```
import numpy as np

def integrate(f, xt, dt):
    """
    This function takes in an initial condition x(t) and a timestep dt,
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x(t). It outputs a vector x(t+dt) at the future
    time step.

    Parameters
    ============
    dyn: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    xt: NumPy array
        current step x(t)
    dt:
        step size for integration

    Return
    ============
    new_xt:
        value of x(t+dt) integrated from x(t)
    """
    k1 = dt * f(xt)
    k2 = dt * f(xt+k1/2.)
    k3 = dt * f(xt+k2/2.)
    k4 = dt * f(xt+k3)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

def simulate(f, x0, tspan, dt, integrate):
    """
    This function takes in an initial condition x0, a timestep dt,
    a time span tspan consisting of a list [min_time, max_time],
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x0. It outputs a full trajectory simulated
    over the time span of dimensions (xvec_size, time_vec_size).

    Parameters
    ============
    f: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    x0: NumPy array
        initial conditions
    tspan: Python list
        tspan = [min_time, max_time], it defines the start and end
        time of simulation
    dt:
        time step for numerical integration
    integrate: Python function
        numerical integration method used in this simulation

    Return
    ============
```

```
    x_traj:
        simulated trajectory of x(t) from t=0 to tf
    """
    N = int((max(tspan)-min(tspan))/dt)
    x = np.copy(x0)
    tvec = np.linspace(min(tspan),max(tspan),N)
    xtraj = np.zeros((len(x0),N))
    for i in range(N):
        xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return xtraj

def animate_double_pend(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ===========================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    L2:
        length of the second pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation.
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML
    import plotly.graph_objects as go

    ######################
    # Browser configuration.
    def configure_plotly_browser_state():
        import IPython
        display(IPython.core.display.HTML('''
            <script src="/static/components/requirejs/require.js"></script>
            <script>
              requirejs.config({
                paths: {
                  base: '/static/base',
                  plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
                },
              });
            </script>
            '''))
    configure_plotly_browser_state()
    init_notebook_mode(connected=False)

    ##############################################
    # Getting data from pendulum angle trajectories.
    xx1=L1*np.sin(theta_array[0])
```

```
    yy1=-L1*np.cos(theta_array[0])
    xx2=xx1+L2*np.sin(theta_array[0]+theta_array[1])
    yy2=yy1-L2*np.cos(theta_array[0]+theta_array[1])
    N = len(theta_array[0]) # Need this for specifying length of simulation

    ###################################
    # Using these to specify axis limits.
    xm=np.min(xx1)-0.5
    xM=np.max(xx1)+0.5
    ym=np.min(yy1)-2.5
    yM=np.max(yy1)+1.5

    ###########################
    # Defining data dictionary.
    # Trajectories are here.
    data=[dict(x=xx1, y=yy1,
               mode='lines', name='Arm',
               line=dict(width=2, color='blue')
              ),
          dict(x=xx1, y=yy1,
               mode='lines', name='Mass 1',
               line=dict(width=2, color='purple')
              ),
          dict(x=xx2, y=yy2,
               mode='lines', name='Mass 2',
               line=dict(width=2, color='green')
              ),
          dict(x=xx1, y=yy1,
               mode='markers', name='Pendulum 1 Traj',
               marker=dict(color="purple", size=2)
              ),
         dict(x=xx2, y=yy2,
               mode='markers', name='Pendulum 2 Traj',
               marker=dict(color="green", size=2)
              ),
        ]

    ###############################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor = "x",dtick=1),
                title='Double Pendulum Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame': {'duration': T, 'redraw': False}}]},
                                           {'args': [[None], {'frame': {'duration': T, 'redraw': False}, 'mode': 'immediate',
                                            'transition': {'duration': 0}}],'label': 'Pause','method': 'animate'}
                                          ]
                              }]
                )

    ########################################
    # Defining the frames of the simulation.
    # This is what draws the lines from
    # joint to joint of the pendulum.
    frames=[dict(data=[dict(x=[0,xx1[k],xx2[k]],
```

```
                         y=[0,yy1[k],yy2[k]],
                         mode='lines',
                         line=dict(color='red', width=3)
                         ),
                     go.Scatter(
                         x=[xx1[k]],
                         y=[yy1[k]],
                         mode="markers",
                         marker=dict(color="blue", size=12)),
                     go.Scatter(
                         x=[xx2[k]],
                         y=[yy2[k]],
                         mode="markers",
                         marker=dict(color="blue", size=12)),
                    ]) for k in range(N)]

    ########################################
    # Putting it all together and plotting.
    figure1=dict(data=data, layout=layout, frames=frames)
    iplot(figure1)
```

PARAMs

```
##WEAPON BAR##
m1 = 0.5
l1 = 1
w1 = 0.25
inertia1 = (m1/12)*(l1**2 + w1**2)

m2 = 3
l2 = 1
w2 = 1
inertia2 =(m2/12)*(l2**2 + w2**2)

```

SETUP AND LAG AND EL

```
import sympy as sym
from sympy import symbols, Matrix ,Function, solve,cos,sin,simplify,Eq,transpose,zeros,eye
from sympy.abc import x, y, z,t,g,r,R,m,M,p
from sympy import sin, cos

import numpy as np
import matplotlib.pyplot as plt

theta1 = Function(r"theta1")(t)
theta2 = Function(r"theta2")(t)
x1 = Function(r"x_1")(t)
x2 = Function(r"x_2")(t)
y1 = Function(r"y_1")(t)
y2 = Function(r"y_2")(t)
```

```
theta1dot = theta1.diff(t)
theta2dot = theta2.diff(t)
x1dot = x1.diff(t)
x2dot = x2.diff(t)
y1dot = y1.diff(t)
y2dot = y2.diff(t)



q = Matrix([x1,y1,theta1,x2,y2,theta2])
qdot = Matrix([x1dot,y1dot,theta1dot,x2dot,y2dot,theta2dot])

#Rbw = Matrix([[sym.cos(theta1),-1*sym.sin(theta1)],[sym.sin(theta1),sym.cos(theta1)]])\
#FORMAT: t_SpaceDescribingIn_thingBeingDescribed

#BATTLEBOT
t_s_a = Matrix([[1, 0, x1],
                        [0,  1, y1],
                        [0, 0, 1]])

t_a_b = Matrix([[sym.cos(theta1), -sym.sin(theta1), 0],
                        [sym.sin(theta1),  sym.cos(theta1), 0],
                        [                        0,                           0, 1]])
t_s_b = t_s_a * t_a_b
#BLOCK
t_s_c = Matrix([[1, 0, x2],
                        [0,  1, y2],
                        [0, 0, 1]])

t_c_d = Matrix([[sym.cos(theta2), -sym.sin(theta2), 0],
                        [sym.sin(theta2),  sym.cos(theta2), 0],
                        [                        0,                           0, 1]])
t_s_d = t_s_c * t_c_d
display(t_s_d)
#######

homo = Matrix([0 ,0,1])
tcubedot = t_s_d.diff(t)
tweapdot = t_s_b.diff(t)

v1bar = tcubedot * homo
v2bar = tweapdot * homo
display(v1bar)
InertiaMat1 = Matrix([[1 ,0 ,0 ],[0, 1 ,0 ],[0, 0 ,inertia1]])
InertiaMat2 = Matrix([[1 ,0 ,0 ],[0, 1 ,0 ],[0, 0 ,inertia2]])

rotKE = 1/2 * inertia1 * theta1dot**2 +  1/2 * inertia2 * theta2dot**2
display(rotKE)
#rotKE = 1/2 * InertiaMat1 * theta1dot**2 +  1/2 * IntertiaMat2 * theta2dot**2
#rotKE = 0
#KE = 1/2 * m1 * v1bar.T* InertiaMat1 *v1bar + 1/2 * m2 * v2bar.T* InertiaMat2*v2bar
KE = 1/2 * m1 * v1bar.T*v1bar + 1/2 * m2 * v2bar.T*v2bar
KE = KE[0] + rotKE
display(KE)
PE = 0

Lag = KE - PE
#Calc Legrange
```

```
print("Lagrangian is:")
display(sym.simplify(Lag))
```

$$\begin{bmatrix} \cos\left(\theta_{2}(t)\right) & -\sin\left(\theta_{2}(t)\right) & \mathrm{x}_{2}\left(t\right) \\ \sin\left(\theta_{2}(t)\right) & \cos\left(\theta_{2}(t)\right) & \mathrm{y}_{2}\left(t\right) \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{d}{dt}\,\mathrm{x}_{2}\left(t\right) \\ \frac{d}{dt}\,\mathrm{y}_{2}\left(t\right) \\ 0 \end{bmatrix}$$

$$0.0221354166666667\left(\frac{d}{dt}\theta_{1}(t)\right)^{2} + 0.25\left(\frac{d}{dt}\theta_{2}(t)\right)^{2}$$

$$0.0221354166666667\left(\frac{d}{dt}\theta_{1}(t)\right)^{2} + 0.25\left(\frac{d}{dt}\theta_{2}(t)\right)^{2} + 1.5\left(\frac{d}{dt}\,\mathrm{x}_{1}\left(t\right)\right)^{2} + 0.25\left(\frac{d}{dt}\,\mathrm{x}_{2}\left(t\right)\right)^{2} + 1.5\left(\frac{d}{dt}\,\mathrm{y}_{1}\left(t\right)\right)^{2} + 0.25\left(\frac{d}{dt}\,\mathrm{y}_{2}\left(t\right)\right)^{2}$$

```
Lagrangian is:
```

$$0.0221354166666667\left(\frac{d}{dt}\theta_{1}(t)\right)^{2} + 0.25\left(\frac{d}{dt}\theta_{2}(t)\right)^{2} + 1.5\left(\frac{d}{dt}\,\mathrm{x}_{1}\left(t\right)\right)^{2} + 0.25\left(\frac{d}{dt}\,\mathrm{x}_{2}\left(t\right)\right)^{2} + 1.5\left(\frac{d}{dt}\,\mathrm{y}_{1}\left(t\right)\right)^{2} + 0.25\left(\frac{d}{dt}\,\mathrm{y}_{2}\left(t\right)\right)^{2}$$

```
J = Lag

# compute derivative wrt a vector, method 1
# wrap the expression into a SymPy Matrix
J_mat = Matrix([J])
# SymPy Matrix has built-in method for Jacobian
term_one = J_mat.jacobian(q)
term_two = J_mat.jacobian(qdot)
#print('\n\033[1mfirst term:')
#display(term_one) # note that dJdq is 1-by-2 vector, not 2-by-1 as q
#print('\n\033[1msecond term:')
#display(term_two) # note that dJdq is 1-by-2 vector, not 2-by-1 as q


term_one = Lag.diff(q)
term_two = Lag.diff(qdot)
term_two_real = term_two.diff(t)
EL = term_one - term_two_real


dLdq = J_mat.jacobian(q).T
dLdqdot = J_mat.jacobian(qdot).T
d_dLdqdot_dt = dLdqdot.diff(t)
EL = sym.simplify(d_dLdqdot_dt - dLdq)


EL = sym.simplify(EL)
display(EL)
```

$$\begin{bmatrix} 3.0\frac{d^{2}}{dt^{2}}\,\mathrm{x}_{1}\left(t\right) \\ 3.0\frac{d^{2}}{dt^{2}}\,\mathrm{y}_{1}\left(t\right) \\ 0.0442708333333333\frac{d^{2}}{dt^{2}}\theta_{1}(t) \\ 0.5\frac{d^{2}}{dt^{2}}\,\mathrm{x}_{2}\left(t\right) \\ 0.5\frac{d^{2}}{dt^{2}}\,\mathrm{y}_{2}\left(t\right) \\ 0.5\frac{d^{2}}{dt^{2}}\theta_{2}(t) \end{bmatrix}$$

```
#Simulate

lhs = Matrix([EL[0],EL[1],EL[2],EL[3],EL[4],EL[5]])
Force = 0
# define right hand side as a Matrix
rhs = Matrix([0,0,Force,0,0,0])#

# define the equations
eqn = Eq(lhs, rhs)

theta1ddot = theta1dot.diff(t)
theta2ddot = theta2dot.diff(t)
x1ddot = x1dot.diff(t)
x2ddot = x2dot.diff(t)
y1ddot = y1dot.diff(t)
y2ddot = y2dot.diff(t)


q = Matrix([x1,y1,theta1,x2,y2,theta2])
qdot = Matrix([x1dot,y1dot,theta1dot,x2dot,y2dot,theta2dot])
qddot = Matrix([x1ddot,y1ddot,theta1ddot,x2ddot,y2ddot,theta2ddot])

#solve for theta1ddot and theta2ddot
soln = solve(eqn, [x1ddot,y1ddot,theta1ddot,x2ddot,y2ddot,theta2ddot])

display(soln)

x1sol = soln[x1ddot]
x2sol = soln[x2ddot]
y1sol = soln[y1ddot]
y2sol = soln[y2ddot]
theta1sol = soln[theta1ddot]
theta2sol = soln[theta2ddot]


x1dub = sym.lambdify([x1,y1,theta1,x2,y2,theta2,x1dot,y1dot,x2dot,y2dot,theta1dot,theta2dot],x1sol,modules=sym)
x2dub = sym.lambdify([x1,y1,theta1,x2,y2,theta2,x1dot,y1dot,x2dot,y2dot,theta1dot,theta2dot],x2sol,modules=sym)
y1dub = sym.lambdify([x1,y1,theta1,x2,y2,theta2,x1dot,y1dot,x2dot,y2dot,theta1dot,theta2dot],y1sol,modules=sym)
y2dub = sym.lambdify([x1,y1,theta1,x2,y2,theta2,x1dot,y1dot,x2dot,y2dot,theta1dot,theta2dot],y2sol,modules=sym)
theta1dub = sym.lambdify([x1,y1,theta1,x2,y2,theta2,x1dot,y1dot,x2dot,y2dot,theta1dot,theta2dot],theta1sol,modules=sym)
theta2dub = sym.lambdify([x1,y1,theta1,x2,y2,theta2,x1dot,y1dot,x2dot,y2dot,theta1dot,theta2dot],theta2sol,modules=sym)

```

```
{Derivative(theta1(t), (t, 2)): 0.0,
 Derivative(theta2(t), (t, 2)): 0.0,
 Derivative(x_1(t), (t, 2)): 0.0,
 Derivative(x_2(t), (t, 2)): 0.0,
 Derivative(y_1(t), (t, 2)): 0.0,
 Derivative(y_2(t), (t, 2)): 0.0}
```

```
def line_point_dist(point1,point2,point3):
  xz = symbols(r"x_z")
  yz = symbols(r"y_z")

  if point2[0] == point1[0]:
    d = ((point3[0]-point2[0])**2)**1/2
```

```
  elif point2[1] == point1[1]:
    d = ((point3[1]-point2[1])**2)**1/2
  else:
    m = ((point2[1]-point1[1])/(point2[0]-point1[0]))
    #display(m)
    line =  m * (xz - point1[0]) + point1[1] - yz
    C = line.subs({xz:0,yz:0})
    A = m
    B = -1
    d = (((A*point3[0] + B * point3[1] + C)**2)**1/2)/((A**2 + B**2)**1/2)
  return d
display(line_point_dist([-0.50, 2.00000000000000],
       [0.500000000000000, 2.00000000000000],
       [-0.0758567746186340, 0.509775195301280]))
#display(line_point_dist([x1,y1],[3,4],[x2,y2]))
```

```
1.1103849842696691
```

Impact Stuff

Min Distance

```
import math
def minDistance(A, B, E) :

    # vector AB
    AB = [None, None];
    AB[0] = B[0] - A[0];
    AB[1] = B[1] - A[1];

    # vector BP
    BE = [None, None];
    BE[0] = E[0] - B[0];
    BE[1] = E[1] - B[1];

    # vector AP
    AE = [None, None];
    AE[0] = E[0] - A[0];
    AE[1] = E[1] - A[1];

    # Variables to store dot product

    # Calculating the dot product
    AB_BE = AB[0] * BE[0] + AB[1] * BE[1];
    AB_AE = AB[0] * AE[0] + AB[1] * AE[1];

    # Minimum distance from
    # point E to the line segment
    reqAns = 0;
    #print(AB_BE)
    # Case 1
```

```
    if (AB_BE > 0) :

        # Finding the magnitude
        y = E[1] - B[1];
        x = E[0] - B[0];
        reqAns = math.sqrt(x * x + y * y);

    # Case 2
    elif (AB_AE < 0) :
        y = E[1] - A[1];
        x = E[0] - A[0];
        reqAns = math.sqrt(x * x + y * y);

    # Case 3
    else:

        # Finding the perpendicular distance
        x1 = AB[0];
        y1 = AB[1];
        x2 = AE[0];
        y2 = AE[1];
        mod = math.sqrt(x1 * x1 + y1 * y1);
        reqAns = abs(x1 * y2 - y1 * x2) / mod;

    return reqAns;

A = Matrix([0,0])
B = Matrix([2,0])
C = Matrix([1,1])
print(minDistance(A,B,C))
```

```
    1.00000000000000
```

Find Impact Eqs

```
x1_pre = symbols('x_1pr')
y1_pre = symbols('y_1pr')
theta1_pre = symbols('theta_1pr')
x2_pre = symbols('x_2pr')
y2_pre = symbols('y_2pr')
theta2_pre = symbols('theta_2pr')
x1_predot = symbols('x_1prd')
y1_predot = symbols('y_1prd')
theta1_predot = symbols('theta_1prd')
x2_predot = symbols('x_2prd')
y2_predot = symbols('y_2prd')
theta2_predot = symbols('theta_2prd')

x1_post = symbols('x_1po')
y1_post = symbols('y_1po')
theta1_post = symbols('theta_1po')
x2_post = symbols('x_2po')
y2_post = symbols('y_2po')
theta2_post = symbols('theta_2po')
x1_postdot = symbols('x_1pod')
```

```
y1_postdot = symbols('y_1pod')
theta1_postdot = symbols('theta_1pod')
x2_postdot = symbols('x_2pod')
y2_postdot = symbols('y_2pod')
theta2_postdot = symbols('theta_2pod')
lam = symbols(r'lam')


p = Lag.diff(qdot)
Hterm_one = p.transpose() * qdot
Hterm_two = Lag
#print("hterm one")
#display(Hterm_one)
#display(Hterm_one.subs(qsub_pre))
Hamil = Hterm_one[0] - Hterm_two
Hamil = sym.simplify(Hamil)


def find_impact_eqs(s):

  z1 = symbols(r"z_1")
  z2 = symbols(r"z_2")
  z3 = symbols(r"z_3")
  z4 = symbols(r"z_4")
  z5 = symbols(r"z_5")
  z6 = symbols(r"z_6")
  subs_dict_q = {x1:s[0],y1:s[1],theta1:s[2],x2:s[3],y2:s[4],theta2:s[5]}
  fake_dict = {x1dot:z1,y1dot:z2,theta1dot:z3,x2dot:z4,y2dot:z5,theta2dot:z6}
  #subs_dict_qdot = {theta1dot:s[3],theta2dot:s[4],theta3dot:s[5]}
  real_dict = {z1:s[6],z2:s[7],z3:s[8],z4:s[9],z5:s[10],z6:s[11]}

  t_s_d_mod = t_s_d.subs(subs_dict_q).subs(fake_dict).subs(real_dict)
  t_s_b_mod = t_s_b.subs(subs_dict_q).subs(fake_dict).subs(real_dict)




  b1 = (t_s_d * Matrix([w2/2,-l2/2,1]))[0:2]
  b2 = (t_s_d * Matrix([w2/2,-l2/2,1]))[0:2]
  b3 = (t_s_d * Matrix([-w2/2,l2/2,1]))[0:2]
  b4 = (t_s_d * Matrix([w2/2,l2/2,1]))[0:2]
  wp1 = (t_s_b * Matrix([w1/2,l1/2,1]))[0:2]
  wp2 = (t_s_b * Matrix([-w1/2,-l1/2,1]))[0:2]
  wp3 = (t_s_b * Matrix([-w1/2,l1/2,1]))[0:2]
  wp4 = (t_s_b * Matrix([w1/2,-l1/2,1]))[0:2]

  block_p1 = (t_s_d_mod * Matrix([-w2/2,-l2/2,1]))[0:2]
  block_p2 = (t_s_d_mod * Matrix([w2/2,-l2/2,1]))[0:2]
  block_p3 = (t_s_d_mod * Matrix([-w2/2,l2/2,1]))[0:2]
  block_p4 = (t_s_d_mod * Matrix([w2/2,l2/2,1]))[0:2]
  weapon_p1 = (t_s_b_mod * Matrix([w1/2,l1/2,1]))[0:2]
  weapon_p2 = (t_s_b_mod * Matrix([-w1/2,-l1/2,1]))[0:2]
  weapon_p3 = (t_s_b_mod * Matrix([-w1/2,l1/2,1]))[0:2]
  weapon_p4 = (t_s_b_mod * Matrix([w1/2,-l1/2,1]))[0:2]

  full_dist = np.array([minDistance(block_p1,block_p2,weapon_p1),minDistance(block_p1,block_p2,weapon_p2),minDistance(block_p1,block_p2,weapon_p3),minDistance(block_p1,block_p2,weapon_p4),
                        minDistance(block_p2,block_p3,weapon_p1),minDistance(block_p2,block_p3,weapon_p2),minDistance(block_p2,block_p3,weapon_p3),minDistance(block_p2,block_p3,weapon_p4),
                        minDistance(block_p3,block_p4,weapon_p1),minDistance(block_p3,block_p4,weapon_p2),minDistance(block_p3,block_p4,weapon_p3),minDistance(block_p3,block_p4,weapon_p4),
```

```
                    minDistance(block_p4,block_p1,weapon_p1),minDistance(block_p4,block_p1,weapon_p2),minDistance(block_p4,block_p1,weapon_p3),minDistance(block_p4,block_p1,weapon_p4)

points_lookup = np.array([ [block_p1,block_p2,weapon_p1],[block_p1,block_p2,weapon_p2],[block_p1,block_p2,weapon_p3],[block_p1,block_p2,weapon_p4],
                           [block_p2,block_p3,weapon_p1],[block_p2,block_p3,weapon_p2],[block_p2,block_p3,weapon_p3],[block_p2,block_p3,weapon_p4],
                           [block_p3,block_p4,weapon_p1],[block_p3,block_p4,weapon_p2],[block_p3,block_p4,weapon_p3],[block_p3,block_p4,weapon_p4],
                           [block_p4,block_p1,weapon_p1],[block_p4,block_p1,weapon_p2],[block_p4,block_p1,weapon_p3],[block_p4,block_p1,weapon_p4]])
var_lookup = np.array([ [b1,b2,wp1],[b1,b2,wp2],[b1,b2,wp3],[b1,b2,wp4],
                           [b2,b3,wp1],[b2,b3,wp2],[b2,b3,wp3],[b2,b3,wp4],
                           [b3,b4,wp1],[b3,b4,wp2],[b3,b4,wp3],[b3,b4,wp4],
                           [b4,b1,wp1],[b4,b1,wp2],[b4,b1,wp3],[b4,b1,wp4]])

fullest_dist = np.argmin(full_dist)
print("fullest dist position")
display(fullest_dist)
fullest_points = points_lookup[fullest_dist]
print("closest point")
display(fullest_points)

phimat = full_dist[fullest_dist]
#print("PHIMAT")
#display(phimat)
fullest_vars = var_lookup[fullest_dist]
#print("fullest vars")
#display(fullest_vars[0][1])

phivars = line_point_dist(fullest_vars[0],fullest_vars[1],fullest_vars[2])
#display(phivars)
phidot = phivars.diff(t)
#display(phidot)
phidiff = phidot.subs(subs_dict_q).subs(fake_dict).subs(real_dict)
#print("phidot")
#display(phidot)

qsub_pre = {x1:x1_pre,y1:y1_pre,theta1:theta1_pre,x2:x2_pre,y2:y2_pre,theta2:theta2_pre, x1dot:x1_predot,y1dot:y1_predot,theta1dot:theta1_predot,x2dot:x2_predot,y2dot:y2_predot,theta2dot:theta2_predot}
qsub_post = {x1:x1_post,y1:y1_post,theta1:theta1_post,x2:x2_post,y2:y2_post,theta2:theta2_post, x1dot:x1_postdot,y1dot:y1_postdot,theta1dot:theta1_postdot,x2dot:x2_postdot,y2dot:y2_postdot,theta2dot:thet
dLdqdot = Lag.diff(qdot)
dLdqdot_pre = dLdqdot.subs(qsub_pre)
dLdqdot_post = dLdqdot.subs(qsub_post)


#dphidq = phimat.jacobian(q)
#print("dphidq")
#display(dphidq)
dphidq = phivars.diff(q)
#print("dphidq")
#display(dphidq)
dphidq_pre = dphidq.subs(qsub_pre)
dphidq_post = dphidq.subs(qsub_post)

Hamil_pre = Hamil.subs(qsub_pre)
Hamil_post = Hamil.subs(qsub_post)
#print("\n Hamiltonian: ")
#display(Hamil_pre)
#display(Hamil_post)
#print(Hamil_pre)

from sympy import factor
#display(dLdqdot_post[0])
```

```
  impact_rhs = Matrix([dLdqdot_post[0]-dLdqdot_pre[0],dLdqdot_post[1]-dLdqdot_pre[1],dLdqdot_post[2]-dLdqdot_pre[2],dLdqdot_post[3]-dLdqdot_pre[3],dLdqdot_post[4]-dLdqdot_pre[4],dLdqdot_post[5]-dLdqdot_pre

  impact_lhs = Matrix([(lam * dphidq_pre[0]),(lam * dphidq_pre[1]),(lam * dphidq_pre[2]),(lam * dphidq_pre[3]),(lam * dphidq_pre[4]),(lam * dphidq_pre[5]),0])
  #display(impact_rhs)
  #display(impact_lhs)
  #print("THER YA GO")
  impact_eqs = sym.Eq(impact_lhs,impact_rhs)
  #display(impact_eqs)
  return impact_eqs,phivars,phimat,phidiff

s0 = np.array([0,0,np.pi/8,0,2.5,0,0,4,10,0,0,0])
impact_eqs,phivars,phimat,phidiff = find_impact_eqs(s0)
print(phidiff)

```

```
fullest dist position
0
closest point
array([[-0.500000000000000, 2.00000000000000],
       [0.500000000000000, 2.00000000000000],
       [-0.0758567746186340, 0.509775195301280]], dtype=object)
0.287928387309317*Derivative(0.0, t) + 0.293557499746779*Derivative(0.392699081698724, t)
```

Impact Update

```
def impact_update_battlebot(x1pre,y1pre,theta1pre,x1dotpre,y1dotpre,theta1dotpre,x2pre,y2pre,theta2pre,x2dotpre,y2dotpre,theta2dotpre):
  #calls some thing to deterine which impact eqs based on s
  s = [x1pre,y1pre,theta1pre,x1dotpre,y1dotpre,theta1dotpre,x2pre,y2pre,theta2pre,x2dotpre,y2dotpre,theta2dotpre]

  impact_eqs,vareq,dumm1,dummy2 = find_impact_eqs(s)
  print("Impact Equations")
  display(impact_eqs)

  first_dict = {x1_predot:x1dotpre,y1_predot:y1dotpre,theta1_predot:theta1dotpre,x2_predot:x2dotpre,"y_2prd":y2dotpre, "theta_2prd":theta2dotpre }
  subs_dict = {x1_pre:x1pre,y1_pre:y1pre,theta1_pre:theta1pre,  x2_pre:x2pre,"y_2pr":y2pre,theta2_pre:theta2pre ,x1_post:x1pre,y1_post:y1pre, theta1_post:theta1pre ,x2_post:x2pre,y2_post : y2pre, theta2_
  cleaned = impact_eqs.subs(first_dict).subs(subs_dict)

  #display(cleaned.rhs)
  #display(cleaned.lhs)
  #impact_test = np.array([s_test[0],test])
  #qpost1,qpost2,qpost3,
  #display(cleaned)
  impact_sol = sym.solve(cleaned,[x1_postdot,y1_postdot,theta1_postdot,x2_postdot,y2_postdot,theta2_postdot,lam],dict = True)

  #sol = [impact_sol[x1_postdot],impact_sol[y1_postdot],impact_sol[theta1_postdot],impact_sol[x2_postdot],impact_sol[y2_postdot],impact_sol[theta2_postdot]]

  if len(impact_sol) == 1:
    print("DISAPPOINTMENT")
    return False
  else:
    for solu in impact_sol:
      lamsol = solu[lam]
      if abs(lamsol) < 0.001:
        print("OH NO")
```

```
        else:
          print("yay")
          impact_sol = solu
          sol = [-impact_sol[x1_postdot],-impact_sol[y1_postdot],-impact_sol[theta1_postdot],impact_sol[x2_postdot],impact_sol[y2_postdot],impact_sol[theta2_postdot]]
          return sol

s = [0,0,0,0,0,0,0,0.5,0,0,0,0]
print(impact_update_battlebot(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10],s[11]))
#print(impact_update_triple_pend(0.00842, 1.52631626131906*10-1000 ,-0.04365, -1.162,-0.440 ,4.973))
```

```
fullest dist position
1
closest point
array([[-0.500000000000000, -0.500000000000000],
       [0.500000000000000, -0.500000000000000],
       [-0.125000000000000, -0.500000000000000]], dtype=object)
Impact Equations
```

$$\begin{bmatrix} lam\left(x_{1pr} - x_{2pr} + 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right)\right) \\ 0 \\ \frac{lam(0.25\sin\left(\theta_{1pr}\right)+\cos\left(\theta_{1pr}\right))(x_{1pr}-x_{2pr}+0.5\sin\left(\theta_{1pr}\right)-0.5\sin\left(\theta_{2pr}\right)-0.125\cos\left(\theta_{1pr}\right)-0.5\cos\left(\theta_{2pr}\right))}{2} \\ lam\left(-x_{1pr} + x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) + 0.5\sin\left(\theta_{2pr}\right) + 0.125\cos\left(\theta_{1pr}\right) + 0.5\cos\left(\theta_{2pr}\right)\right) \\ 0 \\ \frac{lam(\sin\left(\theta_{2pr}\right)-1.0\cos\left(\theta_{2pr}\right))(x_{1pr}-x_{2pr}+0.5\sin\left(\theta_{1pr}\right)-0.5\sin\left(\theta_{2pr}\right)-0.125\cos\left(\theta_{1pr}\right)-0.5\cos\left(\theta_{2pr}\right))}{2} \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 3.0x_{1pod} - 3.0x_{1prd} \\ 3.0y_{1pod} - 3.0y_{1prd} \\ 0.04427083333333330\theta_{1pod} - 0.04427083333333330\theta_{1prd} \\ 0.5x_{2pod} - 0.5x_{2prd} \\ 0.5y_{2pod} - 0.5y_{2prd} \\ 0.5\theta_{2pod} - 0.5\theta_{2prd} \\ 0.02213541666666670\theta_{1pod}^2 - 0.02213541666666670\theta_{1prd}^2 + 0.25\theta_{2pod}^2 - 0.25\theta_{2prd}^2 + 1.5x_{1pod}^2 - 1.5x_{1prd}^2 + 0.25x_{2pod}^2 - 0.25x_{2prd}^2 + 1.5y_{1pod}^2 - 1.5y_{1prd}^2 + 0.25y_{2pod}^2 - 0.25y_{2prd}^2 \end{bmatrix}$$

```
DISAPPOINTMENT
False
```

Impact Condition

```
import numpy as np

def impact_condition_battlebot(s):
  '''
  impact_eqs,vareq,que,qdawt = find_impact_eqs(s)
  threshold = 0.04
  # if que < threshold and qdawt > 0:
  #print(qdawt)
  if que < threshold:
    print("UHOH")
    return True

  else:
      return False

  '''
```

```
z1 = symbols(r"z_1")
z2 = symbols(r"z_2")
z3 = symbols(r"z_3")
z4 = symbols(r"z_4")
z5 = symbols(r"z_5")
z6 = symbols(r"z_6")

subs_dict_q = {x1:s[0],y1:s[1],theta1:s[2],x2:s[3],y2:s[4],theta2:s[5]}
fake_dict = {x1dot:z1,y1dot:z2,theta1dot:z3,x2dot:z4,y2dot:z5,theta2dot:z6}
#subs_dict_qdot = {theta1dot:s[3],theta2dot:s[4],theta3dot:s[5]}
real_dict = {z1:s[6],z2:s[7],z3:s[8],z4:s[9],z5:s[10],z6:s[11]}



t_s_d_mod = t_s_d.subs(subs_dict_q).subs(fake_dict).subs(real_dict)
t_s_b_mod = t_s_b.subs(subs_dict_q).subs(fake_dict).subs(real_dict)

block_p1 = (t_s_d_mod * Matrix([-w2/2,-l2/2,1]))[0:2]

block_p2 = (t_s_d_mod * Matrix([w2/2,-l2/2,1]))[0:2]
block_p3 = (t_s_d_mod * Matrix([-w2/2,l2/2,1]))[0:2]
block_p4 = (t_s_d_mod * Matrix([w2/2,l2/2,1]))[0:2]
weapon_p1 = (t_s_b_mod * Matrix([w1/2,l1/2,1]))[0:2]
weapon_p2 = (t_s_b_mod * Matrix([-w1/2,-l1/2,1]))[0:2]
weapon_p3 = (t_s_b_mod * Matrix([-w1/2,l1/2,1]))[0:2]
weapon_p4 = (t_s_b_mod * Matrix([w1/2,-l1/2,1]))[0:2]
threshold = 0.05

points_lookup = np.array([ [block_p1,block_p2,weapon_p1],[block_p1,block_p2,weapon_p2],[block_p1,block_p2,weapon_p3],[block_p1,block_p2,weapon_p4],
                           [block_p2,block_p3,weapon_p1],[block_p2,block_p3,weapon_p2],[block_p2,block_p3,weapon_p3],[block_p2,block_p3,weapon_p4],
                           [block_p3,block_p4,weapon_p1],[block_p3,block_p4,weapon_p2],[block_p3,block_p4,weapon_p3],[block_p3,block_p4,weapon_p4],
                           [block_p4,block_p1,weapon_p1],[block_p4,block_p1,weapon_p2],[block_p4,block_p1,weapon_p3],[block_p4,block_p1,weapon_p4]])
full_dist = np.array([minDistance(block_p1,block_p2,weapon_p1),minDistance(block_p1,block_p2,weapon_p2),minDistance(block_p1,block_p2,weapon_p3),minDistance(block_p1,block_p2,weapon_p4),
                      minDistance(block_p2,block_p3,weapon_p1),minDistance(block_p2,block_p3,weapon_p2),minDistance(block_p2,block_p3,weapon_p3),minDistance(block_p2,block_p3,weapon_p4),
                      minDistance(block_p3,block_p4,weapon_p1),minDistance(block_p3,block_p4,weapon_p2),minDistance(block_p3,block_p4,weapon_p3),minDistance(block_p3,block_p4,weapon_p4),
                      minDistance(block_p4,block_p1,weapon_p1),minDistance(block_p4,block_p1,weapon_p2),minDistance(block_p4,block_p1,weapon_p3),minDistance(block_p4,block_p1,weapon_p4)])
numtrues = 0

looking = np.zeros((16,1))
#print(looking[4])
for i in range(15):
  looking[i] = full_dist[i]
  if full_dist[i] < threshold:
    numtrues = numtrues + 1


smol = np.argmin(looking)
look = points_lookup[smol]
#print(np.shape(look))
#print(look)
if numtrues == 0:
  return False,look
else:
  return True,look

'''
if minDistance(block_p1,block_p2,weapon_p1) < threshold:
  return True
```

```
  elif minDistance(block_p1,block_p2,weapon_p2) < threshold:
    return True
  elif minDistance(block_p1,block_p2,weapon_p3) < threshold:
    return True
  elif minDistance(block_p1,block_p2,weapon_p4) < threshold:
    return True
  elif minDistance(block_p2,block_p3,weapon_p1) < threshold:
    return True
  elif minDistance(block_p2,block_p3,weapon_p2) < threshold:
    return True
  elif minDistance(block_p2,block_p3,weapon_p3) < threshold:
    return True
  elif minDistance(block_p2,block_p3,weapon_p4) < threshold:
    return True
  elif minDistance(block_p3,block_p4,weapon_p1) < threshold:
    return True
  elif minDistance(block_p3,block_p4,weapon_p2) < threshold:
    return True
  elif minDistance(block_p3,block_p4,weapon_p3) < threshold:
    return True
  elif minDistance(block_p3,block_p4,weapon_p4) < threshold:
    return True
  elif minDistance(block_p4,block_p1,weapon_p1) < threshold:
    return True
  elif minDistance(block_p4,block_p1,weapon_p2) < threshold:
    return True
  elif minDistance(block_p4,block_p1,weapon_p3) < threshold:
    return True
  elif minDistance(block_p4,block_p1,weapon_p4) < threshold:
    return True
  else:
    return False

  '''
  '''
  phi_clean = phimat.subs(subs_dict_q)

  phi_clean = phi_clean[0]
  phidot_clean = phi_clean.subs(fake_dict)
  phidot_clean = phidot_clean.subs(subs_dict_q)
  phidot_clean = phidot_clean.subs(real_dict)
  qdawt = phidot_clean
  phidot = phimat.diff(t)
  using = phidot
  #using = x3dot
  #use dphidq here?
  phidot_clean = using.subs(fake_dict)

  phidot_clean = phidot_clean.subs(subs_dict_q)
  phidot_clean = phidot_clean.subs(real_dict)
  '''
  #que = phi_clean


  #print(que)
  #print(qdawt)

  #print(qdawt<0)
```

```
  #impact = phi.subs(theta,que)
```

```
```

```
1
```

```
1
```

```
def integrate(f, xt, dt):
    k1 = dt * f(xt)
    k2 = dt * f(xt+k1/2.)
    k3 = dt * f(xt+k2/2.)
    k4 = dt * f(xt+k3)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt
```

Impact Simulation Func

```
from sympy import evalf
def simulate_impact(f, x0, tspan, dt, integrate):
    """
    This function takes in an initial condition x0, a timestep dt,
    a time span tspan consisting of a list [min_time, max_time],
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x0. It outputs a full trajectory simulated
    over the time span of dimensions (xvec_size, time_vec_size).

    Parameters
    ============
    f: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    x0: NumPy array
        initial conditions
    tspan: Python list
        tspan = [min_time, max_time], it defines the start and end
        time of simulation
    dt:
        time step for numerical integration
    integrate: Python function
        numerical integration method used in this simulation

    Return
    ============
    x_traj:
        simulated trajectory of x(t) from t=0 to tf
    """
    N = int((max(tspan)-min(tspan))/dt)
    x = np.copy(x0)
    tvec = np.linspace(min(tspan),max(tspan),N)
    xtraj = np.zeros((len(x0),N))

```

```
  for i in range(N-1):
    if i % 10 == 0:
      print(i)

    '''
    xtraj[:,i]=integrate(f,x,dt)
    x = np.copy(xtraj[:,i])
    '''

    check = integrate(f,x,dt)
    check_imp, dummy = impact_condition_battlebot(check)
    Looking_Points[i] = dummy

    Hamilclean = Hamil.subs({x1dot:check[6],y1dot:check[7],theta1dot:check[8],x2dot:check[9],y2dot:check[10],theta2dot:check[11]})
    Hamilray[:,i] = [i,Hamilclean.evalf()]

    if check_imp == True:

      s_test = check
      print("IMPACT")
      #print(check)
      test = impact_update_battlebot(check[0],check[1],check[2],check[3],check[4],check[5],check[6],check[7],check[8],check[9],check[10],check[11])
      print(test)
      if test == None:
        print("Rats")
      elif test != False:

        #print("TEST")
        #rint(test)
        impact_test = np.array([s_test[0],s_test[1],s_test[2],s_test[3],s_test[4],s_test[5],test[0],test[1],test[2],test[3],test[4],test[5]])
        check = impact_test

    #print(check)
    xtraj[:,i] = check
    x = np.copy(xtraj[:,i])

  return xtraj
```

Simulate it

```


#test ddots at initial conditions
#print('theta1ddot = ', theta1dub(-1.57,-1.57,0,0))
#print('theta2ddot = ', theta2dub(-1.57,-1.57,0,0))

def dyn(s):

    thing1 = x1dub(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10] , s[11])
    thing2 = y1dub(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10] , s[11])
    thing3 = theta1dub(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10] , s[11])
    thing4 = x2dub(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10] , s[11])
    thing5 = y2dub(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10] , s[11])
    thing6 = theta2dub(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8],s[9],s[10] , s[11])

```

```
    thing = np.array([ s[6],s[7],s[8],s[9],s[10] , s[11] ,thing1,thing2,thing3,thing4,thing5,thing6])
    return thing

#s0 = np.array([0,0,np.pi/3,4,4,0,2,2,1,0,0,0]) # spinny case,angle strange
#s0 = np.array([1,0,np.pi/8,0,2.5,np.pi/8,-0.33,2,15,0,0,0]) # head on certical
#s0 = np.array([0,0,np.pi/8,0.5,2.5,0,.5,2.5,10,0,0,0])###FORMAT: XYTH1 XYTH2 XYTH_DOT_1 XYTH_DOT_2
s0 = np.array([0,0,0,1.8,0,np.pi/6,0.8,0,12,0,0,12])# sideways spinner



timestep = 0.01
steps_sec = 1/timestep
end_time = 3
steps = end_time/timestep

Hamilray = np.zeros((2,int(steps)))

Looking_Points = np.zeros((int(steps),3,2))

traj = simulate_impact(dyn, s0, [0, end_time], timestep, integrate)
#plt.plot(np.arange(steps)*timestep,traj[2].T)
#plt.plot(np.arange(steps)*timestep,traj[5].T)
plt.plot(Hamilray.T)
#print(Hamilray[0])
#print(Hamilray[1][0:int(steps)])
#plt.plot((Hamil[0][0:int(steps)]),(Hamil[1][0:int(steps)]))
plt.title("Simulated Trajectories")
plt.xlabel("time")
plt.ylabel("Angle")
plt.legend(('theta1(t)'),loc= 'upper right')
#plt.show()
print("Formatting")
```

```
0
10
20
30
40
50
60
70
80
IMPACT
fullest dist position
2
closest point
array([[1.16557293635718, 0.312253584314967],
       [1.48774641568503, -0.634427063642824],
       [1.23475935334016, 0.0254520755306376]], dtype=object)
Impact Equations
```

$$\begin{bmatrix} lam\left(x_{1pr} - x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right)\right) \\ 0 \\ \frac{lam(0.25\sin\left(\theta_{1pr}\right) - 1.0\cos\left(\theta_{1pr}\right))(x_{1pr} - x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right))}{2} \\ lam\left(-x_{1pr} + x_{2pr} + 0.5\sin\left(\theta_{1pr}\right) + 0.5\sin\left(\theta_{2pr}\right) + 0.125\cos\left(\theta_{1pr}\right) + 0.5\cos\left(\theta_{2pr}\right)\right) \\ 0 \\ \frac{lam(\sin\left(\theta_{2pr}\right) - 1.0\cos\left(\theta_{2pr}\right))(x_{1pr} - x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right))}{2} \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 3.0x_{1pod} - 3.0x_{1prd} \\ 3.0y_{1pod} - 3.0y_{1prd} \\ 0.04427083333333330\theta_{1pod} - 0.04427083333333330\theta_{1prd} \\ 0.5x_{2pod} - 0.5x_{2prd} \\ 0.5y_{2pod} - 0.5y_{2prd} \\ 0.5\theta_{2pod} - 0.5\theta_{2prd} \\ 0.02213541666666670\theta_{1pod}^2 - 0.02213541666666670\theta_{1prd}^2 + 0.25\theta_{2pod}^2 - 0.25\theta_{2prd}^2 + 1.5x_{1pod}^2 - 1.5x_{1prd}^2 + 0.25x_{2pod}^2 - 0.25x_{2prd}^2 + 1.5y_{1pod}^2 - 1.5y_{1prd}^2 + 0.25y_{2pod}^2 - 0.25y_{2prd}^2 \end{bmatrix}$$

```
OH NO
yay
[-3.16751937439349, 0, -8.96496643556960, -8.20511624636092, 0.0, 6.33671850340271]
90
IMPACT
fullest dist position
2
closest point
array([[1.06502169555613, 0.271451924409251],
       [1.44649691312714, -0.652927141980263],
       [1.20329568296575, -0.0207363483010927]], dtype=object)
Impact Equations
```

$$\begin{bmatrix} lam\left(x_{1pr} - x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right)\right) \\ 0 \\ \frac{lam(0.25\sin\left(\theta_{1pr}\right) - 1.0\cos\left(\theta_{1pr}\right))(x_{1pr} - x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right))}{2} \\ lam\left(-x_{1pr} + x_{2pr} + 0.5\sin\left(\theta_{1pr}\right) + 0.5\sin\left(\theta_{2pr}\right) + 0.125\cos\left(\theta_{1pr}\right) + 0.5\cos\left(\theta_{2pr}\right)\right) \\ 0 \\ \frac{lam(\sin\left(\theta_{2pr}\right) - 1.0\cos\left(\theta_{2pr}\right))(x_{1pr} - x_{2pr} - 0.5\sin\left(\theta_{1pr}\right) - 0.5\sin\left(\theta_{2pr}\right) - 0.125\cos\left(\theta_{1pr}\right) - 0.5\cos\left(\theta_{2pr}\right))}{2} \\ 0 \end{bmatrix} =$$

$$\begin{bmatrix} 3.0x_{1pod} - 3.0x_{1prd} \\ 3.0y_{1pod} - 3.0y_{1prd} \\ 0.04427083333333330\theta_{1pod} - 0.04427083333333330\theta_{1prd} \\ 0.5x_{2pod} - 0.5x_{2prd} \\ 0.5y_{2pod} - 0.5y_{2prd} \end{bmatrix}$$

$$\begin{bmatrix} 0.5\theta_{2pod} - 0.5\theta_{2prd} \\ 0.0221354166666667\theta_{1pod}^2 - 0.0221354166666667\theta_{1prd}^2 + 0.25\theta_{2pod}^2 - 0.25\theta_{2prd}^2 + 1.5x_{1pod}^2 - 1.5x_{1prd}^2 + 0.25x_{2pod}^2 - 0.25x_{2prd}^2 + 1.5y_{1pod}^2 - 1.5y_{1prd}^2 + 0.25y_{2pod}^2 - 0.25y_{2prd}^2 \end{bmatrix}$$

```
yay
[1.44243666109684, 0, -6.94601582274295, 10.7571967454385, 0.0, 2.04785157392523]
100
110
120
130
140
150
160
170
180
190
200
210
```

ANIMATION

```
def animate_battlebot(theta_array,T=10,xraymode = False):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    =============================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    L2:
        length of the second pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """

    ################################
    # Imports required for animation.
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML
    import plotly.graph_objects as go

    ######################
    # Browser configuration.
    def configure_plotly_browser_state():
        import IPython
        display(IPython.core.display.HTML('''
            <script src="/static/components/requirejs/require.js"></script>
            <script>
              requirejs.config({
                paths: {
                  base: '/static/base',
                  plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
                },
```

```
            });
          </script>
          '''))
    configure_plotly_browser_state()
    init_notebook_mode(connected=False)

    ################################################
    # Getting data from pendulum angle trajectories.
    xx1= theta_array[0]
    yy1= theta_array[1]
    thetat1 = theta_array[2]
    xx2= theta_array[3]
    yy2= theta_array[4]
    thetat2 = theta_array[5]
    N = len(theta_array[0]) # Need this for specifying length of simulation

    ################################################
    # Define arrays containing data for frame axes
    # In each frame, the x and y axis are always fixed
    x_axis = np.array([0.3, 0.0])
    y_axis = np.array([0.0, 0.3])
    bump = 0.001
    b_points_x = np.array([-w1/2-bump,w1/2+bump,w1/2+bump,-w1/2-bump])
    b_points_y = np.array([-l1/2-bump,-l1/2-bump,l1/2+bump,l1/2+bump])
    d_points_x = np.array([-w2/2-bump,w2/2+bump,w2/2+bump,-w2/2-bump])
    d_points_y = np.array([-l2/2-bump,-l2/2-bump,l2/2+bump,l2/2+bump])
    # Use homogeneous tranformation to transfer these two axes/points
    # back to the fixed frame

    shape_b_x = np.zeros((4,N))
    shape_b_y = np.zeros((4,N))
    shape_d_x = np.zeros((4,N))
    shape_d_y = np.zeros((4,N))


    shape_imp_x = np.zeros((3,N))
    shape_imp_y = np.zeros((3,N))

    #sub_d1 = {theta1dot:theta_array[2][k],theta2dot:theta_array[3][k]}

    for i in range(N): # iteration through each time step




        sub_d2 = {theta1:theta_array[0][i],theta2:theta_array[1][i]}
        t_s_a = np.array([[1, 0, 1* theta_array[0][i]],
                          [0,  1, 1* theta_array[1][i]],
                          [0, 0, 1]])

        t_a_b = np.array([[np.cos(theta_array[2][i]), -np.sin(theta_array[2][i]), 0],
                          [np.sin(theta_array[2][i]),  np.cos(theta_array[2][i]), 0],
                          [                        0,                         0, 1]])
        #t_s_b = t_s_a * t_a_b
        t_s_b = np.dot(t_s_a,t_a_b)
        #BLOCK
        t_s_c = np.array([[1, 0, 1 * theta_array[3][i]],
```

```
                      [0,  1, 1 * theta_array[4][i]],
                      [0, 0, 1]])

      t_c_d = np.array([[np.cos(theta_array[5][i]), -np.sin(theta_array[5][i]), 0],
                      [np.sin(theta_array[5][i]),  np.cos(theta_array[5][i]), 0],
                      [                       0,                          0, 1]])

      t_s_d = np.dot(t_s_c,t_c_d)
      #t_s_d = t_s_c * t_c_d
      block_p1 = [-w2/2-bump,-l2/2-bump,1]
      block_p2 = [w2/2+bump,-l2/2-bump,1]
      Weapon_p1 = [w1/2+bump,l1/2+bump,1]

      if xraymode == True:
        point1 = Looking_Points[i][0]
        point2 = Looking_Points[i][1]
        point3 = Looking_Points[i][2]
      else:

        point1 = t_s_b.dot([Weapon_p1[0],Weapon_p1[1],Weapon_p1[2]])
        point2 = t_s_d.dot([block_p1[0],block_p1[1],block_p1[2]])
        point3 = t_s_d.dot([block_p2[0],block_p2[1],block_p2[2]])

      shape_imp_x[:,i] = [point1[0],point2[0],point3[0]]
      shape_imp_y[:,i] = [point1[1],point2[1],point3[1]]
      #print(shape_imp_x[:,i])
      for j in range(4):
        points = t_s_b.dot([b_points_x[j],b_points_y[j] ,1])[0:2]
        shape_b_x[j,i] = points[0]
        shape_b_y[j,i] = points[1]

      for j in range(4):
        points = t_s_d.dot([d_points_x[j],d_points_y[j], 1])[0:2]
        shape_d_x[j,i] = points[0]
        shape_d_y[j,i] = points[1]

      # transfer the x and y axes in body frame back to fixed frame at
      # the current time step
      '''
      shape_b_x[:,i] = t_wb.dot([x_axis[0], x_axis[1], 1])[0:2]
      frame_b_y_axis[:,i] = t_wb.dot([y_axis[0], y_axis[1], 1])[0:2]
      frame_d_x_axis[:,i] = t_wd.dot([x_axis[0], x_axis[1], 1])[0:2]
      frame_d_y_axis[:,i] = t_wd.dot([y_axis[0], y_axis[1], 1])[0:2]
      '''
      #print(t_wb.dot([y_axis[0], y_axis[1], 1])[0:2])


  ####################################
  # Using these to specify axis limits.
  xm = -5 #np.min(xx1)-0.5
  xM = 5 #np.max(xx1)+0.5
  ym = -3 #np.min(yy1)-2.5
  yM = 5#np.max(yy1)+1.5


  print("FINALLY")
```

```
    ############################
    # Defining data dictionary.
    # Trajectories are here.
    data=[
        # note that except for the trajectory (which you don't need this time),
        # you don't need to define entries other than "name". The items defined
        # in this list will be related to the items defined in the "frames" list
        # later in the same order. Therefore, these entries can be considered as
        # labels for the components in each animation frame
        dict(name='Weapon Bar'),
        dict(name='Block'),
        dict(name='Weapon Center'),
        dict(name = 'Impact Points')


        # You don't need to show trajectory this time,
        # but if you want to show the whole trajectory in the animation (like what
        # you did in previous homeworks), you will need to define entries other than
        # "name", such as "x", "y". and "mode".

        # dict(x=xx1, y=yy1,
        #      mode='markers', name='Pendulum 1 Traj',
        #      marker=dict(color="fuchsia", size=2)
        #     ),
        # dict(x=xx2, y=yy2,
        #      mode='markers', name='Pendulum 2 Traj',
        #      marker=dict(color="purple", size=2)
        #     ),
        ]

    ##############################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(autosize=False, width=1000, height=1000,
                xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor = "x",dtick=1),
                title='Double Pendulum Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame': {'duration': T, 'redraw': False}}]},
                                           {'args': [[None], {'frame': {'duration': T, 'redraw': False}, 'mode': 'immediate',
                                            'transition': {'duration': 0}}],'label': 'Pause','method': 'animate'}
                                          ]
                              }]
               )

    ########################################
    # Defining the frames of the simulation.
    # This is what draws the lines from
    # joint to joint of the pendulum.
    #print(shape_b_y)
    frames=[dict(data=[# first three objects correspond to the arms and two masses,
                       # same order as in the "data" variable defined above (thus
                       # they will be labeled in the same order)
                       go.Scatter(
                            x=[shape_b_x[0,k],shape_b_x[1,k],shape_b_x[2,k],shape_b_x[3,k]],
                            y=[shape_b_y[0,k],shape_b_y[1,k],shape_b_y[2,k],shape_b_y[3,k]],
```

```
                         fill="toself",
                         marker=dict(color="blue", size=12)),
                    go.Scatter(
                         x=[xx2[k]],
                         y=[yy2[k]],
                         mode="markers",
                         marker=dict(color="orange", size=12)),
                    go.Scatter(
                         x=[shape_d_x[0,k],shape_d_x[1,k],shape_d_x[2,k],shape_d_x[3,k]],
                         y=[shape_d_y[0,k],shape_d_y[1,k],shape_d_y[2,k],shape_d_y[3,k]],
                         fill= "toself",
                         marker=dict(color="red", size=12)),
                    go.Scatter(
                         x=[shape_imp_x[0,k],shape_imp_x[1,k],shape_imp_x[2,k]],
                         y=[shape_imp_y[0,k],shape_imp_y[1,k],shape_imp_y[2,k]],
                         mode="markers",
                         marker=dict(color="green", size=12)),

                    ]) for k in range(N)]

  ########################################
  # Putting it all together and plotting.
  figure1=dict(data=data, layout=layout, frames=frames)
  iplot(figure1)



animate_battlebot(traj,3,True)
```

```
FINALLY
```

Executing (17s) (› animat… › … › return_figure_f… › _… › _… › valid… › _… › __s… › _set_co… › valid… › _… › __s… › __s… › _set_co… › valid… › _… › __s… › _set_co… › valid… › _… › __s… › _set_co… › valid… › _… › _… › __i… ••• ✕