Starter code for evaluating compliance control

- This code is a boilerplate version to visualize the current commands we will send to the motor when it is in various configurations.
- There is only one place for you to edit to visualize this, you are to add in code to the compliancea_control function at the bottom.
- When this is completed, you can run the last script to move the leg to different xy locations and compute the motor currents.

In [ ]:

```
import numpy as np

import matplotlib as mpl
mpl.use('Qt5Agg')

import matplotlib.pyplot as plt
plt.ion()

# for the symbolic manipulation of jacobian
import sympy as sp
# from sympy import symbols
from sympy import sin, cos, asin, acos, pi, atan2, sqrt
from sympy.utilities.lambdify import lambdify
# from sympy import Matrix

from scipy.optimize import minimize
from scipy.optimize import fsolve

import time

import odrive
import odrive.utils
import odrive.enums
```

In [ ]:

```
## Motor constants
K_T = 0.0285;                    # Nm / A
peak_amp = 30;                 # A
peak_torque = K_T * peak_amp; # Nm

l1 = 0.09;                     # m
l2 = 0.16;                     # m
w = 0.07;                      # m
```

In [ ]:

```
def T(theta, x, y):
    """
    Function to return an arbitrary transformation matrix
    This is for sympy symbolic calculation
    """
    return sp.Matrix([[sp.cos(theta), -sp.sin(theta), x],
                      [sp.sin(theta), sp.cos(theta), y],
                      [0, 0, 1]])

def sym_to_np(T):
    return np.array(T).astype(np.float64)
```

FK Through transformation matrices

In [ ]:

```
(thetaL, sym,
```

```
 thetaR_sym,
 link1_sym,
 link2_sym,
 width_sym) = sp.symbols("""thetaL_sym
                            thetaR_sym
                            link1_sym
                            link2_sym
                            width_sym""", real = True)
```

In [ ]:

```
x_r = width_sym/2 + link1_sym*sp.cos(thetaR_sym)
x_l = -width_sym/2 + link1_sym*sp.cos(thetaL_sym)

y_r = link1_sym*sp.sin(thetaR_sym)
y_l = link1_sym*sp.sin(thetaL_sym)

theta3_sym = sp.atan2(y_r - y_l, x_r - x_l)
L = sp.sqrt((x_l - x_r)**2 + (y_l - y_r)**2)

FK = T(thetaL_sym, -width_sym/2, 0)@T(-(thetaL_sym - theta3_sym), link1_sym, 0)@sp.Matrix([L/2, sp.sqrt
(link2_sym**2 - (L/2)**2), 1])
FK = FK[:2,:]
FK.simplify()

#cartesian
FK = FK.subs([(link1_sym,l1), (link2_sym,l2), (width_sym,w)])
J = FK.jacobian((thetaR_sym, thetaL_sym))
J_fast = lambdify((thetaR_sym, thetaL_sym), J)
FK_fast = lambdify((thetaR_sym, thetaL_sym), FK)
```

In [ ]:

```
xy = FK_fast(np.pi/2, np.pi/2)

J_new = J_fast(np.pi/2, np.pi/2)
```

In [ ]:

```
#polar

FK_polar = sp.Matrix([sp.sqrt(FK[0,0]**2 + FK[1,0]**2), sp.atan2(FK[0,0], FK[1,0])])
FK_polar_fast = lambdify((thetaR_sym, thetaL_sym), FK_polar)
J_polar = FK_polar.jacobian([thetaR_sym, thetaL_sym]).evalf()
J_pol_fast = lambdify((thetaR_sym, thetaL_sym), J_polar)
```

IK through optimization

In [ ]:

```
def IK_5_link(x, y, l1 = l1, l2 = l2, w = w):

    def leg_wide(var):
        return np.linalg.norm([var[0], var[1] - np.pi])

    def x_constraint_equation(var):
        # should be equal to zero when the
        return l1**2 - l2**2 + (x - w/2)**2 + y**2 - 2*l1*(y*np.sin(var[0]) + (x - w/2)*np.cos(var[0]))

    def y_constraint_equation(var):
        return l1**2 - l2**2 + (x + w/2)**2 + y**2 - 2*l1*(y*np.sin(var[1]) + (x + w/2)*np.cos(var[1]))


    res = minimize(leg_wide, (0.1, 9*np.pi/10), method="SLSQP", constraints= ({"type": "eq", "fun": x_c
onstraint_equation},
                                                                                  {"type": "eq", "fun": y_
constraint_equation}))
```

```
    return (res, np.linalg.norm([x_constraint_equation(res.x), y_constraint_equation(res.x)]))
```

In [ ]:

```
def internal_angles(thetaR, thetaL, l1 = l1, l2 = l2, w = w):
    """
    Solves for the internal angles of the leg so that we can visualize
    """
    def sys(x):
        return (w + l1*np.cos(thetaR) + l2*np.cos(x[0]) - l1*np.cos(thetaL) - l2*np.cos(x[1]),
                l1*np.sin(thetaR) + l2*np.sin(x[0]) - l1*np.sin(thetaL) - l2*np.sin(x[1]))

    alphaR, alphaL = fsolve(sys, (np.pi/2, np.pi/2))

    alphaR = alphaR % (2*np.pi)
    alphaL = alphaL % (2*np.pi)

    # Copmute FK for checking
    x = w/2 + l1*np.cos(thetaR) + l2*np.cos(alphaR);
    y = l1*np.sin(thetaR) + l2*np.sin(alphaR);

    return (alphaR, alphaL, x, y)

thetaR = .5
thetaL = np.pi

(alphaR, alphaL, x, y) = internal_angles(thetaR, thetaL)

# Should produce
# alphaL
# Out[17]: 0.8878073988680342

# alphaR
# Out[18]: 2.611036674795031
```

In [ ]:

```
def draw_robot(thetaR, thetaL, link1 = l1, link2 = l2, width = w, ax = None):

    # Solve for internal angles
    (alphaR, alphaL, x, y) = internal_angles(thetaR, thetaL)

    def pol2cart(rho, phi):
        x = rho * np.cos(phi)
        y = rho * np.sin(phi)
        return(x, y)

    if ax is None:
        ax = plt.gca()
        ax.cla()


    ax.plot(-width/2, 0, 'ok')
    ax.plot(width/2, 0, 'ok')

    ax.plot([-width/2, 0], [0, 0], 'k')
    ax.plot([width/2, 0], [0, 0], 'k')

    ax.plot(-width/2 + np.array([0, link1*np.cos(thetaL)]), [0, link1*np.sin(thetaL)], 'k')
    ax.plot(width/2 + np.array([0, link1*np.cos(thetaR)]), [0, link1*np.sin(thetaR)], 'k')

    ax.plot(-width/2 + link1*np.cos(thetaL) + np.array([0, link2*np.cos(alphaL)]), \
             link1*np.sin(thetaL) + np.array([0, link2*np.sin(alphaL)]), 'k');

    ax.plot(width/2 + link1*np.cos(thetaR) + np.array([0, link2*np.cos(alphaR)]), \
             np.array(link1*np.sin(thetaR) + np.array([0, link2*np.sin(alphaR)])), 'k');

    ax.plot(x, y, 'ro');
# plt.set_aspect
    ax.set_aspect('equal')
#     plt.plot(x_end, y_end, 'go');

    ax.axis([-.3,.3,-.1,.3])
```

```
thetaR = np.pi/4
thetaL = 3*np.pi/4

draw_robot(thetaR, thetaL)
```

Starter point for compliance control

In [ ]:

```
# x_eq = 0
# y_eq = 0.1896167

# k_x = 10
# k_y = 10

# k = np.array([[k_x, 0],
#                [0, k_y]])

def cartesian_compliance(x_disp, y_disp, J, theta_dot, C, K_T=0.0285):
    """
    Implement the cartesian controller in this function.
    This should return the motor currents as an array (i.e. the output of the matrix equation given in
class)
    """
    kx=2000
    ky=2000
    disp = np.array([[x_disp],[y_disp]])
    #print(disp.shape)
    k_matrix = np.array([[-1*kx, 0],[0, -1*ky]])
    velocity = np.dot(J,theta_dot)
    taus = np.dot(J.T, np.dot(k_matrix, disp[:,:,0]) - C*velocity)
    currents = taus/K_T

    currents[0] = min(currents[0], 30)
    currents[1] = min(currents[1], 30)

    return currents

# cartesian_compliance(0,0)
```

In [ ]:

```
# IDLE
odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_IDLE
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_IDLE
```

In [ ]:

```
## PUT LEG IN 0 CONFIGURATION
zero_position = (odrv0.axis0.encoder.pos_estimate+4096, odrv0.axis1.encoder.pos_estimate)
print(zero_position)

equilibrium_pos = (zero_position[0] - 3072, zero_position[1] - 1024) #m0, m1
```

In [ ]:

```
odrive.utils.dump_errors(odrv0, True)
odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL
odrv0.axis0.controller.set_pos_setpoint(equilibrium_pos[0],0,0)
odrv0.axis1.controller.set_pos_setpoint(equilibrium_pos[1],0,0)
```

In [ ]:

```
theta_eq = get_joints(odrv0, zero_pos = zero_position)

# xy_eq = FK.subs([(thetaL_sym, 1*theta_eq[0]), (thetaR_sym, 1*theta_eq[1]), (link1_sym, l1), (link2_sym
```

```
# xy_eq = FK.subs([(thetaL_sym,-1*theta_eq[0]), (thetaR_sym,-1*theta_eq[1]), (link1_sym,l1), (link2_sym
,l2), (width_sym,w)])
# print(xy_eq)
xy_eq = FK_fast(-1*theta_eq[1], -1*theta_eq[0])
print(xy_eq)
```

In [ ]:

```
thetaL, thetaR = get_joints(odrv0, zero_pos = zero_position)
xy_new = FK.subs([(thetaL_sym,thetaL), (thetaR_sym,thetaR), (link1_sym,l1), (link2_sym,l2), (width_sym,
w)])
print(xy_new)
```

In [ ]:

```
start = time.time()
time_passed=0


C=60
#time.sleep(2)
while time_passed < 30:

    thetaL, thetaR = get_joints(odrv0, zero_pos = zero_position)
    #xy_new = FK.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l2),
(width_sym,w)])
    xy_new = FK_fast(-1*thetaR, -1*thetaL)

    x_disp = xy_new[0] - xy_eq[0]
    y_disp = xy_new[1] - xy_eq[1]

    #J_new = J.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l2), (w
idth_sym,w)])
    J_new = J_fast(-1*thetaR, -1*thetaL)
    theta_R_dot = -1*odrv0.axis1.encoder.vel_estimate * 2 * np.pi / 8192
    theta_L_dot = -1*odrv0.axis0.encoder.vel_estimate * 2 * np.pi / 8192

    theta_dot = np.array([[theta_R_dot],[theta_L_dot]])
    current = cartesian_compliance(x_disp, y_disp, np.array(J_new).astype(np.float64), theta_dot, C, K_
T=2.85)
    current0 = current[0]
    current1 = current[1]


    odrv0.axis0.controller.set_current_setpoint(-1*current1)
    odrv0.axis1.controller.set_current_setpoint(-1*current0)
    time_passed = time.time() - start


odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_IDLE
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_IDLE
```

In [ ]:

```
def polar_compliance(r, theta_disp, theta_dot, J_polar):
    """

    Implement the polar controller in this function.
    This should return the motor currents as an array (i.e. the output of the matrix equation given in
class)
    """

    K_T = 2.85;                 # Nm / A
    peak_amp = 30;              # A
    peak_torque = K_T * peak_amp; # Nm

    Kr = 2000
    Kt = 0
    C = 5

    disp = np.array([[r],[theta_disp]])
    k_matrix = np.array([[-1*Kr, 0],[0, -1*Kt]])
```

```
    k_matrix = np.array([[-1*Kr, 0],[0, -1*Kt]])
    velocity_r = np.dot(J_polar,theta_dot)

    #velocity_r = np.array([[np.sqrt(velocity[0,0]**2 + velocity[1,0]**2)],[np.arctan2(velocity[0,0],ve
locity[1,0])]])

    #J_polar = np.array([[np.sign(J[0,0])*np.sqrt(J[0,0]**2 + J[1,0]**2), np.sign(J[0,1])*np.sqrt(J[0,1
]**2 + J[1,1]**2)]
    #                    ,[np.arctan2(J[0,0], J[1,0]), np.arctan2(J[0,1], J[1,1])]])


    taus = np.dot(k_matrix, disp) - C*velocity_r
    taus = np.dot(J_polar.T, taus)

    currents = taus/K_T


    if currents[0] < 30:
        if currents[0] < -30:
            currents[0] = -30
    else:
        currents[0] = 30

    if currents[1] < 30:
        if currents[1] < -30:
            currents[1] = -30
    else:
        currents[1] = 30


    return currents
```

In [ ]:

```
odrive.utils.dump_errors(odrv0, True)
odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL
odrv0.axis0.controller.set_pos_setpoint(equilibrium_pos[0],0,0)
odrv0.axis1.controller.set_pos_setpoint(equilibrium_pos[1],0,0)
```

In [ ]:

```
theta_eq = get_joints(odrv0, zero_pos = zero_position)

xy_eq = FK.subs([(thetaL_sym,-1*theta_eq[0]), (thetaR_sym,-1*theta_eq[1]), (link1_sym,l1), (link2_sym,l
2), (width_sym,w)])

r_eq = np.sqrt(float(xy_eq[0])**2 + float(xy_eq[1])**2)
theta_eq = np.arctan2(float(xy_eq[0]), float(xy_eq[1]))

print(r_eq)
print(theta_eq)
```

In [ ]:

```
start = time.time()
time_passed=0

#time.sleep(2)
while time_passed < 30:

    thetaL, thetaR = get_joints(odrv0, zero_pos = zero_position)
    #xy_new = FK.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l2),
(width_sym,w)])
    xy_new = FK_fast(thetaR*-1, thetaL * -1)

    r_new = np.sqrt(float(xy_new[0])**2 + float(xy_new[1])**2)
    theta_new = np.arctan2(float(xy_new[0]), float(xy_new[1]))

    r_disp = r_new - r_eq
```

```
    theta_disp = theta_new - theta_eq

    #J_new = J.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l2), (w
idth_sym,w)])
    J_new = J_fast(thetaR*-1, thetaL * -1)

    theta_R_dot = -1*odrv0.axis1.encoder.vel_estimate * 2 * np.pi / 8192
    theta_L_dot = -1*odrv0.axis0.encoder.vel_estimate * 2 * np.pi / 8192

    theta_dot = np.array([[theta_R_dot],[theta_L_dot]])
    #J_pol = J_polar.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l
2), (width_sym,w)])
    J_pol = J_pol_fast(thetaR*-1, thetaL * -1)
    current = polar_compliance(r_disp, theta_disp, theta_dot, np.array(J_pol).astype(np.float64))
    current0 = current[0]
    current1 = current[1]

    #print(current0, current1)
    odrv0.axis0.controller.set_current_setpoint(-1*current1)
    odrv0.axis1.controller.set_current_setpoint(-1*current0)
    time_passed = time.time() - start


odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_IDLE
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_IDLE
```

In [ ]:

```
# OLD tests

start = time.time()
time_passed=0

thetaL, thetaR = get_joints(odrv0, zero_pos = zero_position)
rtheta_eq = FK_polar.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l
2), (width_sym,w)])
r_eq = rtheta_eq[0]
theta_eq = rtheta_eq[1]

C=0.001
#time.sleep(2)
while time_passed < 20:

    thetaL, thetaR = get_joints(odrv0, zero_pos = zero_position)
    rtheta_new = FK_polar.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_
sym,l2), (width_sym,w)])

    #x_disp = float(xy_new[0] - xy_eq[0])
    #y_disp = float(xy_new[1] - xy_eq[1])

    r_new = rtheta_new[0] #np.sqrt(float(xy_new[0])**2 + float(xy_new[1])**2)
    #r_eq = np.sqrt(float(xy_eq[0])**2 + float(xy_eq[1])**2)
    r_disp = r_new - r_eq
    print(r_disp)
    time.sleep(2)
    theta_new = rtheta_new[1]  #np.arctan2(float(xy_new[0]), float(xy_new[1]))
    #theta_eq = np.arctan2(float(xy_eq[0]), float(xy_eq[1]))
    theta_disp = theta_new - theta_eq
    J_new = J_polar.subs([(thetaL_sym,thetaL*-1), (thetaR_sym,thetaR*-1), (link1_sym,l1), (link2_sym,l2
), (width_sym,w)])
#

    theta_R_dot = -1*odrv0.axis1.encoder.vel_estimate * 2 * np.pi / 8192
    theta_L_dot = -1*odrv0.axis0.encoder.vel_estimate * 2 * np.pi / 8192

    theta_dot = np.array([[theta_R_dot],[theta_L_dot]])

    current = polar_compliance(r_disp, theta_disp, theta_dot, 20, 20, C, np.array(J_new).astype(np.floa
t64))
    #current = cartesian_compliance(x_disp, y_disp, np.array(J_new).astype(np.float64), theta_dot, C)
    current0 = current[0]
    current1 = current[1]

#     theta_R_dot = -1*odrv0.axis1.encoder.vel_estimate * 2 * np.pi / 8192
```

```
#     theta_L_dot = -1*odrv0.axis0.encoder.vel_estimate * 2 * np.pi / 8192
#     theta_dot = np.array([[theta_R_dot],[theta_L_dot]])



#     odrv0.axis0.controller.set_current_setpoint(-1*current1)
#     odrv0.axis1.controller.set_current_setpoint(-1*current0)
    time_passed = time.time() - start
    #print(time_passed)


#odrv0.axis0.controller.set_current_setpoint(0)
#odrv0.axis1.controller.set_current_setpoint(0)
odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_IDLE
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_IDLE
```

In [ ]:

```
# OLD tests

odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL

start = time.time()
time_passed=0

encoder_eq = odrv0.axis1.encoder.pos_estimate


while time_passed < 10:

    encoder_L = odrv0.axis1.encoder.pos_estimate

    theta_dot = odrv0.axis1.encoder.vel_estimate * 2 * np.pi / 8192

    delta_encoder = encoder_L - encoder_eq

    delta_theta = delta_encoder * 2 * np.pi / 8192

    current = polar_compliance(delta_theta, theta_dot, 2, 0.01)

    odrv0.axis1.controller.set_current_setpoint(current)

    time_passed = time.time() - start
    #print(time_passed)

odrv0.axis1.controller.set_current_setpoint(0)
```

In [ ]:

```
odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_CLOSED_LOOP_CONTROL

odrv0.axis0.controller.set_pos_setpoint(2000,0,0)
#odrv0.axis1.controller.set_pos_setpoint(-24236.765625,0,0)
```

In [ ]:

```
odrive.utils.dump_errors(odrv0)
```

In [ ]:

```
odrive.utils.dump_errors(odrv0, True)
```

In [ ]:

```
odrv0 = odrive.find_any()
```

```
if odrv0 is not None:
    print('Connected!')
    print('Odrive serial {}'.format(odrv0.serial_number))

    m0 = odrv0.axis0.motor.is_calibrated
    m1 = odrv0.axis1.motor.is_calibrated

    print('Motor 0 calibrated: {}'.format(m0))
    print('Motor 1 calibrated: {}'.format(m1))

else:
    print('Not connected')
```

In [ ]:

```
odrv0.axis0.controller.config.vel_limit = 200000
odrv0.axis1.controller.config.vel_limit = 200000
```

In [ ]:

```
odrv0.axis0.requested_state = odrive.enums.AXIS_STATE_FULL_CALIBRATION_SEQUENCE
odrv0.axis1.requested_state = odrive.enums.AXIS_STATE_FULL_CALIBRATION_SEQUENCE

time.sleep(15)

print('\t Motor 0 calibration result: {} \r\n'.format(odrv0.axis0.motor.is_calibrated),
      '\t Motor 1 calibration result: {}'.format(odrv0.axis1.motor.is_calibrated))
```

In [ ]:

```
odrv0.axis0.controller.set_current_setpoint(current_0)
odrv0.axis1.controller.set_current_setpoint(current_1)
```

In [ ]:

```
motor_cpr = (odrv0.axis0.encoder.config.cpr,
             odrv0.axis1.encoder.config.cpr)

print('encoder0: ', odrv0.axis0.encoder.pos_estimate)
print('encoder1: ', odrv0.axis1.encoder.pos_estimate)

def convert_joints(angles, cpr=motor_cpr, zero_pos = (0,0)):
    encoder_vals = (angles * cpr[0] / (2 * np.pi)) + zero_pos
    return encoder_vals
```

In [ ]:

```
motor_cpr = (odrv0.axis0.encoder.config.cpr,
             odrv0.axis1.encoder.config.cpr)

def get_joints(odrv, cpr = motor_cpr, zero_pos = (3863.234375,-26339.015625+4096)):
    m0 = 2*np.pi*(odrv.axis0.encoder.pos_estimate - zero_pos[0])/motor_cpr[0]
    m1 = 2*np.pi*(odrv.axis1.encoder.pos_estimate - zero_pos[1])/motor_cpr[1]

    return (m0, m1)

print(get_joints(odrv0))
```

In [ ]:

```
motor_cpr = (odrv0.axis0.encoder.config.cpr,
             odrv0.axis1.encoder.config.cpr)

def 1get_joints(odrv, cpr = motor_cpr, zero_pos = (3540.75 - 4096, -26342.015625)):
    m0 = 2*np.pi*(odrv.axis0.encoder.pos_estimate - zero_pos[0])/motor_cpr[0]
    m1 = 2*np.pi*(odrv.axis1.encoder.pos_estimate - zero_pos[1])/motor_cpr[1]
```

```
    return (m0, m1)

print(get_joints(odrv0))
```