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))
```

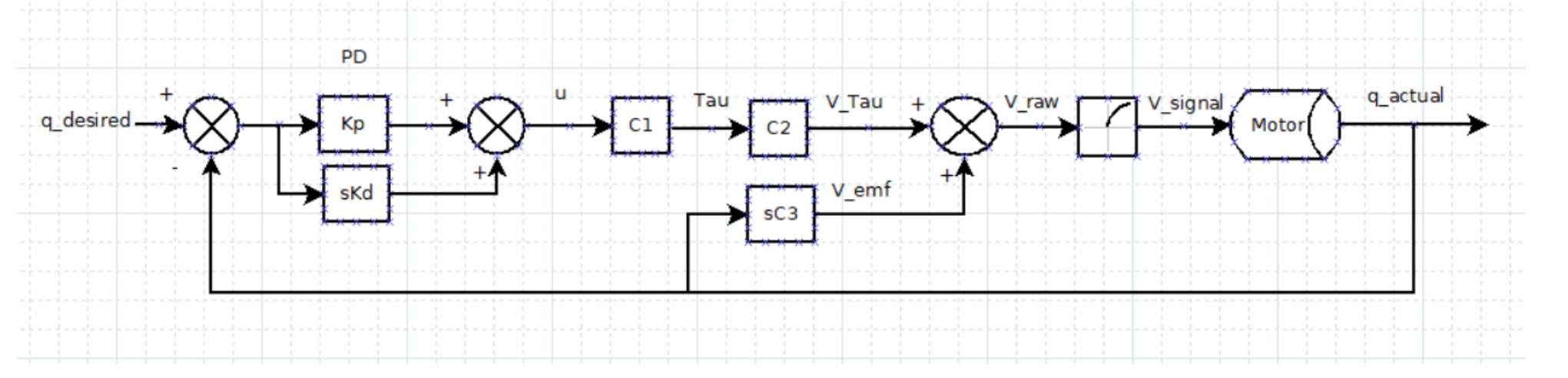
$$TM = \left[\left(L_B^2 bodyB_{mass} + bodyA_{izz} gear_{ratio}^2 + bodyB_{izz}\right)\ddot{q}_1 \right]$$

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,         # a.u
}
```

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))
u
```

Out[22]: $Kp\left(-q_{1current} + q_{1desired}\right) + Kv\left(-q1d_{current} + q1d_{desired}\right)$

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

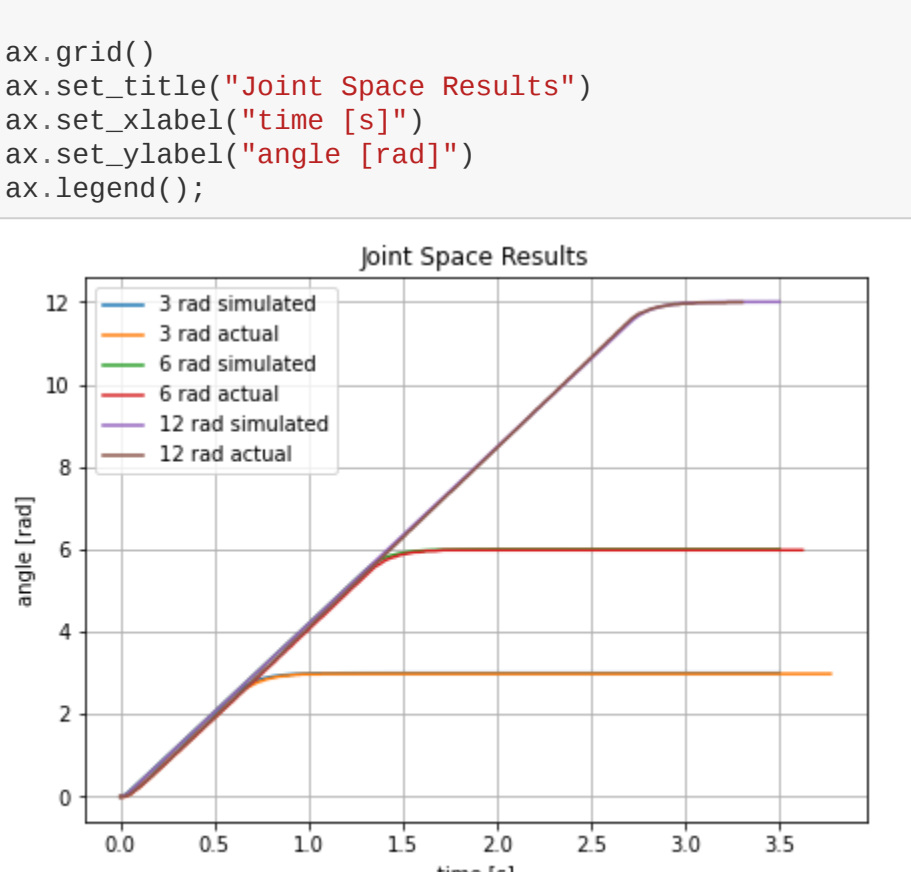
In [29]:
```
# create motor torque callable to drive q1dd
rpms = (0, 40)
taus = (0.980665, 0.4413)
slope, intercept = polyfit(rpms, taus, 1)

def tau_motor(x):
    if rpms[0] <= x <= rpms[1]:
        return (slope*x) + intercept
    else:
        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

    # 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
    else:
        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

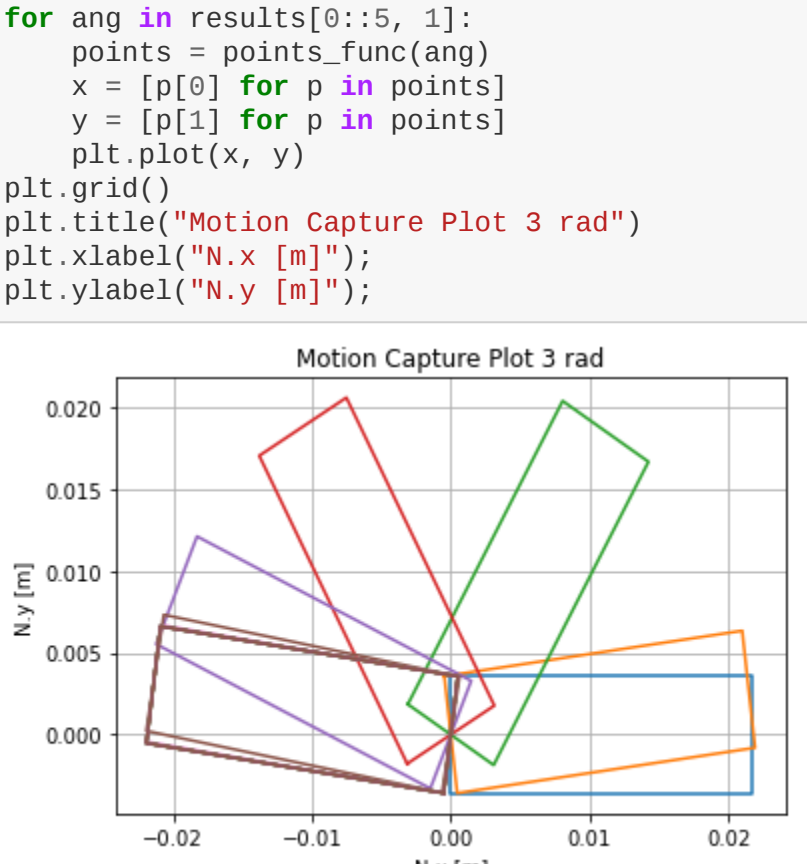
In [31]:
```
# plot the results from the 3 second simulation and experimentation
fig, ax = plt.subplots(figsize=(7, 5))
with open("joint_space_3rad.csv", "r") as f:
    res = [l.split(",") for l in f.readlines()]
    times = [float(v[0]) for v in res]
    angles = [float(v[1]) for v in res]
    ax.plot(times, angles, label="3 rad simulated")

with open("joint_space_3rad_actual.csv", "r") as f:
    res = [l.split(",") for l in f.readlines()]
    times = [float(v[0])/1000 for v 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 = [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="6 rad simulated")

with open("joint_space_6rad_actual.csv", "r") as f:
    res = [l.split(",") for l in f.readlines()]
    times = [float(v[0])/1000 for v 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 = [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="12 rad simulated")

with open("joint_space_12rad_actual.csv", "r") as f:
    res = [l.split(",") for l in f.readlines()]
    times = [float(v[0])/1000 for v 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();
```



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]");
```



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.

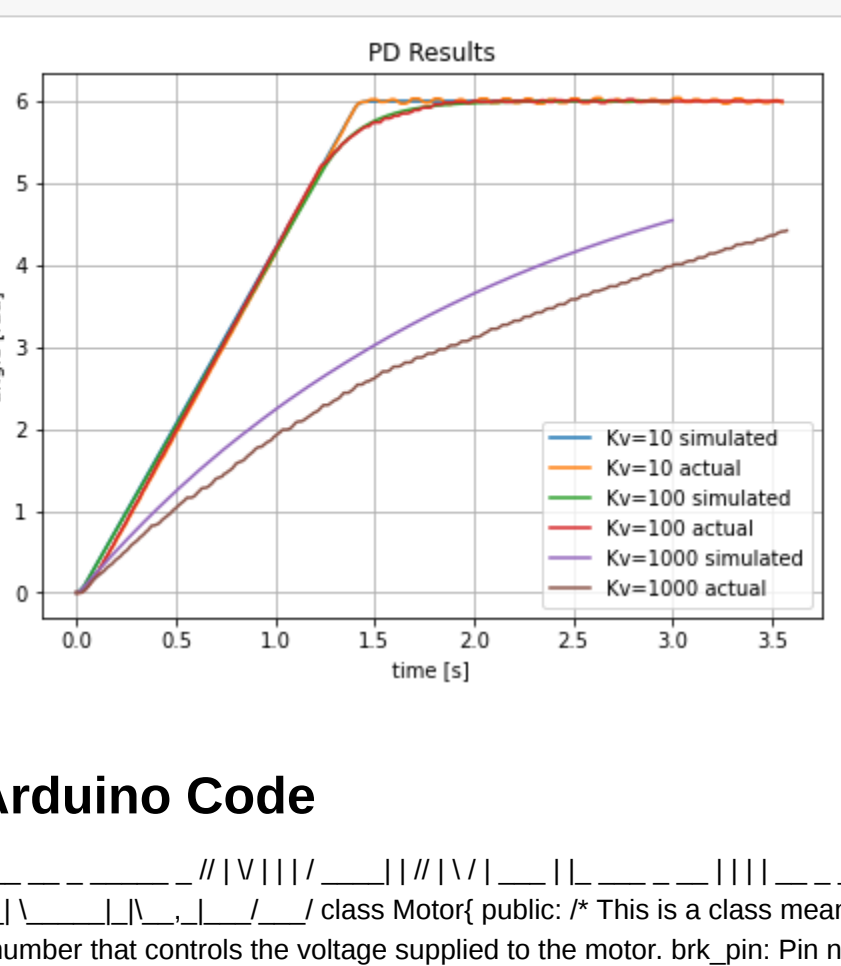
In [34]:
```
# create all the callables
simulation_parameters = {
    q1_desired: 6,
    q1d_desired: 0,
    Kp: int(Kp_max),
    Kv: 10
}
```

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 = [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=10 simulated")

with open("PD_10_actual.csv", "r") as f:
    res = [l.split(",") for l in f.readlines()]
    times = [float(v[0])/1000 for v 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 = [l.split(",") for l in f.readlines()]
    times = [float(v[0])/1000 for v 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 = [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=1000 simulated")

with open("PD_1000_actual.csv", "r") as f:
    res = [l.split(",") for l in f.readlines()]
    times = [float(v[0])/1000 for v 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();
```



Arduino Code

#include #include // __ __ _ ______ _ // | \/ | | | / ____| | // | \ / | ___ | |_ ___ _ __ | | | | __ _ ___ ___ // | |\/| |/ _ \| __/ _ \| '__| | | | |/ _` / __/ __| // | | | | (_) | || (_) | | | |____| | (_| __ __ \ // |_| |_|___/ _____/|_| _____|_|__,_|___/___/ 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. */ // ============================ constructor ============================ Motor(int pwm_pin, int brk_pin, int dir_pin, uint8_t COUNTER_CLOCKWISE_, uint8_t 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); } // ============================== variables ============================== // safe to playwith controller variables double bodyA_izz{3.3e-7}; // kg double 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 = (0.980665/2.6666666660); double kt = 0.891514; // float R = 4.5; float R = 7.5; // ================================ functions ================================ // 1 for counter clockwise, -1 for clockwise // same as the convection for positive 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(); } // =============================== controller ================================ // implements the controller double calculate_Vs(long millis){ // get the newest angles update_angles(); // Serial.print(", q1 = "); // Serial.println(q[1]); // calculate qdot qdot = (double)(1000*(q[1] - q[0])/millis); // Serial.println(qdot); // calculate back emf Vemf = ki*qdot; // 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 = ((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 = map(Vs, 0, 12, 0, 255); // Serial.println(Vs) return Vs; } }; // _____ _ _ _ // / ____| | | | | | // | | __| | ___ | |__ ___ _| |___ // | | |_ | |/ _ \| '_ \ / _` | / __| // | |__| | | (_) | |_) | (_| | __ \ // _____|_|___/|_.__/ __,_|_|___/ // time variables long new_time{0}; long old_time{0}; long dt{0}; float V{0}; // define motors Motor right_motor(3, 9, 12, HIGH, LOW, 20, 21, 1); // _ _ _____ _ // /\ | | (_) / ____| | | // / \ _ __ __| |_ _ _ _ __ ___ | (___ ___| |_ _ _ _ __ // / /\ \ | '__/ _` | | | | | '_ \ / _ \ ___ \ / _ \ __| | | | '_ \ // / ____ \| | | (_| | |_| | | | | | (_) | ____) | __/ |_| |_| | | |_) | // /_/ __| __,_|__,_|_|_| |_|___/ |_____/ ___|__|__,_| .__/ // | | // |_| void setup(){ // start the coms Serial.begin(9600); // setup the probe that trips if the robot is going to break something pinMode(2, INPUT); attachInterrupt(digitalPinToInterrupt(2), [](){Motor::probe(right_motor.m_brk_pin);}, HIGH); // set desired angle right_motor.q_desired = 6; // digitalWrite(right_motor.m_brk_pin, HIGH); } // _ _ _ // /\ | | (_) | | // / \ _ __ __| |_ _ _ _ __ ___ | | ___ ___ _ __ // / /\ \ | '__/ _` | | | | | '_ \ / _ \ | | / _ \ / _ \| '_ \ // / ____ \| | | (_| | |_| | | | | | (_) | |___| (_) | (_) | |_) | // /_/ __| __,_|__,_|_|_| |_|___/ |_________/ ___/| .__/ // | | // |_| void loop(){ // capture the new time new_time = millis(); // print out csv for Exercise 3 Serial.print(new_time); Serial.print(","); Serial.println(right_motor.q[1]); // calculate dt dt = (new_time - old_time); // calculate Vs V = right_motor.calculate_Vs(dt); // Serial.println(V); // output it to the motor analogWrite(right_motor.m_pwm_pin, V); // overwrite the old time old_time = new_time; }