RK45(model, 0, [0, 0], 3.5, max_step=0.05)
              results = array([0, 0])
              while True:
                  try:
                      simulator.step()
                  except:
                       break
                       results = vstack([results, (simulator.t, simulator.y[0])])
              # save the results to a file
              # with open("joint_space_3rad.csv", "w") as f:
                    for r in results:
                         f.write(f"{r[0]}, {r[1]}\n")
              Joint Space Results
    In [31]: # plot the results from the 3 second simulation and experimentation
              fig, ax = plt.subplots(figsize=(7, 5))
              with open("joint_space_3rad.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])  for v  in res]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="3 rad simulated")
              with open("joint_space_3rad_actual.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])/1000 \text{ for } v \text{ in res}]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="3 rad actual")
              with open("joint_space_6rad.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0]) for v in res]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="6 rad simulated")
              with open("joint_space_6rad_actual.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])/1000 \text{ for } v \text{ in res}]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="6 rad actual")
              with open("joint_space_12rad.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0]) for v in res]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="12 rad simulated")
              with open("joint_space_12rad_actual.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])/1000 \text{ for } v \text{ in res}]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="12 rad actual")
              ax.grid()
              ax.set_title("Joint Space Results")
              ax.set_xlabel("time [s]")
              ax.set_ylabel("angle [rad]")
              ax.legend();
                                     Joint Space Results

    3 rad simulated

                     3 rad actual

    6 rad simulated

    6 rad actual

                     — 12 rad simulated
                     — 12 rad actual
                                1.0
                                      1.5
                                                  2.5
                          0.5
                                            2.0
                                         time [s]
              Motion Capture Plot
    In [32]: # create the points to connect
              bodyB.W = bodyB.L/3
              p1 = pN.locatenew("p1", 0.5*bodyB.W*bodyB.frame.y)
              p2 = p1.locatenew("p2", bodyB.L*bodyB.frame.x)
              p3 = p2.locatenew("p3", bodyB.W*-bodyB.frame.y)
              p4 = p3.locatenew("p4", bodyB.L*-bodyB.frame.x)
              # turn the points into a callable
              points = [p1, p2, p3, p4, p1]
              points = [p.pos_from(pN).to_matrix(N)[0:2] for p in points]
              points = [[p.subs(vals) for p in tup] for tup in points]
              points = [[p.subs(vals) for p in tup] for tup in points]
              points_func = sm.lambdify([q1], points, modules="numpy")
    In [33]: for ang in results[0::5, 1]:
                  points = points_func(ang)
                  x = [p[0]  for p  in points]
                  y = [p[1]  for p  in points]
                  plt.plot(x, y)
              plt.grid()
              plt.title("Motion Capture Plot 3 rad")
              plt.xlabel("N.x [m]");
              plt.ylabel("N.y [m]");
                                 Motion Capture Plot 3 rad
                 0.020
                 0.015
               돌 0.010
                 0.005
                 0.000
                                                          0.02
                       -0.02
                                -0.01
                                         0.00
                                                 0.01
                                        N.x [m]
              PD Controller Simulation
              The PD controller is close to the joint space controller, however, the
              PD controller does not feed into a constant C1 in order to account for mass
              properties when computing the desired torque.
    In [34]: # create all the callables
              simulation_parameters = {
                  q1_desired: 6,
                  q1d_desired: 0,
                  Kp: int(Kp_max),
                  Kv: 10
    In [35]: # create controller callable
              controller_signal = sm.lambdify([q1_current, q1d_current], u.subs(simulation_parameters), modules="numpy")
              controller_signal(6, 0)
    Out[35]: 0
    In [36]: # create the RK45 simulation
              def model(t, x):
                  # unpack the vector
                  q1, q1d = x
                  # calculate the controller signal
                  u = controller_signal(q1, q1d)
                  # calculate tau
                  tau_desired = abs(u)
                  # calculate RPM
                  RPM = abs(q1d*30/pi)
                  # check if desired torque is in the opposite direction of RPM
                  if sign(u) != sign(q1d):
                      tau_max = tau_motor(0)
                  else:
                       tau_max = tau_motor(RPM)
                  # determine value of tau
                  tau = tau_desired if tau_desired < tau_max else tau_max
                  # get globals
                  global C1
                  # calculate the acceleration due to torque
                  q1dd = tau*sign(u)/C1
                  # return the xdots
                  x1d = q1d
                  x2d = q1dd
                  return (x1d, x2d)
              # create the initial conditions
              simulator = RK45(model, 0, [0, 0], 3, max_step=0.1)
              results = array([0, 0])
              while True:
                  try:
                       simulator.step()
                  except:
                      break
                  else:
                      results = vstack([results, (simulator.t, simulator.y[0])])
              # save the results to a file
              with open("PD_10.csv", "w") as f:
                  for r in results:
                      f.write(f"{r[0]}, {r[1]}\n")
    In [37]: # plot the results from the 3 second simulation and experimentation
              fig, ax = plt.subplots(figsize=(7, 5))
              with open("PD_10.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0]) for v in res]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="Kv=10 simulated")
              with open("PD_10_actual.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])/1000 \text{ for } v \text{ in res}]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="Kv=10 actual")
              with open("PD_100.csv", "r") as f:
                   res = [l.split(",") for l in f.readlines()]
                  times = [float(v[0])  for v  in res]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="Kv=100 simulated")
              with open("PD_100_actual.csv", "r") as f:
                   res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])/1000 \text{ for } v \text{ in res}]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="Kv=100 actual")
              with open("PD_1000.csv", "r") as f:
                  res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])  for v  in res]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="Kv=1000 simulated")
              with open("PD_1000_actual.csv", "r") as f:
                   res = [1.split(",") for 1 in f.readlines()]
                  times = [float(v[0])/1000 \text{ for } v \text{ in res}]
                  angles = [float(v[1]) for v in res]
                  ax.plot(times, angles, label="Kv=1000 actual")
              ax.grid()
              ax.set_title("PD Results")
              ax.set_xlabel("time [s]")
              ax.set_ylabel("angle [rad]")
              ax.legend();
                                       PD Results

    Kv=10 simulated

                                                     Kv=10 actual

    Kv=100 simulated

    Kv=100 actual

    Kv=1000 simulated

                                                     - Kv=1000 actual
                                                  2.5
                                                        3.0
                                1.0
                                      1.5
                                            2.0
              Arduino Code
|_| |_|\___/\___/|_|\____| \___/ class Motor{ public: /* This is a class meant to implement a PD or operational space controller for a motor connected to a robot arm.
args: pwm_pin: Pin number that controls the voltage supplied to the motor. brk_pin: Pin number that cuts off the motor (LOW means the motor turns). dir_pin: Pin number that
controls the direction of the pin. COUNTER_CLOCKWISE_: HIGH or LOW, this defines what the dir_pin is set to when the motor is asked to turn counter clockwise.
CLOCKWISE_: HIGH or LOW, this defines what the dir_pin is set to when the motor is asked to go clockwise. enc_pin_A: Pin that encoder channel A is plugged into (must be an
interrupt pin) enc pin B: Pin that encoder challen B is plugged into (must be an interrupt pin) encoder dir : Set to either 1 or -1, used to correct the reading for the angle. If a
counter clockwise rotation causes the encoder to read negative, flip the value from -1 to 1 or 1 to -1 depending on what it is currently set to. */ //
CLOCKWISE_, int enc_pin_A, int enc_pin_B, int encoder_dir_) // set default values :m_encoder(Encoder(enc_pin_A, enc_pin_B)), // instantiates the encoder
encoder_dir{encoder_dir_}, // sets the encoder corrector m_pwm_pin{pwm_pin}, // sets the motor pwm pin m_brk_pin{brk_pin}, // sets the motor brk pin m_dir_pin{dir_pin}, // sets
the motor pwm pin COUNTER_CLOCKWISE{COUNTER_CLOCKWISE_}, // sets the CCW variable CLOCKWISE{CLOCKWISE_} // sets the CC variable // perform default
function calls { // set the pin modes pinMode(m_pwm_pin, OUTPUT); pinMode(m_brk_pin, OUTPUT); pinMode(m_dir_pin, OUTPUT); // set pin values analogWrite(m_pwm_pin,
0); digitalWrite(m_brk_pin, LOW); // if it turns clockwise by mistake, flip the HIGH and LOW assignments digitalWrite(m_dir_pin, COUNTER_CLOCKWISE); } //
bodyB length{0.02727751}; // m double bodyB mass{7.28e-3}; // kg double bodyB izz{6.39e-6}; // kg*m^2 double Kp{478}; // proportional gain double Kv{100}; // derivative gain
double K{5e-1}; // Voltage booster // pin variables int m_pwm_pin{0}; const int m_brk_pin; int m_dir_pin{0}; int m_dir_state_pin{0}; uint8_t CLOCKWISE{LOW}; uint8_t
COUNTER_CLOCKWISE{HIGH}; int encoder_dir{1}; // encoder variables Encoder m_encoder; // controller variables double q[2] = {0, 0}; // rad double q_desired{0}; // rad double
qdot = {0}; // rad/s double U{0}; // units/s^2 double Tau{0}; // Nm double Vemf{0}; // Volts double Vs{0}; // Volts // motor/encoder constants int CPR{1000}; // double ki = (10.52 -
0.108605472*4.5)/45; double ki = 2.29183; // double kt = (0.980665/2.666666660); double kt = 0.891514; // float R = 4.5; float R = 7.5; //
angle rotations void set_direction(int dir){ if (dir == 1){ digitalWrite(m_dir_pin, COUNTER_CLOCKWISE); } else if (dir == -1){ digitalWrite(m_dir_pin, CLOCKWISE); } // returns the
current angle of the motor in radians double angle(){ return (double)(encoder dir*2*M PI*m encoder.read()/CPR); } // kills the motor in the event that the probe is tripped,
preventint // damage to the robot. Is completely optional, and should be set in the // setup loop of the ardino code using a lambda static void probe(int probe_pin){
digitalWrite(probe_pin, HIGH); } // updates the angles to where q1 is always the most recent angle void update_angles(){ // q[1] is the newest q[0] = q[1]; q[1] = angle(); } //
update angles(); // Serial.print(", g1 = "); // Serial.println(g[1]); // calculate gdot gdot = (double)(1000*(g[1] - g[0])/millis); // Serial.println(gdot); // calculate back emf Vemf = ki*gdot; //
```

Serial.println(Vemf); // calculate U using PID controller U = K\*((Kv\*(0-qdot)) + (Kp\*(q\_desired - q[1]))); // Serial.println(U); // calculate Tau from controller value // Tau =

// output it to the motor analogWrite(right\_motor.m\_pwm\_pin, V); // overwrite the old time old\_time = new\_time; }

((bodyB\_length\*bodyB\_length\*bodyB\_mass) + (20736\*bodyA\_izz) + bodyB\_izz)\*U; Tau = U; // Serial.println(Tau, 2); // calculate Vs Vs = (Tau\*R/kt) + Vemf; // Serial.println(Vs); // toggle the direction if needed set\_direction(1); if (Vs < 0){ Vs = -Vs; set\_direction(-1); } // constrain the voltage Vs = constrain(Vs, 0, 12); // Serial.println(Vs); // map the voltage Vs =