ME314 Homework 7 (Template)

*Please note that a **single** PDF file will be the only document that you turn in, which will include your answers to the problems with corresponding derivations and any code used to complete the problems. When including the code, please make sure you also include **code outputs**, and you don't need to include example code. Problems and deliverables that should be included with your submission are shown in **bold**.*

This Juputer Notebook file serves as a template for you to start homework, since we recommend to finish the homework using Jupyter Notebook. You can start with this notebook file with your local Jupyter environment, or upload it to Google Colab. You can include all the code and other deliverables in this notebook Jupyter Notebook supports $\LaTeX$ for math equations, and you can export the whole notebook as a PDF file. But this is not the only option, if you are more comfortable with other ways, feel free to do so, as long as you can submit the homework in a single PDF file.

In [5]:
```
# !pip install --upgrade sympy
import sympy as sym
print(sym.__version__)
```

```
1.7.1
```

Problem 1 (20pts)

Show that if $R \in SO(n)$, then the matrix $A = \frac{d}{dt}(R)R^{-1}$ is skew symmetric.

Turn in: A scanned (or photograph from your phone or webcam) copy of your hand written solution. Or you can use \LaTeX.

see pdf attachment

Problem 2 (20pts)

Show that $\underline{\widehat{\omega}}\,\underline{r}_b = -\underline{\hat{r}}_b\,\underline{\omega}$.

Turn in: A scanned (or photograph from your phone or webcam) copy of your hand written solution. Or you can use \LaTeX.

see pdf attachment

In [6]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/master/biped_simplified.jpg' width=700' height='350'></table>"))
```

Problem 3 (60pts)

As shown in the image, consider one is doing the splits. 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:

$$\theta_1^d(t) = \frac{\pi}{20} + \frac{\pi}{3}\sin^2(\frac{t}{2})$$
$$\theta_2^d(t) = -\frac{\pi}{20} - \frac{\pi}{3}\sin^2(\frac{t}{2})$$

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 $q = [0, L_1\cos(\frac{\pi}{20}), \frac{\pi}{20}, -\frac{\pi}{20}]$. 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 3D world --- the z coordinate is always zero and the rotation is around 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 transformation with 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 returns SE(3) matrices given rotation angle and 2D translation vector, functions for "hat" and "unhat" operations, a function for matrix inverse of SE(3) (which is definitely not the same as the SymPy matrix inverse function), and a function return 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, and include t as a separate symbol to be solved for. For the same reason (the dynamic now explicitly depends 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 animation separately through Canvas. The video should be in ".mp4" format, you can use screen capture or record the screen directly with your phone.

In [7]:
```
import sympy as sym
from sympy import symbols, Function, Matrix, sin, cos, solve, simplify, lambdify, pi, Inverse
from sympy.abc import t
```

config variables set up

In [10]:

```
import math
#define configuration variables
x, y, th1, th2 = symbols(r'x, y, theta1, theta2')

m, l1, l2, J, k, g, w = symbols('m, l1, l2, J, k, g, w')

x = Function(r'x')(t)
y = Function(r'y')(t)
th1 = Function(r'theta1')(t)
th2 = Function(r'theta2')(t)

q = Matrix([x, y, th1, th2])
qdot = q.diff(t)
qddot = qdot.diff(t)

#define the applied torque functions
th_d1 = (math.pi/20) + (math.pi/3)*(sin(t/2)**2)
th_d2 = -(math.pi/20) - (math.pi/3)*(sin(t/2)**2)
trq1 = -k*(th1 - th_d1)
trq2 = -k*(th2 - th_d2)

trqs = Matrix([0, 0, trq1, trq2])
```

transformation matrices set up

In [11]:

```
#set up transformation matrices (is it correct to call these quaternions?)
g_wa = Matrix([
    [1, 0, 0, x],
    [0, 1, 0, y],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

#rotation matrix for left leg
g_wa_cw = Matrix([
    [cos(-th2), -sin(-th2), 0, 0],
    [ sin(-th2), cos(-th2), 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

#rotation matrix for right leg
g_wa_ccw = Matrix([
    [cos(th1), -sin(th1), 0, 0],
    [ sin(th1), cos(th1), 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

#translation from {A} to {C}
g_ac = Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, -l2/2],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

#translation from {A} to {B}
g_ab = Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, -l1/2],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

#translation from {C} to {E}
g_ce = Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, -l2/2],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

#translation from {B} to {D}
g_bd = Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, -l1/2],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_wc = g_wa * g_wa_cw * g_ac
g_wb = g_wa * g_wa_ccw * g_ab

g_we = g_wc * g_ce
g_wd = g_wb * g_bd
```

compute Lagrangian

In [26]:

```
el = symbols('L')
from sympy import Eq

#define unhat function
def unhat(g):
    unhat_output = Matrix([0, 0, 0, 0, 0, 0])

    #select out individual elements to unhat the input matrix
    unhat_output[0, 0] = g[0,3]
    unhat_output[1, 0] = g[1,3]
    unhat_output[2, 0] = g[2,3]
    unhat_output[3, 0] = g[2,1]
    unhat_output[4, 0] = -g[2,0]
    unhat_output[5, 0] = g[1,0]

    return unhat_output

#compute body velocity of left leg
vleft_hat = simplify(Inverse(g_wc) * g_wc.diff(t))
vleft = unhat(vleft_hat)

#compute body velocity of right leg
vright_hat = simplify(Inverse(g_wb) * g_wb.diff(t))
vright = unhat(vright_hat)

#inertia matrix
inertia = Matrix([
    [m, 0, 0, 0, 0, 0],
    [0, m, 0, 0, 0, 0],
    [0, 0, m, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1],
])

KE = ((vleft.T)*inertia*vleft)/2 + ((vright.T)*inertia*vright)/2
V = m*g*(g_wc[1, 3] + g_wb[1,3])

l = simplify(KE[0] - V)

# l = l.subs()
l = Matrix([l])
dldq = l.jacobian(q).T
dldqdot = l.jacobian(qdot).T
dldqdot_dot = dldqdot.diff(t)
```

set up conditions for impact update by solving dL/dqdot, dphi/dq, and Hamiltonian

In [34]:
```
lam1, lam2 = symbols('lambda1, lambda2')

#have to set up impact update equations for each leg individually

#impact update for left leg:
phi1 = g_wc[1,3] - 0.5*cos(th2)
phi1_dot = phi1.diff(t)
phi1_ddot = phi1_dot.diff(t)
phi1_gradient = Matrix([phi1]).jacobian(q).T

#impact update for right leg:
phi2 = g_wb[1,3] - 0.5*cos(th1)
phi2_dot = phi2.diff(t)
phi2_ddot = phi2_dot.diff(t)
phi2_gradient = Matrix([phi2]).jacobian(q).T

#derive Euler-Lagrange equation
rhs = simplify((dldqdot_dot - dldq).row_insert(5, Matrix([phi2_ddot])).row_insert(6, Matrix([phi1_ddot])))
lhs = simplify((phi1_gradient*lam1 + phi2_gradient*lam2 + trqs).row_insert(5, Matrix([0])).row_insert(6, Matrix([0])))
eL = Eq(rhs, lhs)
eL = eL.subs({l1:1, l2:1, w:0.2, m:1, J:1, k:20, g:9.8})
```

In [35]:
```
# qddot_star = Matrix([(x.diff(t)).diff(t), (y.diff(t)).diff(t), (th1.diff(t)).diff(t), (th2.diff(t)).diff(t), lam1, lam2])

soln = solve(eL, [qddot[0], qddot[1], qddot[2], qddot[3], lam1, lam2], dict=True)
```

set up trajectory computation

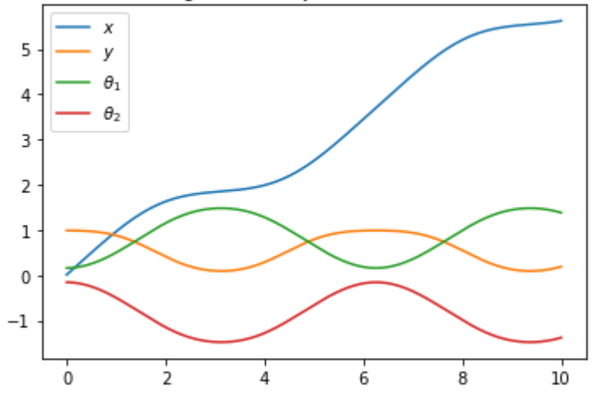
In [51]:
```
qddot_a = Matrix([qdot[0], soln[qddot[0]],
                  qdot[1], soln[qddot[1]],
                  qdot[2], soln[qddot[2]],
                  qdot[3], soln[qddot[3]]])

xx, yy, xxdot, yydot = symbols('xx, yy, xxdot, yydot')
th1, th2, th1dot, th2dot = symbols('th1 th2 th1_dot th2_dot')

qddot_dum = qddot_a.subs({q[0]:xx, q[1]:yy, q[2]:th1, q[3]:th2, q_dot[0]:xxdot, q_dot[1]:yydot, q_dot[2]:th1dot, q_dot[3]:th2dot})

qddot_lamb = sym.lambdify([xx, yy, xxdot, yydot, th1, th1dot, th2, th2dot, t],qddot_dum)
```

In [60]:

```
#SIMULATE
import sympy as sym
from sympy import symbols
import matplotlib.pyplot as plt

def integrate(f, xt, dt, time):
    """
    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
    time

    Return
    ============
    new_xt:
        value of x(t+dt) integrated from x(t)
    """
    k1 = dt * (np.append(f(xt), time))
    k2 = dt * (np.append(f(xt+k1/2.), time))
    k3 = dt * (np.append(f(xt+k2/2.), time))
    k4 = dt * (np.append(f(xt+k3), time))
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    new_xt[-1] = time
    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)
```

```
    xtraj = np.zeros((len(x0),N))
    time = 0
    for i in range(N):
        time = time+dt
        xtraj[:,i]=integrate(f,x,dt,time)
        x = np.copy(xtraj[:,i])
    return xtraj

##############################################

def dyn(s):
    return qddot_lamb(s[0],s[1],s[2],s[3],s[4],s[5],s[6],s[7],s[8])

# define initial state
s0 = np.array([0, 0, L1 * cos(np.pi/20), 0, pi/20, 0, -pi/20, 0, 0])
traj = simulate(dyn, s0, [0,10], 0.01, integrate)
print('shape of traj: ', traj.shape)
print("")
N = int(10/0.01)
time_axis = np.linspace(0,10,N)

plt.plot(time_axis,traj[0])
plt.plot(time_axis,traj[2])
plt.plot(time_axis,traj[4])
plt.plot(time_axis,traj[6])
plt.title('Configuration Trajectories Over Time')
plt.legend([r'$x$',r'$y$',r'$\theta_1$',r'$\theta_2$'])
plt.show()
```

```
shape of traj:  (9, 1000)
```



copy animation functions from prev homework and modify

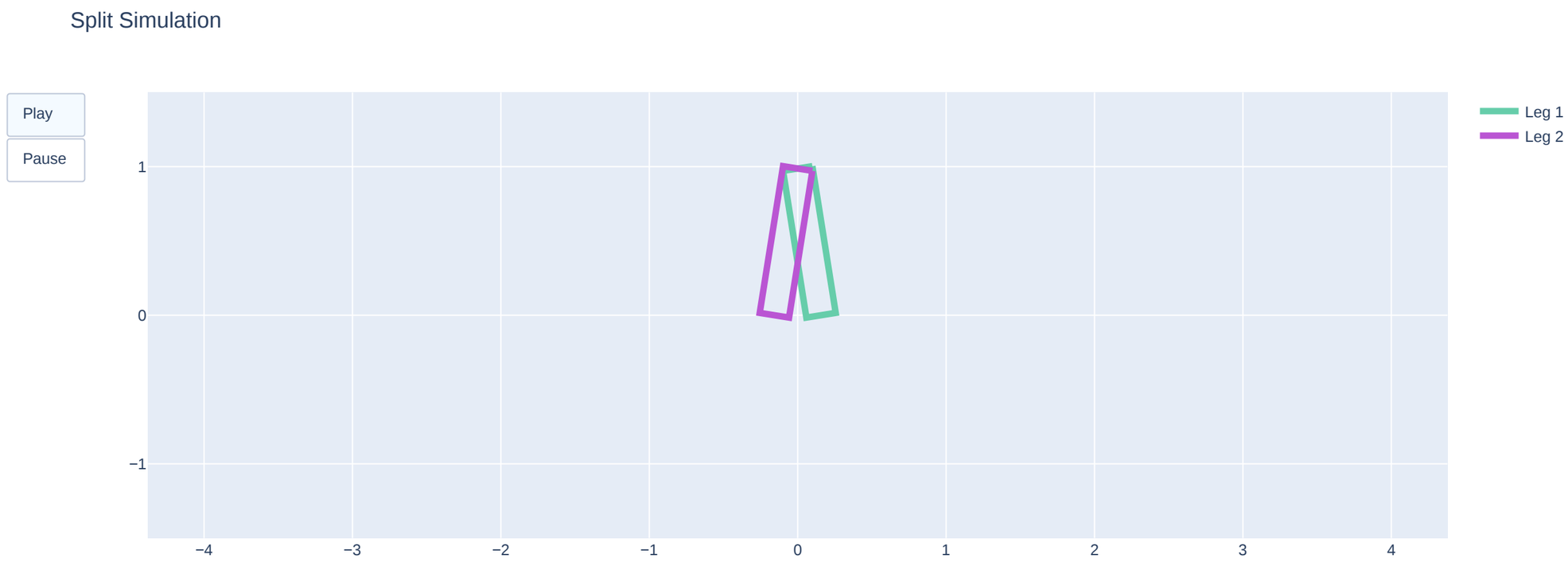
In [56]:

```
def animate_the_splits(theta_array, l=1 , w=0.2, T=10):
    ###############################
    # Imports required for animation.
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML

    ######################
    # 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)

    ###############################################

    xx1 = (w/2)*np.cos(theta_array[0])
    yy1 = l*np.cos(theta_array[0])+(w/2)*np.sin(theta_array[0])
    xx2 = l*np.sin(theta_array[0])+(w/2)*np.cos(theta_array[0])
    yy2 = (w/2)*np.sin(theta_array[0])
    xx3 = l*np.sin(theta_array[0])-(w/2)*np.cos(theta_array[0])
    yy3 = -(w/2)*np.sin(theta_array[0])
    xx4 = -(w/2)*np.cos(theta_array[0])
    yy4 = l*np.cos(theta_array[0])-(w/2)*np.sin(theta_array[0])
    xx5 = (w/2)*np.cos(theta_array[1])
    yy5 = l*np.cos(theta_array[1])+(w/2)*np.sin(theta_array[1])
    xx6 = l*np.sin(theta_array[1])+(w/2)*np.cos(theta_array[1])
    yy6 = (w/2)*np.sin(theta_array[1])
    xx7 = l*np.sin(theta_array[1])-(w/2)*np.cos(theta_array[1])
    yy7 = -(w/2)*np.sin(theta_array[1])
    xx8 = -(w/2)*np.cos(theta_array[1])
    yy8 = l*np.cos(theta_array[1])-(w/2)*np.sin(theta_array[1])

    N = len(theta_array[0]) # Need this for specifying length of simulation

    ###################################
    # Using these to specify axis limits.
    xm= -1
    xM= 1
    ym= -1.5
    yM= 1.5

    ############################
    # Defining data dictionary.
    # Trajectories are here.
    data=[dict(x=xx1, y=yy1,
               mode='lines', name='Leg 1',
               line=dict(width=2, color='blue')
              ),
          dict(x=xx5, y=yy5,
               mode='lines', name='Leg 2',
               line=dict(width=2, color='red')
              ),
        ]

    ###############################
    # Preparing simulation layout.
    # Title and axis ranges are here.
```

```
            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'}
                                      ]
                          }]
           )

    ########################################
    frames=[dict(data=[dict(x=[xx1[k],xx2[k],xx3[k],xx4[k],xx1[k]],
                y=[yy1[k],yy2[k],yy3[k],yy4[k],yy1[k]],
                mode='lines',
                line=dict(color='mediumaquamarine', width=5)
                ),
            dict(x=[xx5[k],xx6[k],xx7[k],xx8[k],xx5[k]],
                y=[yy5[k],yy6[k],yy7[k],yy8[k],yy5[k]],
                mode='lines',
                line=dict(color='mediumorchid', width=5)
                )
                ]) for k in range(N)]

    ########################################
    figure1=dict(data=data, layout=layout, frames=frames)
    iplot(figure1)
```

In [57]:

```
# Here we animate
theta_array = np.array([traj[4],traj[6]])
animate_the_splits(theta_array,1,0.2,10)
```



In [ ]: