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) and include simplified code outputs (e.g. `.simplify()`).
- 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 secti
# ctrl+'/' on your and keyboard. Run it before you start programming, this will ena
# LaTeX "display()" function for you. If you're using the local Jupyter environment
################################################################################
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=Te
    javascript(url=url)
    return sym.printing.latex(exp,**options)
sym.init_printing(use_latex="mathjax",latex_printer=custom_latex_printer)
```

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.

```
# handwritten solution attached to end of pdf
```

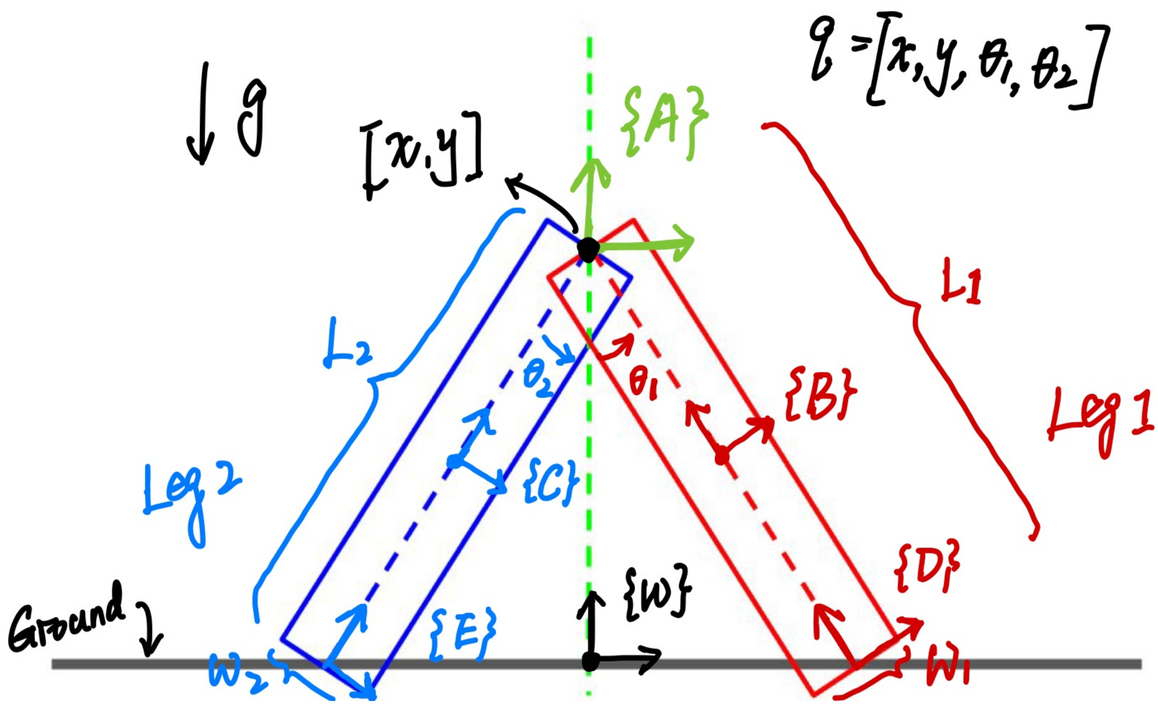
Problem 2 (20pts)

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

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

```
# handwritten solution attached to end of pdf
```

```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/raw/r
```



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.

```
## Your code goes here
```

```
import numpy as np
import sympy as sym
from sympy import Function, symbols, lambdify, solve, Eq, Matrix
import matplotlib.pyplot as plt


def integrate(f, xt, dt, tt):
  ## slightly modified as referenced in hint 3
    """
    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, tt)
    k2 = dt * f(xt+k1/2., tt+dt/2.)
    k3 = dt * f(xt+k2/2., tt+dt/2.)
    k4 = dt * f(xt+k3, tt+dt)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

def simulate(f, x0, tspan, dt, integrate):
  ## slightly modified as referenced in hint 3

    """
    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, tvec[i])
        x = np.copy(xtraj[:,i])
    return xtraj
```

```
### helper functions ###

# helper functions for getting SE3 with given theta, x, y(hint 2)
def SE3_sym(theta, x, y):
  SE3_sym = sym.Matrix([[sym.cos(theta), -sym.sin(theta), 0.0, x],
                       [sym.sin(theta), sym.cos(theta), 0.0, y],
                       [0.0, 0.0, 1.0, 0.0],
                       [0.0, 0.0, 0.0, 1.0]])
  return SE3_sym

def SE3_np(theta, x, y):
  SE3_np = sym.Matrix([[np.cos(theta), -np.sin(theta), 0.0, x],
                       [np.sin(theta), np.cos(theta), 0.0, y],
                       [0.0, 0.0, 1.0, 0.0],
                       [0.0, 0.0, 0.0, 1.0]])
  return SE3_np

# helper function for getting the inverse of the input matrix (hint 2)
def inverse_matrix(mat):
  R = sym.Matrix([[mat[0,0], mat[0,1], mat[0,2]],
                  [mat[1,0], mat[1,1], mat[1,2]],
                  [mat[2,0], mat[2,1], mat[2,2]]])

  p = sym.Matrix([mat[0,3], mat[1,3], mat[2,3]])

  R_inverse = R.T
  p_inverse = -R_inverse * p

  inverse_mat = sym.Matrix([[R_inverse[0,0], R_inverse[0,1], R_inverse[0,2], p_inve
                            [R_inverse[1,0], R_inverse[1,1], R_inverse[1,2], p_inve
                            [R_inverse[2,0], R_inverse[2,1], R_inverse[2,2], p_inve
                            [0.0, 0.0, 0.0, 1.0]])

  return inverse_mat


# helper function for unhatting input matrix (hint 2)
def unhat(mat):
  unhat_vec = sym.Matrix([mat[0,3], mat[1,3], mat[2,3], mat[2,1], mat[0,2], mat[1,(
  return unhat_vec



```

```
## constants ##
```

```
L1 = 1
L2 = 1
W1 = 0.2
W2 = 0.2
m1 = 1
m2 = 1
J1 = 1
J2 = 1
g = 9.8
k1 = 20


## variables ##
t = sym.symbols('t')
lam1 = sym.symbols(r'\lambda_1')
lam2 = sym.symbols(r'\lambda_2')
lam = sym.Matrix([lam1, lam2])

x = sym.Function(r'x')(t)
y = sym.Function(r'y')(t)

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

q = sym.Matrix([x, y, theta1, theta2])
qdot = q.diff(t)
qddot = qdot.diff(t)
```

```
## transformations ##
## referencing lecture
g_wa = SE3_sym(0,q[0], q[1])
#display(g_wa)

g_wb = g_wa * SE3_sym(theta1, 0, 0) * SE3_sym(0, 0, -L1/2.0)
#display(g_wb)

g_wc = g_wa * SE3_sym(theta2, 0, 0) * SE3_sym(0, 0, -L2/2.0)
#display(g_wc)

# constraint transformations
g_wb_base = g_wa * SE3_sym(theta1, 0, 0) * SE3_sym(0, 0, -L1)
g_wc_base = g_wa * SE3_sym(theta2, 0, 0) * SE3_sym(0, 0, -L2)





## body velocities ##
Vb_ab = unhat(inverse_matrix(g_wb) * g_wb.diff(t)) # body velocity for b
```

```
Vb_ab = unhat(inverse_matrix(g_wb) * g_wb.diff(t)) # body velocity for b
#display(Vb_ab)
Vb_ac = unhat(inverse_matrix(g_wc) * g_wc.diff(t)) # body velocity for c
#display(Vb_ac)


## body inertia matricies (lecture notes)##
I_b = sym.Matrix([[m1, 0.0, 0.0, 0.0, 0.0, 0.0],
                  [0.0, m1, 0.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, m1, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, J1]])

I_c = sym.Matrix([[m2, 0.0, 0.0, 0.0, 0.0, 0.0],
                  [0.0, m2, 0.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, m2, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                  [0.0, 0.0, 0.0, 0.0, 0.0, J2]])


## Potential Energies ##
PE_b = m1 * g * (g_wb * sym.Matrix([0.0, 0.0, 0.0, 1.0]))[1]
PE_c = m2 * g * (g_wc * sym.Matrix([0.0, 0.0, 0.0, 1.0]))[1]

## Kinetic Energies ##
KE_b = 0.5 * (Vb_ab.T * I_b * Vb_ab)[0]
KE_c = 0.5 * (Vb_ac.T * I_c * Vb_ac)[0]

## Lagrangian ##
L = KE_b + KE_c - (PE_b + PE_c)


## constraint ##
phi1 = (g_wb_base * sym.Matrix([0.0, 0.0, 0.0, 1.0]))[1]
#display(phi1)
phi2 = (g_wc_base * sym.Matrix([0.0, 0.0, 0.0, 1.0]))[1]
#display(phi2)
phi = sym.Matrix([phi1, phi2])
#display(phi)
```

```
## Euler-Lagrange Equations ##
dLdq = L.diff(q)
dLdqdot = L.diff(qdot)
d_dLdqdot_dt = dLdqdot.diff(t)

## desired thetas ##
```

```
## desired thetas ##
theta1_d = sym.pi/15.0 + sym.pi/3.0 * sym.sin(t/2.0)**2
theta2_d = -sym.pi/15.0 - sym.pi/3.0 * sym.sin(t/2.0)**2

## torques ##
F_theta1 = -k1 * (theta1 - theta1_d)
F_theta2 = -k1 * (theta2 - theta2_d)
F_ext = sym.Matrix([0.0, 0.0, F_theta1, F_theta2])


EL_eqns_LHS1 = sym.simplify(d_dLdqdot_dt - dLdq)
EL_eqns_LHS2 = phi.diff(t,t)
EL_eqns_LHS = sym.Matrix([EL_eqns_LHS1, EL_eqns_LHS2])

EL_eqns_RHS1 = phi1.diff(q) * lam1 + phi2.diff(q) * lam2 + F_ext
EL_eqns_RHS = sym.Matrix([EL_eqns_RHS1, 0.0, 0.0])

EL_eqns = sym.Eq(EL_eqns_LHS, EL_eqns_RHS)
display(EL_eqns)

EL_solns = sym.solve(EL_eqns, [*qddot, *lam])
#display(EL_solns)

xddot = sym.lambdify([*q, *qdot, t], EL_solns[qddot[0]])
yddot = sym.lambdify([*q, *qdot, t], EL_solns[qddot[1]])
theta1ddot = sym.lambdify([*q, *qdot, t], EL_solns[qddot[2]])
theta2ddot = sym.lambdify([*q, *qdot, t], EL_solns[qddot[3]])
```

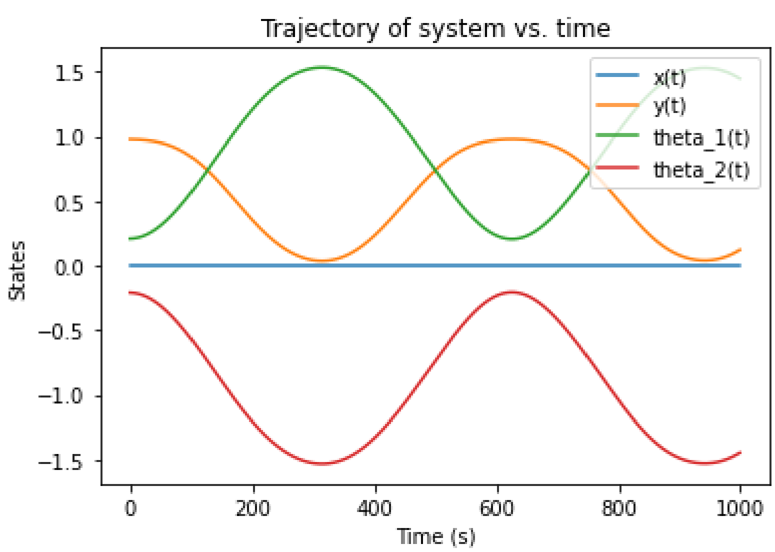
$$\begin{bmatrix} -0.5\sin(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 - 0.5\sin(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.5\cos(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t) + 0.5\cos(\theta_2 \\ 0.5\sin(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t) + 0.5\sin(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t) + 0.5\cos(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5\cos(\theta_2(t))\left(\frac{d}{dt} \\ 0.5\sin(\theta_1(t))\frac{d^2}{dt^2}y(t) + 4.9\sin(\theta_1(t)) + 0.5\cos(\theta_1(t))\frac{d^2}{dt^2}x(t) + 1.25\frac{d^2}{dt^2} \\ 0.5\sin(\theta_2(t))\frac{d^2}{dt^2}y(t) + 4.9\sin(\theta_2(t)) + 0.5\cos(\theta_2(t))\frac{d^2}{dt^2}x(t) + 1.25\frac{d^2}{dt^2} \\ 1.0\left(\sin(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t) + \cos(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + \frac{d^2}{dt^2}y(t)\right) \\ 1.0\left(\sin(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t) + \cos(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 + \frac{d^2}{dt^2}y(t)\right) \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1.0\lambda_1 + 1.0\lambda_2 \\ 1.0\lambda_1\sin(\theta_1(t)) - 20\theta_1(t) + 6.66666666666667\pi\sin^2(0.5t) + 1.33333333333333\pi \\ 1.0\lambda_2\sin(\theta_2(t)) - 20\theta_2(t) - 6.66666666666667\pi\sin^2(0.5t) - 1.33333333333333\pi \\ 0.0 \end{bmatrix}$$

```
#sym.simplify(EL_solns[qddot[0]])
#sym.simplify(EL_solns[qddot[1]])
#sym.simplify(EL_solns[qddot[2]])
sym.simplify(EL_solns[qddot[3]])
```

$$\frac{2400.0\theta_1(t)\sin(\theta_1(t))\sin(\theta_2(t)) + 2400.0\theta_2(t)\sin^2(\theta_1(t)) + 2513.27412287183\sin^2(0.5t)\sin^2}{}$$

```
def dyn(s, t):
  return np.array([s[4], s[5], s[6], s[7], xddot(*s, t), yddot(*s, t), theta1ddot(*


IC = [0.0, L1*np.cos(np.pi/15.0), np.pi/15.0, -np.pi/15.0, 0.0, 0.0, 0.0, 0.0]
traj = simulate(dyn, IC, [0,10], 0.01, integrate)
traj.shape

plt.figure()
plt.plot(traj[0:4].T)
plt.title('Trajectory of system vs. time')
plt.legend(['x(t)', 'y(t)', 'theta_1(t)', 'theta_2(t)'], loc='upper right')
plt.xlabel('Time (s)')
plt.ylabel('States')
plt.show()
```




```
def animate_legs(config_array,L1=1.0,L2=1.0,W1=0.2,W2=0.2,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
    config_array:
        trajectory of x, y, theta1 and theta2, should be a NumPy array with
        shape of (4,N)
    L1:
        length of the first leg
    L2:
        length of the second leg
    W1:
        width of first leg
```

```
W2:
    width of 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 pendulum angle trajectories.
# xx1=L1*np.sin(config_array[0])
# yy1=-L1*np.cos(_array[0])
# xx2=xx1+L2*np.sin(theta_array[0]+theta_array[1])
# yy2=yy1-L2*np.cos(theta_array[0]+theta_array[1])

# data from config array
N = len(config_array[0]) # Need this for specifying length of simulation
x_array = config_array[0]
y_array = config_array[1]
theta1_array = config_array[2]
theta2_array = config_array[3]



###############################################
# 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])
```

```
    # x_axis = np.array([0.3, 0.0])
    # y_axis = np.array([0.0, 0.3])
    # # Use homogeneous tranformation to transfer these two axes/points
    # # back to the fixed frame

    # need to add b, c, and d frames
    # frame_a_x_axis = np.zeros((2,N))
    # frame_a_y_axis = np.zeros((2,N))
    # frame_b_x_axis = np.zeros((2,N))
    # frame_b_y_axis = np.zeros((2,N))
    # frame_c_x_axis = np.zeros((2,N))
    # frame_c_y_axis = np.zeros((2,N))
    # frame_d_x_axis = np.zeros((2,N))
    # frame_d_y_axis = np.zeros((2,N))


    # 4 points per leg (x and y for each), therefore 16 arrays needed
    leg1_point1_xarray = np.zeros(N, dtype=np.float32)
    leg1_point1_yarray = np.zeros(N, dtype=np.float32)
    leg1_point2_xarray = np.zeros(N, dtype=np.float32)
    leg1_point2_yarray = np.zeros(N, dtype=np.float32)
    leg1_point3_xarray = np.zeros(N, dtype=np.float32)
    leg1_point3_yarray = np.zeros(N, dtype=np.float32)
    leg1_point4_xarray = np.zeros(N, dtype=np.float32)
    leg1_point4_yarray = np.zeros(N, dtype=np.float32)

    leg2_point1_xarray = np.zeros(N, dtype=np.float32)
    leg2_point1_yarray = np.zeros(N, dtype=np.float32)
    leg2_point2_xarray = np.zeros(N, dtype=np.float32)
    leg2_point2_yarray = np.zeros(N, dtype=np.float32)
    leg2_point3_xarray = np.zeros(N, dtype=np.float32)
    leg2_point3_yarray = np.zeros(N, dtype=np.float32)
    leg2_point4_xarray = np.zeros(N, dtype=np.float32)
    leg2_point4_yarray = np.zeros(N, dtype=np.float32)



    for i in range(N): # iteration through each time step
        # evaluate homogeneous transformation
        # t_wa = np.array([[np.cos(theta_array[0][i]), -np.sin(theta_array[0][i]),
        #                  [np.sin(theta_array[0][i]),  np.cos(theta_array[0][i]),
        #                  [                         0,                           0,
        # # transfer the x and y axes in body frame back to fixed frame at
        # # the current time step
        # frame_a_x_axis[:,i] = t_wa.dot([x_axis[0], x_axis[1], 1])[0:2]
        # frame_a_y_axis[:,i] = t_wa.dot([y_axis[0], y_axis[1], 1])[0:2]

        # # now have to transfer the x and y axes from a to b, b to c, and c to d
        # t_ab = np.array([[1,0,0], [0,1,-L1], [0,0,1]])
        # frame_b_x_axis[:,i] = (t_wa.dot(t_ab)).dot([x_axis[0], x_axis[1], 1])[0:2
        # frame_b_y_axis[:,i] = (t_wa.dot(t_ab)).dot([y_axis[0], y_axis[1], 1])[0:2

```

```
        # t_bc = np.array([[np.cos(theta_array[1][i]), -np.sin(theta_array[1][i]),
        #                   [np.sin(theta_array[1][i]),  np.cos(theta_array[1][i]),
        #                   [                        0,                         0,
        # frame_c_x_axis[:,i] = (t_wa.dot(t_ab.dot(t_bc))).dot([x_axis[0], x_axis[1
        # frame_c_y_axis[:,i] = (t_wa.dot(t_ab.dot(t_bc))).dot([y_axis[0], y_axis[1

        # t_cd = t_ab = np.array([[1,0,0], [0,1,-L2], [0,0,1]])
        # frame_d_x_axis[:,i] = (t_wa.dot(t_ab.dot(t_bc.dot(t_cd)))).dot([x_axis[0]
        # frame_d_y_axis[:,i] = (t_wa.dot(t_ab.dot(t_bc.dot(t_cd)))).dot([y_axis[0]

        ## transformations
        np_g_wa = SE3_np(0, x_array[i], y_array[i])
        np_g_wb = np_g_wa * (SE3_np(theta1_array[i], 0, 0) * (SE3_np(0, 0, -L1/2.0)
        np_g_wc = np_g_wa * (SE3_np(theta2_array[i], 0.0, 0.0) * (SE3_np(0.0, 0.0,

        leg1_point1 = np_g_wb.dot(np.array([W1/2.0, L1/2.0, 0.0, 1.0]))
        leg1_point1_xarray[i] = leg1_point1[0]
        leg1_point1_yarray[i] = leg1_point1[1]

        leg1_point2 = np_g_wb.dot(np.array([-W1/2.0, L1/2.0, 0.0, 1.0]))
        leg1_point2_xarray[i] = leg1_point2[0]
        leg1_point2_yarray[i] = leg1_point2[1]

        leg1_point3 = np_g_wb.dot(np.array([-W1/2.0, -L1/2.0, 0.0, 1.0]))
        leg1_point3_xarray[i] = leg1_point3[0]
        leg1_point3_yarray[i] = leg1_point3[1]

        leg1_point4 = np_g_wb.dot(np.array([W1/2.0, -L1/2.0, 0.0, 1.0]))
        leg1_point4_xarray[i] = leg1_point4[0]
        leg1_point4_yarray[i] = leg1_point4[1]

        leg2_point1 = np_g_wc.dot(np.array([W2/2.0, L2/2.0, 0.0, 1.0]))
        leg2_point1_xarray[i] = leg2_point1[0]
        leg2_point1_yarray[i] = leg2_point1[1]

        leg2_point2 = np_g_wc.dot(np.array([-W2/2.0, L2/2.0, 0.0, 1.0]))
        leg2_point2_xarray[i] = leg2_point2[0]
        leg2_point2_yarray[i] = leg2_point2[1]

        leg2_point3 = np_g_wc.dot(np.array([-W2/2.0, -L2/2.0, 0.0, 1.0]))
        leg2_point3_xarray[i] = leg2_point3[0]
        leg2_point3_yarray[i] = leg2_point3[1]

        leg2_point4 = np_g_wc.dot(np.array([W2/2.0, -L2/2.0, 0.0, 1.0]))
        leg2_point4_xarray[i] = leg2_point4[0]
        leg2_point4_yarray[i] = leg2_point4[1]




    ####################################
    # Using these to specify axis limits.
```

```
    xm = -1.5 #np.min(xx1)-0.5
    xM = 1.5 #np.max(xx1)+0.5
    ym = -1.5 #np.min(yy1)-2.5
    yM = 1.5 #np.max(yy1)+1.5

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

        # have to add B, C, and D frames
        # dict(name='Arm'),
        # dict(name='Mass 1'),
        # dict(name='Mass 2'),
        # dict(name='World Frame X'),
        # dict(name='World Frame Y'),
        # dict(name='A Frame X Axis'),
        # dict(name='A Frame Y Axis'),
        # dict(name='B Frame X Axis'),
        # dict(name='B Frame Y Axis'),
        # dict(name='C Frame X Axis'),
        # dict(name='C Frame Y Axis'),
        # dict(name='D Frame X Axis'),
        # dict(name='D Frame Y Axis'),

        dict(name='Leg 1'),
        dict(name='Leg 2'),
        dict(x=[xm, xM], y=[0,0],
             mode='lines', name='Ground',
             line=dict(color="brown",width=3),
             ),


        # 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 tha
        # "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,scaleanc
                title='Simulation of Legs of Body',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                             'args': [None, {'frame': {'duration': 1
                                            {'args': [[None], {'frame': {'duration':
                                             'transition': {'duration': 0}}],'label'
                                          ]
                              }]
               )

    ########################################
    # Defining the frames of the simulation.
    # This is what draws the lines from
    # joint to joint of the pendulum.
    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)
                      #  dict(x=[0,xx1[k],xx2[k]],
                      #       y=[0,yy1[k],yy2[k]],
                      #       mode='lines',
                      #       line=dict(color='orange', 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)),
                      #  # display x and y axes of the fixed frame in each animatic
                      #  dict(x=[0,x_axis[0]],
                      #       y=[0,x_axis[1]],
                      #       mode='lines',
                      #       line=dict(color='green', width=3),
                      #       ),
                      #  dict(x=[0,y_axis[0]],
                      #       y=[0,y_axis[1]],
                      #       mode='lines',
                      #       line=dict(color='red', width=3),
                      #       ),
```

```
                        #  # display x and y axes of the {A} frame in each animation
                        #  dict(x=[0, frame_a_x_axis[0][k]],
                        #       y=[0, frame_a_x_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='green', width=3),
                        #       ),
                        #  dict(x=[0, frame_a_y_axis[0][k]],
                        #       y=[0, frame_a_y_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='red', width=3),
                        #       ),

                        #  # display x and y axes of the {B} frame in each animation
                        #  dict(x=[xx1[k], frame_b_x_axis[0][k]],
                        #       y=[yy1[k], frame_b_x_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='green', width=3),
                        #       ),
                        #  dict(x=[xx1[k], frame_b_y_axis[0][k]],
                        #       y=[yy1[k], frame_b_y_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='red', width=3),
                        #       ),

                        #  # display x and y axes of the {C} frame in each animation
                        #  dict(x=[xx1[k], frame_c_x_axis[0][k]],
                        #       y=[yy1[k], frame_c_x_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='green', width=3),
                        #       ),
                        #  dict(x=[xx1[k], frame_c_y_axis[0][k]],
                        #       y=[yy1[k], frame_c_y_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='red', width=3),
                        #       ),

                        #   # display x and y axes of the {D} frame in each animatior
                        #   # now use xx2 and yy2 for D frame
                        #  dict(x=[xx2[k], frame_d_x_axis[0][k]],
                        #       y=[yy2[k], frame_d_x_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='green', width=3),
                        #       ),
                        #  dict(x=[xx2[k], frame_d_y_axis[0][k]],
                        #       y=[yy2[k], frame_d_y_axis[1][k]],
                        #       mode='lines',
                        #       line=dict(color='red', width=3),
                        #       ),

                         ## from point1 --> point2 --> point3 --> point4 --> point 1
                            dict(x=[leg1_point1_xarray[k],
                                    leg1_point2_xarray[k],
```

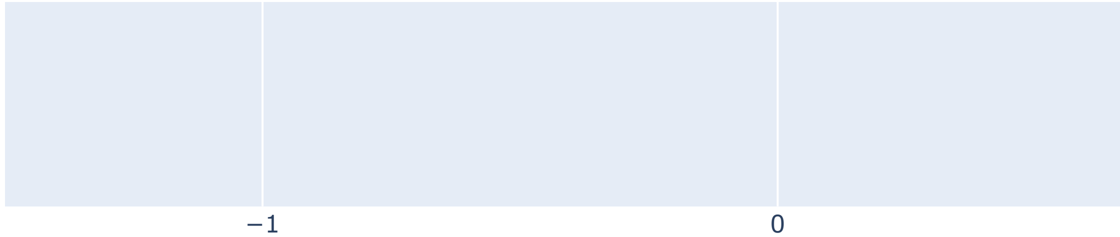
```
                                          leg1_point2_xarray[k],
                                          leg1_point3_xarray[k],
                                          leg1_point4_xarray[k],
                                          leg1_point1_xarray[k]],
                                        y=[leg1_point1_yarray[k],
                                          leg1_point2_yarray[k],
                                          leg1_point3_yarray[k],
                                          leg1_point4_yarray[k],
                                          leg1_point1_yarray[k]],
                                        mode='lines',
                                        line=dict(color='blue',width=3),),

                                   dict(x=[leg2_point1_xarray[k],
                                          leg2_point2_xarray[k],
                                          leg2_point3_xarray[k],
                                          leg2_point4_xarray[k],
                                          leg2_point1_xarray[k]],
                                        y=[leg2_point1_yarray[k],
                                          leg2_point2_yarray[k],
                                          leg2_point3_yarray[k],
                                          leg2_point4_yarray[k],
                                          leg2_point1_yarray[k]],
                                        mode='lines',
                                        line=dict(color='green',width=3),),



                             ]) for k in range(N)]

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

```
animate_legs(traj)
```

```
/usr/local/lib/python3.7/dist-packages/sympy/matrices/matrices.py:1357: SymPyDep:
```

```
Dot product of non row/column vectors has been deprecated since SymPy
1.2. Use * to take matrix products instead. See
https://github.com/sympy/sympy/issues/13815 for more info.
```

Simulation of Legs of Body



✓ 26s completed at 6:24 PM

Brad Goudie

MECH_ENG 314 Homework 7

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

$R = e^{\hat{\omega}\theta}$ (lecture notes)

$\frac{d}{dt}(R) = \frac{d}{dt} e^{\hat{\omega}\theta} = \frac{\partial}{\partial\theta}(e^{\hat{\omega}\theta}) \cdot \frac{d\theta}{dt}$

$= \hat{\omega} e^{\hat{\omega}\theta} \cdot \dot{\theta} = R\hat{\omega}\dot{\theta}$

$\hat{\omega}$ is skew-symmetric matrix

$\hat{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$

$A = \frac{d}{dt}(R)R^{-1} = R\hat{\omega}\dot{\theta} \cdot R^{-1} = \hat{\omega}\dot{\theta}\underbrace{R \cdot R^{-1}}_{I}$

$\boxed{A = \hat{\omega}\dot{\theta}}$ A skew-symmetric matrix because of $\hat{\omega}$ ✓

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

$\hat{\omega} r_b = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} -\omega_3 r_2 + \omega_2 r_3 \\ \omega_3 r_1 - \omega_1 r_3 \\ -\omega_2 r_1 + \omega_1 r_2 \end{bmatrix}$

$-\hat{r}_b \omega = \begin{bmatrix} 0 & r_3 & -r_2 \\ -r_3 & 0 & r_1 \\ r_2 & -r_1 & 0 \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{bmatrix} r_3\omega_2 - r_2\omega_3 \\ -r_3\omega_1 + r_1\omega_3 \\ r_2\omega_1 - r_1\omega_2 \end{bmatrix}$

equal!

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