Problem 1

$$SO(n) = \{ A \in \mathbb{R}^{n \times n} \mid A^T A = I_{n\times n} \text{ and } \det(A) = 1 \}$$

$$A = \frac{d}{dt}(R)R^{-1} = \dot{R}R^T \qquad A^T = R\dot{R}$$

↳ skew symmetric means that $A^T = -A$ and if $R \in SO(n)$, then $R^T R = I_{n\times n}$ and $\det(R) = 1$

Since we know $R^T R = RR^T = I_{n\times n}$, $\frac{d}{dt}(RR^T) = 0$

$$\frac{d}{dt}(RR^T) = 0$$

$$\dot{R}R^T + R\dot{R}^T = 0$$

$$\dot{R}R^T = -R\dot{R}^T$$

where $A = \dot{R}R^T$ and $A^T = (\dot{R}R^T)^T = R\dot{R}^T$

such that $\Rightarrow \dot{R}R^T = -R\dot{R}^T$

$\Rightarrow A = -A^T$

Problem 2

$$\hat{\omega} r_b = -\hat{r}_b \omega$$

$$\omega \times r = -(r \times \omega)$$

$$\begin{vmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{vmatrix} \times \begin{vmatrix} r_1 \\ r_2 \\ r_3 \end{vmatrix} = - \begin{vmatrix} r_1 \\ r_2 \\ r_3 \end{vmatrix} \times \begin{vmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{vmatrix}$$

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ \omega_1 & \omega_2 & \omega_3 \\ r_1 & r_2 & r_3 \end{vmatrix} = - \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ r_1 & r_2 & r_3 \\ \omega_1 & \omega_2 & \omega_3 \end{vmatrix}$$

$$\Rightarrow (\omega_2 r_3 - \omega_3 r_2)\hat{i} - (\omega_1 r_3 - \omega_3 r_1)\hat{j} + (\omega_1 r_2 - \omega_2 r_1)\hat{k} - \left[(r_2\omega_3 - r_3\omega_2)\hat{i} - (r_1\omega_3 - r_3\omega_1)\hat{j} + (r_1\omega_2 - r_2\omega_1)\hat{k}\right]$$

$$\Rightarrow \begin{vmatrix} \omega_2 r_3 - \omega_3 r_2 \\ \omega_3 r_1 - \omega_1 r_3 \\ \omega_1 r_2 - \omega_2 r_1 \end{vmatrix} = - \begin{vmatrix} r_2 \omega_3 - r_3 \omega_2 \\ r_3 \omega_1 - r_1 \omega_3 \\ r_1 \omega_2 - r_2 \omega_1 \end{vmatrix}$$

$$\Rightarrow \begin{vmatrix} \omega_2 r_3 - \omega_3 r_2 \\ \omega_3 r_1 - \omega_1 r_3 \\ \omega_1 r_2 - \omega_2 r_1 \end{vmatrix} = \begin{vmatrix} r_3 \omega_2 - r_2 \omega_3 \\ r_1 \omega_3 - r_3 \omega_1 \\ r_2 \omega_1 - r_1 \omega_2 \end{vmatrix}$$

$$\Rightarrow \hat{\omega} r_b = - \hat{r}_b \omega$$

ME314 Homework 7

Submission instructions

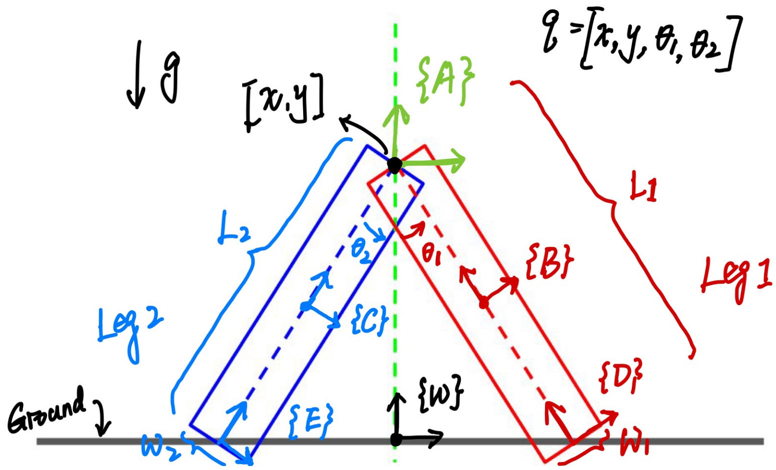
Deliverables that should be included with your submission are shown in **bold** at the end of each problem statement and the corresponding supplemental material. **Your homework will be graded IFF you submit a single PDF, .mp4 videos of animations when requested and a link to a Google colab file that meet all the requirements outlined below.**

- List the names of students you've collaborated with on this homework assignment.
- Include all of your code (and handwritten solutions when applicable) used to complete the problems.
- Highlight your answers (i.e. **bold** and outline the answers) for handwritten or markdown questions and include simplified code outputs (e.g. .simplify()) for python questions.
- Enable Google Colab permission for viewing
    - Click Share in the upper right corner
    - Under "Get Link" click "Share with..." or "Change"
    - Then make sure it says "Anyone with Link" and "Editor" under the dropdown menu
- Make sure all cells are run before submitting (i.e. check the permission by running your code in a private mode)
    - Please don't make changes to your file after submitting, so we can grade it!
- Submit a link to your Google Colab file that has been run (before the submission deadline) and don't edit it afterwards!

NOTE: This Jupyter Notebook file serves as a template for you to start homework. Make sure you first copy this template to your own Google driver (click "File" -> "Save a copy in Drive"), and then start to edit it.

```
##################################################################################################
# If you're using Google Colab, uncomment this section by selecting the whole section and press
# ctrl+'/' on your and keyboard. Run it before you start programming, this will enable the nice
# LaTeX "display()" function for you. If you're using the local Jupyter environment, leave it alone
##################################################################################################
import sympy as sym
def custom_latex_printer(exp,**options):
    from google.colab.output._publish import javascript
    url = "https://cdnjs.cloudflare.com/ajax/libs/mathjax/3.1.1/latest.js?config=TeX-AMS_HTML"
    javascript(url=url)
    return sym.printing.latex(exp,**options)
sym.init_printing(use_latex="mathjax",latex_printer=custom_latex_printer)
```

```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/master/biped_simplified.jpg' width=600' heigh
```



Problem 3 (60pts)

Consider a person doing the splits (shown in the image above). To simplify the model, we ignore the upper body and assume the knees can not bend --- which means we only need four variables $q = [x, y, \theta_1, \theta_2]$ to configure the system. x and y are the position of the intersection point of the two legs, θ_1 and θ_2 are the angles between the legs and the green vertical dash line. The feet are constrained on the ground, and there is no friction between the feet and the ground.

Each leg is a rigid body with length $L = 1$, width $W = 0.2$, mass $m = 1$, and rotational inertia $J = 1$ (assuming the center of mass is at the center of geometry). Moreover, there are two torques applied on θ_1 and θ_2 to control the legs to track a desired trajectory:

$$\theta_1^d(t) = \frac{\pi}{15} + \frac{\pi}{3}\sin^2(\frac{t}{2})$$
$$\theta_2^d(t) = -\frac{\pi}{15} - \frac{\pi}{3}\sin^2(\frac{t}{2})$$

and the torques are:

$$F_{\theta_1} = -k_1(\theta_1 - \theta_1^d)$$
$$F_{\theta_2} = -k_1(\theta_2 - \theta_2^d)$$

In this problem we use $k_1 = 20$.

Given the model description above, define the frames that you need (several example frames are shown in the image as well), simulate the motion of the biped from rest for $t \in [0, 10]$, $dt = 0.01$, with initial condition $q = [0, L_1\cos(\frac{\pi}{15}), \frac{\pi}{15}, -\frac{\pi}{15}]$. You will need to modify the animation function to display the leg as a rectangle, an example of the animation can be found at https://youtu.be/w8XHYrEoWTc.

Hint 1: Even though this is a 2D system, in order to compute kinetic energy from both translation and rotation you will need to model the system in the 3D world --- the z coordinate is always zero and the rotation is around the z axis (based on these facts, what should the SE(3) matrix and rotational inertia tensor look like?). This also means you need to represent transformations in SE(3) and the body velocity $\mathcal{V}_b \in \mathbb{R}^6$.

Hint 2: It could be helpful to define several helper functions for all the matrix operations you will need to use. For example, a function that returns SE(3) matrices given a rotation angle and 2D translation vector, functions for "hat" and "unhat" operations, a function for the matrix inverse of SE(3) (which is definitely not the same as the SymPy matrix inverse function), and a function that returns the time derivative of SO(3) or SE(3).

Hint 3: In this problem the external force depends on time t. Therefore, in order to solve for the symbolic solution you need to substitute your configuration variables, which are defined as symbolic functions of time t (such as $\theta_1(t)$ and $\frac{d}{dt}\theta_1(t)$), with dummy symbolic variables. For the same reason (the dynamics now explicitly depend on time), you will need to do some tiny modifications on the "integrate" and "simulate" functions, a good reference can be found at https://en.wikipedia.org/wiki/Runge-Kutta_methods.

Hint 4: Symbolically solving this system should be fast, but if you encountered some problem when solving the dynamics symbolically, an alternative method is to solve the system numerically --- substitute in the system state at each time step during simulation and solve for the numerical solution --- but based on my experience, this would cost more than one hour for 500 time steps, so it's not recommended.

Hint 5: The animation of this problem is similar to the one in last homework --- the coordinates of the vertices in the body frame are constant, you just need to transfer them back to the world frame using the the transformation matrices you already have in the simulation.

Hint 6: Be careful to consider the relationship between the frames and to not build in any implicit assumptions (such as assuming some variables are fixed).

Hint 7: The rotation, by convention, is assumed to follow the right hand rule, which means the z-axis is out of the screen and the positive rotation orientation is counter-clockwise. Make sure you follow a consistent set of positive directions for all the computation.

Hint 8: This problem is designed as a "mini-project", it could help you estimate the complexity of your final project, and you could adjust your proposal based on your experience with this problem.

Turn in: A copy of the code used to simulate and animate the system. Also, include a plot of the trajectory and upload a video of the animation separately through Canvas. The video should be in ".mp4" format, you can use screen capture or record the screen directly with your phone.

```
from os import get_blocking
from sympy import sin, cos
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Markdown, display

# Defining necessary symbols
t = sym.symbols('t')
l = sym.symbols('l')
m = sym.symbols('m')
g = sym.symbols('g')
J = sym.symbols('J')
k = sym.symbols('k')


# define configuration variables
x = sym.Function('x')(t)
y = sym.Function('y')(t)
```

```python

theta1 = sym.Function('theta_1')(t)
theta2 = sym.Function('theta_2')(t)

# derivatives of configuration variables
x_d = x.diff(t)
y_d = y.diff(t)
theta1_d = theta1.diff(t)
theta2_d = theta2.diff(t)

x_dd = x_d.diff(t)
y_dd = y_d.diff(t)
theta1_dd = theta1_d.diff(t)
theta2_dd = theta2_d.diff(t)

### ----------------- DEFINING TRANSFORMATIONS ----------------- ###


# Defining Function to compute SE(3) given rotation angle and translation.
def SE3(theta, p):
  assert len(p) == 3
  if theta == 0:
    g = sym.BlockMatrix([[sym.eye(3), p], [sym.zeros(1, 3), sym.Matrix([1])]])
    g = g.as_explicit()
  else:
    R = sym.Matrix([ [cos(theta), -sin(theta), 0], [sin(theta), cos(theta), 0], [0, 0, 1] ])
    Rot = sym.BlockMatrix([[R, sym.zeros(3, 1)], [sym.zeros(1, 3), sym.Matrix([1])]])
    p = sym.BlockMatrix([[sym.eye(3), p], [sym.zeros(1, 3), sym.Matrix([1])]])
    g = Rot*p
    g = g.as_explicit()
  return g

### DEFINING RIGID BODY TRANSFORMATIONS
g_wa = SE3(0, sym.Matrix([x, y, 0]))
g_ab = SE3(theta1, sym.Matrix([0, -l/2, 0]))
g_ac = SE3(theta2, sym.Matrix([0, -l/2, 0]))
g_bd = SE3(0, sym.Matrix([0, -l/2, 0]))
g_ce = SE3(0, sym.Matrix([0, -l/2, 0]))

# Defining function to compute the derivative of an SE3
def derivate(m):
  return m.diff(t)

# Defining a function to compute the inverse of an SE3
def inverse(m):
  temp = sym.zeros(4, 4)
  temp[0:3, 0:3] = m[0:3, 0:3].T
  temp[0:3, 3] = -temp[0:3, 0:3] * m[0:3, 3]
  temp[3, 0:3] = sym.zeros(1, 3)
  temp[3, 3] = 1
  return temp

# Defining a function that converst a hatted 4x4 matrix into an unhatted 6x1 vector
def unhat(m):
    assert m.shape == (4, 4)
    w_hat = m[0:3, 0:3]
    w = sym.Matrix([w_hat[2, 1], -w_hat[2, 0], w_hat[1, 0]])
    v = m[0:3, 3]  # Select the first three elements of the fourth column
    unhat = sym.Matrix.vstack(v, w)
    return unhat

# Defining a function that converst an unhatted 6x1 vector into a hatted 4x4 matrix
def hat(m):
    assert m.shape == (6, 1)
    v = m[0:3, 0]
    w = m[3:6, 0]
    w_hat = sym.Matrix([[0, -w[2], w[1]], [w[2], 0, -w[0]], [-w[1], w[0], 0]])
    hat = sym.BlockMatrix([[w_hat, v], [sym.zeros(1, 3), sym.zeros(1, 1)]])
    hat = hat.as_explicit()
    return hat

# DEFINING BODY VELOCITIES
# define the transformation from W --> C
g_wc = g_wa*g_ac
# define body velocity at c
Vc_hat = inverse(g_wc)*derivate(g_wc)
# unhat body velocity
```

```
Vc = unhat(Vc_hat)

# define the transformation from W --> B
g_wb = g_wa*g_ab
# define the body velocity at b
Vb_hat = inverse(g_wb)*derivate(g_wb)
# unhat the body velocity
Vb = unhat(Vb_hat)


# Defining the rotational Inertia Tensors
# there is only rotational inertia in the z-direction
I_rot_c = sym.Matrix([[0, 0, 0], [0, 0, 0], [0, 0, J]])
# for the rectangle with COM at b, it has the same rotational inertia
I_rot_b = sym.Matrix([[0, 0, 0], [0, 0, 0], [0, 0, J]])

# defining the dynamic matrix
dyn_mat = sym.BlockMatrix( [ [m*sym.eye(3), sym.zeros(3,3)], [sym.zeros(3,3), I_rot_c] ] )
dyn_mat = dyn_mat.as_explicit()

# DEFINING THE KINETIC ENERGY #
KE = 0.5*(Vb.T)*dyn_mat*Vb + 0.5*(Vc.T)*dyn_mat*Vc

# DEFINING THE POTENTIAL ENERGY #
rc_bar = sym.Matrix([0, 0, 0, 1])
rwc_bar = g_wc*rc_bar
hc = sym.Matrix([0, 1, 0, 0]).T*rwc_bar

rb_bar = sym.Matrix([0, 0, 0, 1])
rwb_bar = g_wb*rb_bar
hb = sym.Matrix([0, 1, 0, 0]).T*rwb_bar

PE = m*g*hc + m*g*hb


# LAGRANGIAN
L = KE - PE

# determening forcing terms and constraints

# CONSTRAINTS
# defining the transformation from the world frame to frame d
g_wd = g_wa*g_ab*g_bd
# position of the center of frame d wrt frame d
rd_bar = sym.Matrix([0, 0, 0, 1])
# defining the coordinate change from w to d
rwd_bar = g_wd*rd_bar
# height of center of frame d wrt world frame
hd = sym.Matrix([0, 1, 0, 0]).T*rwd_bar
phi1 = hd[0] # the height of the center of d wrt to the world frame has to be 0.

# defining the transformation from the world frame to frame e
g_we = g_wa*g_ac*g_ce
# position of the center of frame e wrt frame e
re_bar = sym.Matrix([0, 0, 0, 1])
# defining the coordinate change from w to e
rwe_bar = g_we*re_bar
# height of center of frame e wrt world frame
he = sym.Matrix([0, 1, 0, 0]).T*rwe_bar
phi2 = he[0] # the height of the center of e wrt to the world frame has to be 0.

lamb1 = sym.symbols('lambda_1')
lamb2 = sym.symbols('lambda_2')

# determening the gradients of constraint 1 for each configuration
dphi1_dx = phi1.diff(x)
dphi1_dy = phi1.diff(y)
dphi1_dtheta1 = phi1.diff(theta1)
dphi1_dtheta2 = phi1.diff(theta2)

# determening the gradients of constraint 2 for each configuration
dphi2_dx = phi2.diff(x)
dphi2_dy = phi2.diff(y)
dphi2_dtheta1 = phi2.diff(theta1)
dphi2_dtheta2 = phi2.diff(theta2)
```

```
# FORCING
# desired trajectories
theta1_desired = np.pi/15 + (np.pi/3)*(sin(t/2))**2
theta2_desired = -np.pi/15 - (np.pi/3)*(sin(t/2))**2

# torques
F1 = -k*(theta1 - theta1_desired)
F2 = -k*(theta2 - theta2_desired)


## EULER-LAGRANGE and CONSTRAINT EQUATIONS

# define subs
subs = {m: 1, k:20, J:1, l:1, g:9.8}

# lhs of EL and constraint
lhs = sym.Matrix([ (L.diff(x_d).diff(t) - L.diff(x)).simplify(), (L.diff(y_d).diff(t) - L.diff(y)).simplify(), (L.diff(theta1_c

# rhs of EL and constraint
# only forcing on theta1 and theta2
rhs = sym.Matrix([ (lamb1*dphi1_dx + lamb2*dphi2_dx),  (lamb1*dphi1_dy + lamb2*dphi2_dy), (lamb1*dphi1_dtheta1 + lamb2*dphi2_dt
# combine into equation to get euler lagrange
eqs = sym.Eq(lhs.subs(subs), rhs.subs(subs))
# determine what we are solving for
c = [lamb1, lamb2, x_dd, y_dd, theta1_dd, theta2_dd]

soln = sym.solve(eqs, c)
display(soln)
```

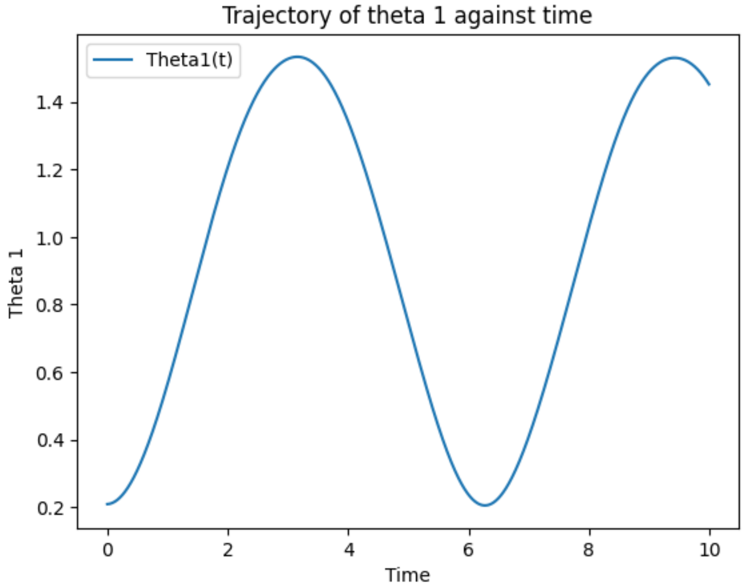
$$\Bigg\{\lambda_1 : -\frac{8.0\cdot 10^{15}\theta_1(t)\sin(\theta_1(t))\sin^2(\theta_2(t))}{100000000000000.0\sin^2(\theta_1(t))\cos^2(\theta_2(t)) - 1.0\cdot 10^{15}\sin^2(\theta_1(t)) + 200000000000000.0\sin(\theta_1(t))\sin(\theta_2(t))\cos(\theta_1(t))\cos(\theta_2(t)) + 10000}$$

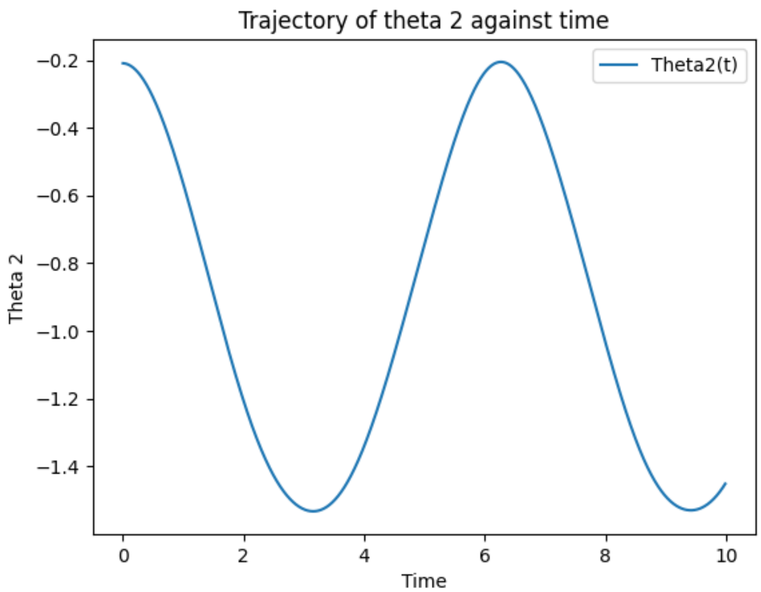
```
xddot = soln[x_dd]
yddot = soln[y_dd]
theta1ddot = soln[theta1_dd]
theta2ddot = soln[theta2_dd]

xddot_simplified = sym.simplify(xddot)
yddot_simplified = sym.simplify(yddot)
theta1ddot_simplified = sym.simplify(theta1ddot)
theta2ddot_simplified = sym.simplify(theta2ddot)

l1 = sym.lambdify([x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t], xddot_simplified, "sympy")
l2 = sym.lambdify([x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t], yddot_simplified, "sympy")
l3 = sym.lambdify([x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t], theta1ddot_simplified, "sympy")
l4 = sym.lambdify([x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t], theta2ddot_simplified, "sympy")


### PLOTTING TRAJECTORY ###
def integrate(f, xt, dt, t):
    """
    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, t)
    k2 = dt * f(xt+k1/2., t)
    k3 = dt * f(xt+k2/2., t)
```

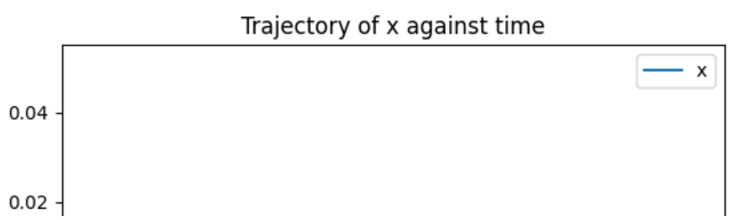
```
    k4 = dt * f(xt+k3, t)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

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), theta(t), xdot(t), thetadot(t) from t=0 to tf
    """
    N = int((max(tspan)-0)/dt)
    x = np.copy(x0)
    tvec = np.linspace(0,max(tspan),N)
    xtraj = np.zeros((len(x0),N))
    t = tspan[0]
    for i in range(N):
        xtraj[:,i]=integrate(f,x,dt,t)
        x = np.copy(xtraj[:,i])
        t = t + dt
    return xtraj

##############################################
def xdd(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t):
  return 0

def ydd(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t):
  return l2(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t)

def theta1dd(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t):
  return l3(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t)

def theta2dd(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t):
  return l4(x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d, t)

def dyn(s, t):
    """
    System dynamics function (extended)

    Parameters
    ============
    s: NumPy array
        s = [x, y, theta1, theta2, x_d, y_d, theta1_d, theta2_d] is the extended system
        state vector, including the position and
        the velocity of the particle

    Return
    ============
    sdot: NumPy array
        time derivative of input state vector,
        sdot = [x_d, y_d, theta1_d, theta2_d, x_dd, y_dd, theta1_dd, theta2_dd]
    """
    return np.array([s[4], s[5], s[6], s[7],
                     xdd(s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], t),
                     ydd(s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], t),
                     theta1dd(s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], t),
                     theta2dd(s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], t)])
```

```
                        theta2dd(s[0], s[1], s[2], s[3], s[4], s[5], s[6], s[7], t)])


# define initial state (theta1 = theta2 = -pi/2 , theta1_d = theta2_d = 0)
# define initial state
x0 = 0
y0 = 1*np.cos(np.pi/15)
theta1_0 = np.pi/15
theta2_0 = -np.pi/15
x_d_0 = 0
y_d_0 = 0
theta1_d_0 = 0
theta2_d_0 = 0
s0 = np.array([x0, y0, theta1_0, theta2_0, x_d_0, y_d_0, theta1_d_0, theta2_d_0])
# simulat from t=0 to 10, since dt=0.01, the returned trajectory
# will have 10/0.01=1000 time steps, each time step contains extended
# system state vector [x(t), xdot(t)]
traj = simulate(dyn, s0, [0, 10], 0.01, integrate)

import matplotlib.pyplot as plt

# determening trajectories of configuration variables and their derivatives

#traj = simulate(dyn, s0, [0, 10], 0.01, integrate)

x_traj = traj[0]
y_traj = traj[1]
theta1_traj = traj[2]
theta2_traj = traj[3]
x_d_traj = traj[4]
y_d_traj = traj[5]
theta1_d_traj = traj[6]
theta2_d_traj = traj[7]


# initial state is already defined above and is part of theta 1 and theta 2 trajectories

# time step array
# 3/0.01 will give 300 time steps
timespan = np.arange(0, 10, 0.01)


plt.plot(timespan, theta1_traj, label='Theta1(t)')
plt.title("Trajectory of theta 1 against time")
plt.ylabel('Theta 1')
plt.xlabel('Time')
plt.legend()
plt.show()

plt.plot(timespan, theta2_traj, label='Theta2(t)')
plt.title('Trajectory of theta 2 against time')
plt.ylabel('Theta 2')
plt.legend()
plt.xlabel('Time')
plt.show()

plt.plot(timespan, x_traj, label='x')
plt.title("Trajectory of x against time")
plt.ylabel('x_traj')
plt.xlabel('Time')
plt.legend()
plt.show()

plt.plot(timespan, y_traj, label='y')
plt.title('Trajectory of y against time')
plt.ylabel('y_traj')
plt.legend()
plt.xlabel('Time')
plt.show()
```

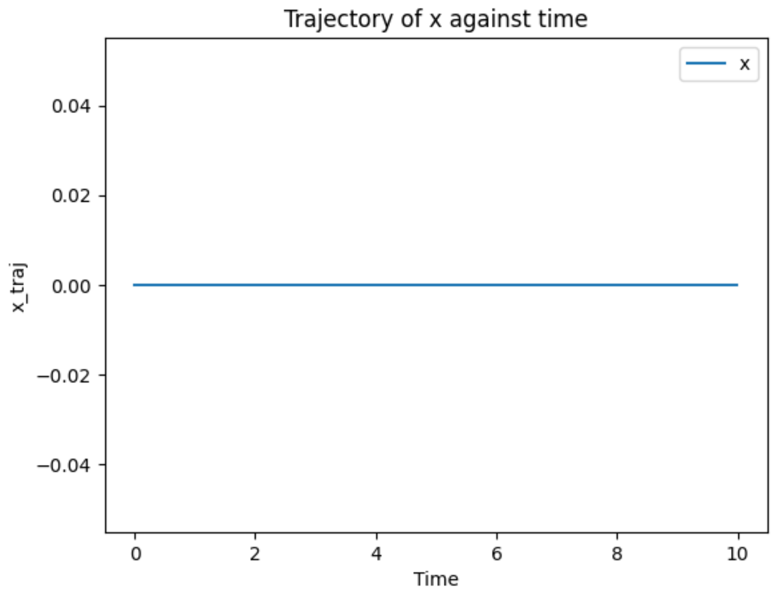
```
Shape of traj:  (8, 1000)
```

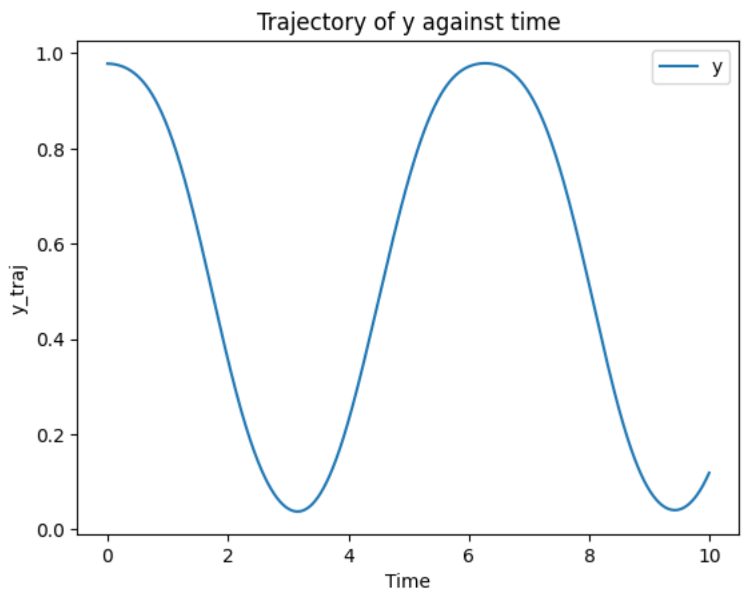






198

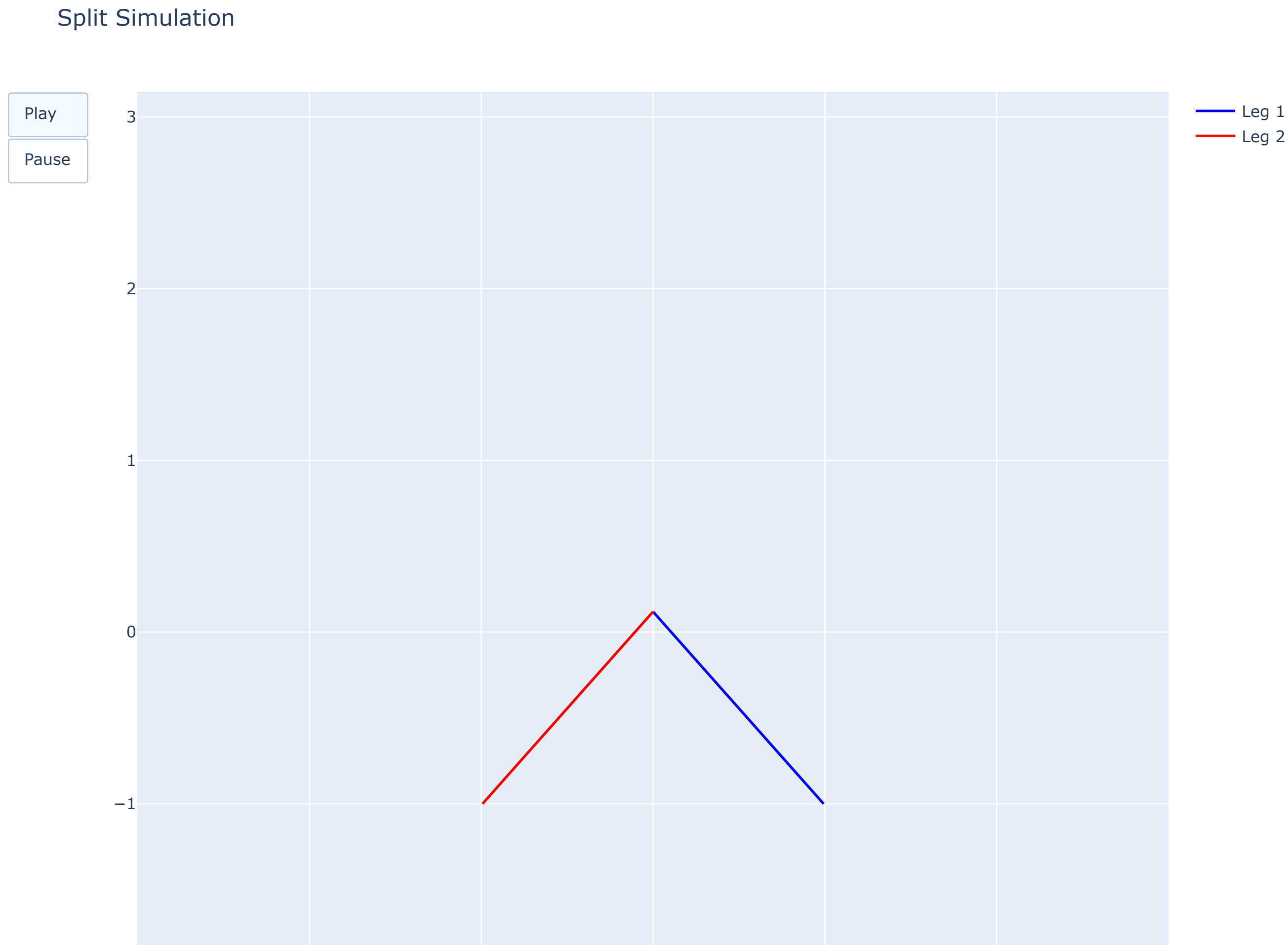





```
def animate_split(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of a person doing a split

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first leg
    L2:
        length of the second leg
    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

```

```python
def animate_split(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of a person doing a split

    Parameters:
    ================================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first leg
    L2:
        length of the second leg
    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 split angle trajectories.
    x1 = 0
    y1 = theta_array[1]
    xx1= L1*np.sin(theta_array[2])
    yy1=-L1*np.cos(theta_array[2])
    xx2=L2*np.sin(theta_array[3])
    yy2=-L2*np.cos(theta_array[3])
    N = len(theta_array[0]) # Need this for specifying length of simulation

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

    ###########################
    # Defining data dictionary
```

```
    # Defining data dictionary.
    # Trajectories are here.
    data=[
        # first two objects correspond to the legs
        dict(name='Leg 1'),
        dict(name='Leg 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='Split 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.
    frames=[dict(data=[
                      # display legs in each animation frame
                      dict(x=[x1, xx1[k]],
                           y=[y1[k], -L1*np.cos(theta_array[0])[k]],
                           mode='lines', line=dict(width=2, color='blue')),
                      dict(x=[x1, xx2[k]],
                           y=[y1[k], -L1*np.cos(theta_array[0])[k]],
                           mode='lines', line=dict(width=2, color='red')),
                     ]) for k in range(N)]

    #################################
    # Merging everything into one dictionary and producing the animation.
    figure1=dict(data=data, layout=layout, frames=frames)
    iplot(figure1)


animate_split(traj, 1, 1, 10)
```



✓ 3s completed at 8:39 AM

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.