ME314 Homework 7

Sean Morton
Collaborators: Noah Yi, Sophia Schiffer

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 drive (click "File" -> "Save a copy in Drive"), and then start to edit it.

In [1]:
```
##############################################################################################
# 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
import numpy as np
import matplotlib.pyplot as plt
import time
import tqdm

# 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)
```

In [119]:

```python
#helper functions

def SOnAndRnToSEn(R, p):

    #do type checking for the matrix types
    if type(R) == list:
        R = np.matrix(R)

    n = R.shape[0]
    if ((R.shape[0] != R.shape[1]) or                               #R is NP array or Sym matrix
        ((type(p) is np.ndarray and max(p.shape) != R.shape[0]) or  #p is NP array and shape mismatch or..
          ((isinstance(p, list) or isinstance(p, sym.Matrix)) and
            ( len(p) != R.shape[0] ))    )  ):                      #p is Sym matrix or "list" and shape mismatch
        raise Exception(f"Shape of R {R.shape} and p ({len(p)}) mismatch; exiting.")
        return None

    #construct a matrix based on returning a Sympy Matrix
    if isinstance(R, sym.Matrix) or isinstance(p, sym.Matrix):
        #realistically one of these needs to be symbolic to do this

        if isinstance(R, np.ndarray) or isinstance(p, np.ndarray):
            raise Exception("R and p cannot mix/match Sympy and Numpy types")
            return None

        G = sym.zeros(n+1)
        G[:n, n] = sym.Matrix(p)

    #construct a matrix based on returning a Numpy matrix
    elif isinstance(R, np.ndarray) or isinstance(R, list):
        G = np.zeros([n+1, n+1])
        G[:n, n] = np.array(p).T

    else:
        raise Exception("Error: type not recognized")
        return None

    G[:n,:n] = R
    G[-1,-1] = 1
    return G

def SEnToSOnAndRn(SEnmat):
    '''Decomposes a SE(n) vector into its rotation matrix and displacement components.
    '''
    if isinstance(SEnmat, list):
        SEnmat = np.matrix(SEnmat)
    n = SEnmat.shape[0]
    return SEnmat[:(n-1), :(n-1)], SEnmat[:(n-1), n-1]

#test cases removed to decrease size of submission file
```

In [79]:

```python
def HatVector3(w):
    '''Turns a vector in R3 to a skew-symmetric matrix in so(3).
    Works with both Sympy and Numpy matrices.
    '''
    #create different datatype representations based on type of w
    if isinstance(w, list) or isinstance(w, np.ndarray) \
        or isinstance(w, np.matrix):
        f = np.array
    elif isinstance(w, sym.Matrix): #NP and Sym
        f = sym.Matrix

    return f([
        [    0, -w[2],  w[1]],
        [ w[2],     0, -w[0]],
        [-w[1],  w[0],     0]
    ])

###

def UnhatMatrix3(w_hat):
    '''Turns a skew-symmetric matrix in so(3) into a vector in R3.
    '''
    if isinstance(w_hat, list) or isinstance(w_hat, np.ndarray) \
        or isinstance(w_hat, np.matrix):
        f = np.array
        w_hat = np.array(w_hat)
    elif isinstance(w_hat, sym.Matrix) or isinstance(w_hat, sym.ImmutableMatrix):
        f = sym.Matrix
    else:
        raise Exception(f"UnhatMatrix3: Unexpected type of w_hat: {type(w_hat)}")

    #matrix checking, for use in potential debug. generalized to both Sympy and Numpy
    same = np.array([w_hat + w_hat.T == f([
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]
    ) ] )

    #skew-symmetric checking removed because simplify() call in CalculateVb6() was slowing down
    #solve time; unsimplified Vb's resulted in non-skew-symmetric w_hat matrices

#     if (not same.all()):
#         raise Exception("UnhatMatrix3: w_hat not skew_symmetric")

    #NP and Sym
    return f([
        -w_hat[1,2],
        w_hat[0,2],
        -w_hat[0,1],
    ])

def InvSEn(SEnmat):
    '''Takes the inverse of a SE(n) matrix.
    Compatible with Numpy, Sympy, and list formats.
    '''
    if isinstance(SEnmat, list):
        SEnmat = np.matrix(SEnmat)
    ###
    n = SEnmat.shape[0]
```

```
    R = SEnmat[:(n-1), :(n-1)]
    p = SEnmat[:(n-1),   n-1 ]

    return SOnAndRnToSEn(R.T, -R.T @ p)

###

#test cases
```

In [4]:

```python
def InertiaMatrix6(m, scriptI):
    '''Takes the mass and inertia matrix properties of an object in space,
    and constructs a 6x6 matrix corresponding to [[mI 0]; [0 scriptI]].
    Currently only written for Sympy matrix representations.
    '''
    if (m.is_Matrix or not scriptI.is_square):
        raise Exception("Type error: m or scriptI in InertiaMatrix6")

    mat = sym.zeros(6)
    mI = m * sym.eye(3)
    mat[:3, :3] = mI
    mat[3:6, 3:6] = scriptI
    return mat

def HatVector6(vec):
    '''Convert a 6-dimensional body velocity into a 4x4 "hatted" matrix,
    [[w_hat v]; [0 0]], where w_hat is skew-symmetric.
    w =
    '''
    if isinstance(vec, np.matrix) or isinstance(vec, np.ndarray):
        vec = np.array(vec).flatten()

    v = vec[:3]
    w = vec[3:6]

    #this ensures if there are symbolic variables, they stay in Sympy form
    if isinstance(vec, sym.Matrix):
        v = sym.Matrix(v)
        w = sym.Matrix(w)

    w_hat = HatVector3(w)

    #note that the result isn't actually in SE(3) but
    #that the function below creates a 4x4 matrix from a 3x3 and
    #1x3 matrix - with type checking - so we'll use it
    mat = SOnAndRnToSEn(w_hat, v)
    return mat

def UnhatMatrix4(mat):
    '''Convert a 4x4 "hatted" matrix,[[w_hat v]; [0 0]], into a 6-dimensional
    body velocity [v, w].
    '''
    #same as above - matrices aren't SE(3) and SO(3) but the function
    #can take in a 4x4 mat and return a 3x3 and 3x1 mat
    [w_hat, v] = SEnToSOnAndRn(mat)
    w = UnhatMatrix3(w_hat)

    if (isinstance(w, np.matrix) or isinstance(w, np.ndarray)):
        return np.array([v, w]).flatten()
    elif isinstance(w, sym.Matrix):
        return sym.Matrix([v, w])
    else:
        raise Exception("Unexpected datatype in UnhatMatrix4")
```

In [120]:

```python
def CalculateVb6(G):
    '''Calculate the body velocity, a 6D vector [v, w], given a trans-
    formation matrix G from one frame to another.
    '''
    G_inv = InvSEn(G)
    Gdot = G.diff(t) #for sympy matrices, this also carries out chain rule
    V_hat = G_inv @ Gdot

#     if isinstance(G, sym.Matrix):
#         V_hat = sym.simplify(V_hat)

    return UnhatMatrix4(V_hat)

# test cases
```

In [81]:

```python
def compute_EL_lhs(lagrangian, q):
    '''
    Helper function for computing the Euler-Lagrange equations for a given system,
    so I don't have to keep writing it out over and over again.

    Inputs:
    - lagrangian: our Lagrangian function in symbolic (Sympy) form
    - q: our state vector [x1, x2, ...], in symbolic (Sympy) form

    Outputs:
    - eqn: the Euler-Lagrange equations in Sympy form
    '''

    # wrap system states into one vector (in SymPy would be Matrix)
    #q = sym.Matrix([x1, x2])
    qd = q.diff(t)
    qdd = qd.diff(t)

    # compute derivative wrt a vector, method 1
    # wrap the expression into a SymPy Matrix
    L_mat = sym.Matrix([lagrangian])
    dL_dq = L_mat.jacobian(q)
    dL_dqdot = L_mat.jacobian(qd)

    #set up the Euler-Lagrange equations
    #LHS = dL_dq - dL_dqdot.diff(t)
    LHS = dL_dqdot.diff(t) - dL_dq

    return LHS.T
```

In [116]:

```python
def rk4(dxdt, x, t, dt):
    '''
    Applies the Runge-Kutta method, 4th order, to a sample function,
    for a given state q0, for a given step size. Currently only
    configured for a 2-variable dependent system (x,y).
    ==========
    dxdt: a Sympy function that specifies the derivative of the system of interest
    t: the current timestep of the simulation
    x: current value of the state vector
    dt: the amount to increment by for Runge-Kutta
    ======
    returns:
    x_new: value of the state vector at the next timestep
    '''
    k1 = dt * dxdt(t, x)
    k2 = dt * dxdt(t + dt/2.0, x + k1/2.0)
    k3 = dt * dxdt(t + dt/2.0, x + k2/2.0)
    k4 = dt * dxdt(t + dt, x + k3)
    x_new = x + (k1 + 2.0*k2 + 2.0*k3 + k4)/6.0

    return x_new


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

```
    return xtraj
```

In [77]:
```
def format_solns(soln):
    eqns_solved = []
    #eqns_new = []

    for i, sol in enumerate(soln):
        for x in list(sol.keys()):
            eqn_solved = sym.Eq(x, sol[x])
            eqns_solved.append(eqn_solved)

    return eqns_solved
```

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 written work

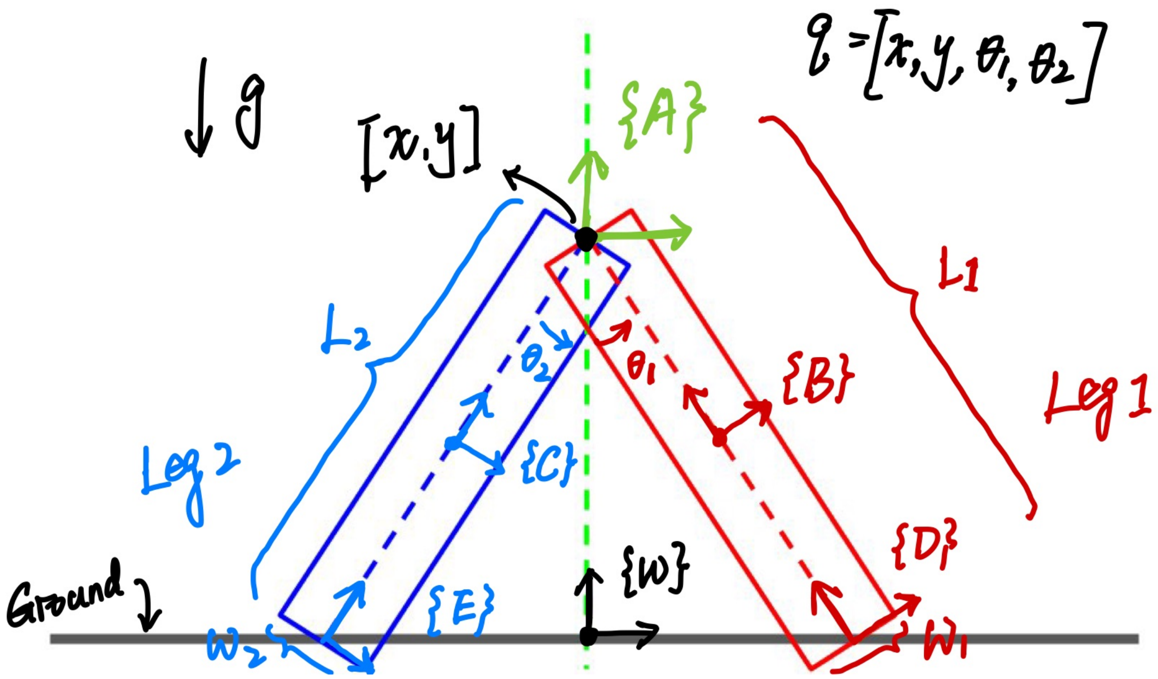
Problem 2 (20pts)

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.

See written work

In [9]:
```
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' height='350'></table>"))
```



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 (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 (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.

In [10]:

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

In [115]:

```
#define variables
L1, L2, m, J, W, g = sym.symbols(r'L_1, L_2, m, J, W, g')
t = sym.symbols(r't')
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])
qd = q.diff(t)
qdd = qd.diff(t)

#define transformation matrices. let A1 be in the direction of
#the right leg and A2 be in the direction of the left leg

#------right leg------#
Raa1 = sym.Matrix([
    [sym.cos(theta1), -sym.sin(theta1), 0],
    [sym.sin(theta1),  sym.cos(theta1), 0],
    [              0,                0, 1]
])

Rdf = sym.Matrix([
    [sym.cos(-theta1), -sym.sin(-theta1), 0],
    [sym.sin(-theta1),  sym.cos(-theta1), 0],
    [              0,                 0,   1]
])

p_a1b = sym.Matrix([0, -L1/2, 0])
p_bd =  sym.Matrix([0, -L1/2, 0])

Ga1 = SOnAndRnToSEn(Raa1, [0,0,0])
Ga1b = SOnAndRnToSEn(sym.eye(3), p_a1b)
Gbd  = SOnAndRnToSEn(sym.eye(3),  p_bd)
Gdf  = SOnAndRnToSEn(Rdf,  [0,0,0])

#------left leg------#

Raa2 = sym.Matrix([
    [sym.cos(theta2), -sym.sin(theta2), 0],
    [sym.sin(theta2),  sym.cos(theta2), 0],
    [              0,                0,   1]

])

Reg = sym.Matrix([
    [sym.cos(-theta2), -sym.sin(-theta2), 0],
    [sym.sin(-theta2),  sym.cos(-theta2), 0],
    [              0,                0, 1]
])

p_a2c = sym.Matrix([0, -L2/2, 0])
p_ce =  sym.Matrix([0, -L2/2, 0])

Gaa2 = SOnAndRnToSEn(Raa2, [0,0,0])
Ga2c = SOnAndRnToSEn(sym.eye(3), p_a2c)
Gce  = SOnAndRnToSEn(sym.eye(3), p_ce)
Geg  = SOnAndRnToSEn(Reg, [0,0,0])

#--------------------#
```

See attached written work for the additional frames A1, A2, F, and G that I defined.

```
#combine transformation matrices
Gsa = SOnAndRnToSEn(sym.eye(3),sym.Matrix([x,y,0]))
Gsb = Gsa @ Gaa1 @ Ga1b
Gsc = Gsa @ Gaa2 @ Ga2c
Gsd = Gsb @ Gbd
Gse = Gsc @ Gce
Gsf = Gsd @ Gdf
Gsg = Gse @ Geg

#define important positions in space
posn_CM2 = Gsc @ sym.Matrix([0,0,0,1]) #"bar" version of 3D posn
posn_CM1 = Gsb @ sym.Matrix([0,0,0,1])

bottom2 = Gsg @ sym.Matrix([0,0,0,1])
bottom1 = Gsf @ sym.Matrix([0,0,0,1])
```

In [114]:

```
#define lagrangian
ybottom1 = bottom1[1]
ybottom2 = bottom2[1]
y_CM1    = posn_CM1[1]
y_CM2    = posn_CM2[1]

#calculate KE of each object, then find Lagrangian
Vsb = CalculateVb6(Gsb)
Vsc = CalculateVb6(Gsc)
scriptI = J*sym.eye(3)
inertia_mat = InertiaMatrix6(m, scriptI)

U = m*g* (y_CM1 + y_CM2)
KE1 = 0.5 * (Vsb.T @ inertia_mat @ Vsb)[0]
KE2 = 0.5 * (Vsc.T @ inertia_mat @ Vsc)[0]
lagrangian = KE1 + KE2 - U
```

In [113]:

```
#removed all simplification in the variables used in calculations -
#this is just for show. simplify() calls slowed down my solve time to 1hr+
t0 = time.time()
lagrangian_disp = lagrangian.simplify()
print("\nLagrangian:")
display(lagrangian_disp)

tf = time.time()
print(f"Elapsed: {tf - t0} seconds")
```

```
Lagrangian:
```

$$0.5J\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5J\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.125L_1^2 m\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5L_1 gm\cos(\theta_1(t)) + 0.5L_1 m\sin(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}y(t) + 0.5L_1 m\cos(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}x(t) + 0.125L_2^2 m\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.5L_2 gm\cos(\theta_2(t)) + 0.5L_2 m\sin(\theta_2(t))\frac{d}{dt}\theta_2(t)\frac{d}{dt}y(t) + 0.5L_2 m\cos(\theta_2(t))\frac{d}{dt}\theta_2(t)\frac{d}{dt}x(t) - 2.0gmy(t) + m\left(\frac{d}{dt}x(t)\right)^2 + m\left(\frac{d}{dt}y(t)\right)^2$$

```
Elapsed: 8.89724612236023 seconds
```

In [108]:

```
#set up the left and right hand sides of Euler-Lagrange EQs
F1_v2, F2_v2 = sym.symbols(r'F_\theta_1, F_\theta_2')
F_mat = sym.Matrix([0, 0, F1_v2, F2_v2])
lamb1 = sym.symbols(r'\lambda_1')
lamb2 = sym.symbols(r'\lambda_2')

lamb_list = [lamb1, lamb2]
phi_list = [ybottom1, ybottom2]

print("Constraint equations (set equal to 0):")
for i, a in enumerate(phi_list):
    display(a)
```

Constraint equations (set equal to 0):

$$-L_1 \cos(\theta_1(t)) + y(t)$$

$$-L_2 \cos(\theta_2(t)) + y(t)$$

In [112]:

```python
#solve for the Euler-Lagrange equations. previously was in a function; pulled it out so it would be easier
#to isolate where the slowdown is
qd = q.diff(t)
qdd = qd.diff(t)

lhs = compute_EL_lhs(lagrangian, q)
expr_matrix = lhs
RHS = sym.zeros(len(expr_matrix), 1)
RHS = RHS + F_mat

phidd_matrix = sym.Matrix([0])
phidd_list = []

#add constraint terms to matrix of expressions to solve
for i in range(len(phi_list)):
    phi = phi_list[i]
    lamb = lamb_list[i]

    phidd = phi.diff(t).diff(t)
    lamb_grad = sym.Matrix([lamb * phi.diff(a) for a in q])
    qdd = qdd.row_insert(len(qdd), sym.Matrix([lamb]) )

    #format equations so they're all in one matrix
    RHS = RHS + lamb_grad
    phidd_list.append(sym.Matrix([phidd])  )

RHS = RHS.row_insert(len(qdd)-1, sym.Matrix([0]))
RHS = RHS.row_insert(len(qdd), sym.Matrix([0]))

for phidd_matrix in phidd_list:
    expr_matrix = expr_matrix.row_insert(len(expr_matrix), phidd_matrix)

#do symbolic substitutions before solving to speed up computation
subs_dict = {
    L1: 1,
    L2: 1,
    m:  1,
    g:  9.8,
    J:  1,
    k1: 20
}
expr_matrix = expr_matrix.subs(subs_dict)
RHS = RHS.subs(subs_dict)
total_eq = sym.Eq(expr_matrix, RHS)
```

In [92]:

```python
t0 = time.time()
total_eq_simplify = total_eq.simplify()
tf = time.time()
print(f"Elapsed: {tf - t0} seconds")
```

```
Elapsed: 7.705913066864014 seconds
```

In [111]:
```
print("Equations to solve (LHS = lambda*grad(phi) + F_ext):")
display(total_eq_simplify)
print("Variables to solve for (transposed):")
display(qdd.T)
```

Equations to solve (LHS = lambda*grad(phi) + F_ext):

$$\left[\begin{matrix}0\\ \lambda_1+\lambda_2\\ F_{\theta 1}+\lambda_1\sin(\theta_1(t))\\ F_{\theta 2}+\lambda_2\sin(\theta_2(t))\\ 0\\ 0\end{matrix}\right]=\left[\begin{matrix}-0.5\sin(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2-0.5\sin(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2+0.5\cos(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t)+0.5\cos(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t)+2.0\frac{d^2}{dt^2}x(t)\\ 0.5\sin(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t)+0.5\sin(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t)+0.5\cos(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2+0.5\cos(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2+2.0\frac{d^2}{dt^2}y(t)+19.6\\ 0.5\sin(\theta_1(t))\frac{d^2}{dt^2}y(t)+4.9\sin(\theta_1(t))+0.5\cos(\theta_1(t))\frac{d^2}{dt^2}x(t)+1.25\frac{d^2}{dt^2}\theta_1(t)\\ 0.5\sin(\theta_2(t))\frac{d^2}{dt^2}y(t)+4.9\sin(\theta_2(t))+0.5\cos(\theta_2(t))\frac{d^2}{dt^2}x(t)+1.25\frac{d^2}{dt^2}\theta_2(t)\\ \sin(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t)+\cos(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2+\frac{d^2}{dt^2}y(t)\\ \sin(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t)+\cos(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2+\frac{d^2}{dt^2}y(t)\end{matrix}\right]$$

Variables to solve for (transposed):

$$\left[\begin{matrix}\frac{d^2}{dt^2}x(t) & \frac{d^2}{dt^2}y(t) & \frac{d^2}{dt^2}\theta_1(t) & \frac{d^2}{dt^2}\theta_2(t) & \lambda_1 & \lambda_2\end{matrix}\right]$$

In [94]:
```
t0 = time.time()

soln = sym.solve(total_eq_simplify, qdd, dict = True, simplify = False)
#soln = sym.solve((total_eq_simplify.lhs - total_eq_simplify.rhs).tolist(), qdd, \
#     dict = True, simplify = False, manual = True, quick = True, particular = True)

print("Solve_EL: solved.")
tf = time.time()
print(f"Elapsed: {tf - t0} seconds")
```

Solve_EL: solved.
Elapsed: 103.22751140594482 seconds

In [95]:
```
eqns_solved = format_solns(soln)
```

In [96]:
```python
#simplify the solutions for readability and evaluability
eqns_new = []
t0 = time.time()
i = 0

for eq in tqdm.tqdm(eqns_solved):
    i += 1
    if i > 4:
        continue
    eq_new = eq.simplify()
    eqns_new.append(eq_new)
    print(f"Iteration {i} finished at time {round(time.time() - t0, 2)} seconds")
```

```
 17%|██████████████                                                                      | 1/6 [00:55<04:37, 55.42s/it]

Iteration 1 finished at time 55.43 seconds

 33%|████████████████████████████                                                        | 2/6 [02:21<04:52, 73.19s/it]

Iteration 2 finished at time 141.05 seconds

 50%|██████████████████████████████████████████                                          | 3/6 [04:26<04:50, 96.88s/it]

Iteration 3 finished at time 266.13 seconds

100%|████████████████████████████████████████████████████████████████████████████████████| 6/6 [06:36<00:00, 66.08s/it]

Iteration 4 finished at time 396.5 seconds
Iteration 5 finished at time 396.5 seconds
Iteration 6 finished at time 396.5 seconds
```

In [103]:
```python
# for eq in eqns_new:
#     display(eq)
```

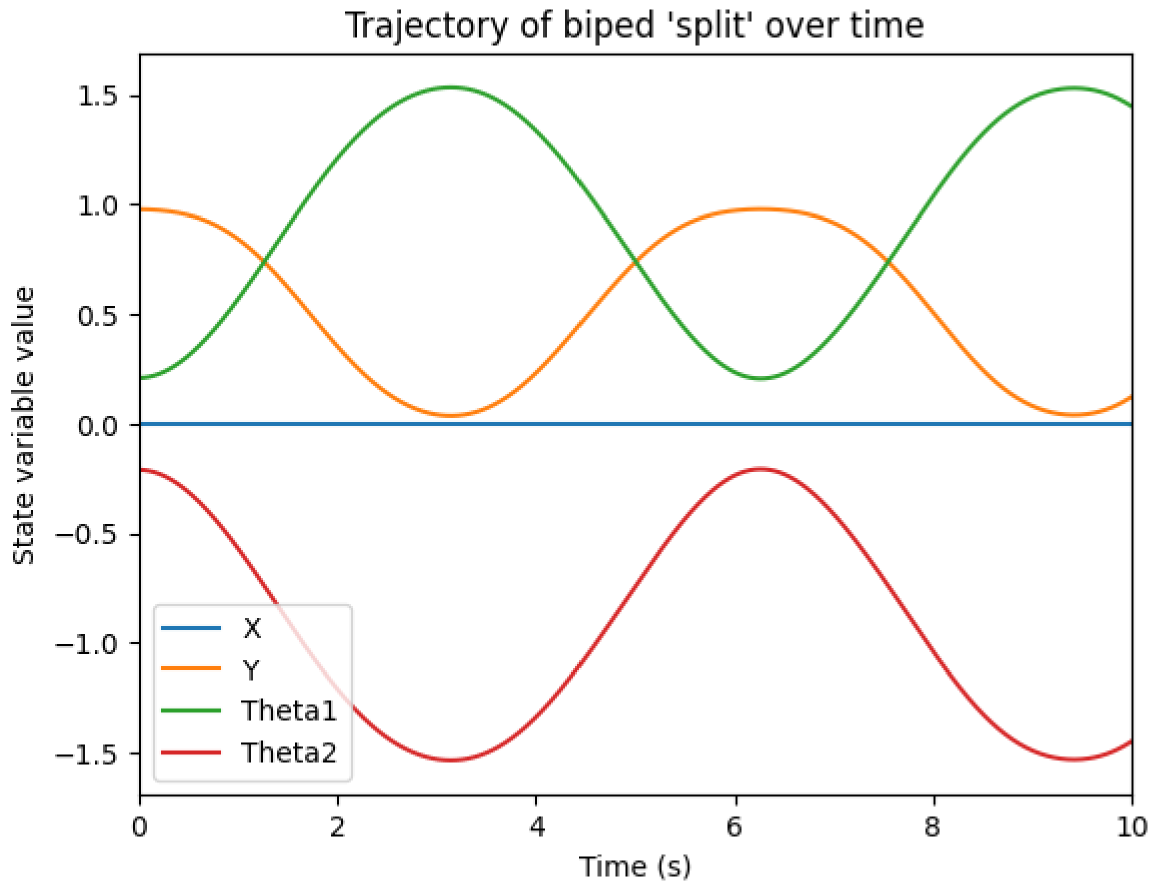
In [102]:
```python
#take results and turn them into numerical functions
xdd_sy      = eqns_new[0].rhs
ydd_sy      = eqns_new[1].rhs
theta1dd_sy = eqns_new[2].rhs
theta2dd_sy = eqns_new[3].rhs

xd      = x.diff(t)
yd      = y.diff(t)
theta1d = theta1.diff(t)
theta2d = theta2.diff(t)

q_ext = sym.Matrix([x, y, theta1, theta2, xd, yd, theta1d, theta2d, F1_v2, F2_v2])

xdd_np = sym.lambdify(q_ext, xdd_sy)
ydd_np = sym.lambdify(q_ext, ydd_sy)
theta1dd_np = sym.lambdify(q_ext, theta1dd_sy)
theta2dd_np = sym.lambdify(q_ext, theta2dd_sy)

#help(xdd_np)
```

In [118]:

```
#define simulation functions
def dxdt(t, s):
    #dependence on t matters for this problem
    k1 = 20
    theta1 = s[2]
    theta2 = s[3]
    F1 = -k1 * (theta1 - (  np.pi/15  + np.pi/3 * (np.sin(t/2))**2  ))
    F2 = -k1 * (theta2 - ( -np.pi/15  - np.pi/3 * (np.sin(t/2))**2  ))

    s_ext = np.append(s, [F1, F2])
    return np.array([s[4], s[5], s[6], s[7],
        xdd_np(*s_ext), ydd_np(*s_ext), theta1dd_np(*s_ext), theta2dd_np(*s_ext)])

t_span = [0, 10]
dt = 0.01
th0 = np.pi/15
ICs = [0, np.cos(th0), th0, -th0, 0, 0, 0, 0]

q_array = simulate(dxdt, ICs, t_span, dt, rk4)
```

In [100]:

```
plt.figure(1)
t_array = np.linspace(t_span[0], t_span[1], len(q_array[0]))

plt.plot(t_array, q_array[0], label="X")
plt.plot(t_array, q_array[1], label="Y")
plt.plot(t_array, q_array[2], label="Theta1")
plt.plot(t_array, q_array[3], label="Theta2")
#plt.ylim(-4,4)
plt.xlim(0,10)

plt.ylabel("State variable value")
plt.xlabel("Time (s)")
plt.title("Trajectory of biped 'split' over time")

plt.legend()
plt.show()
```



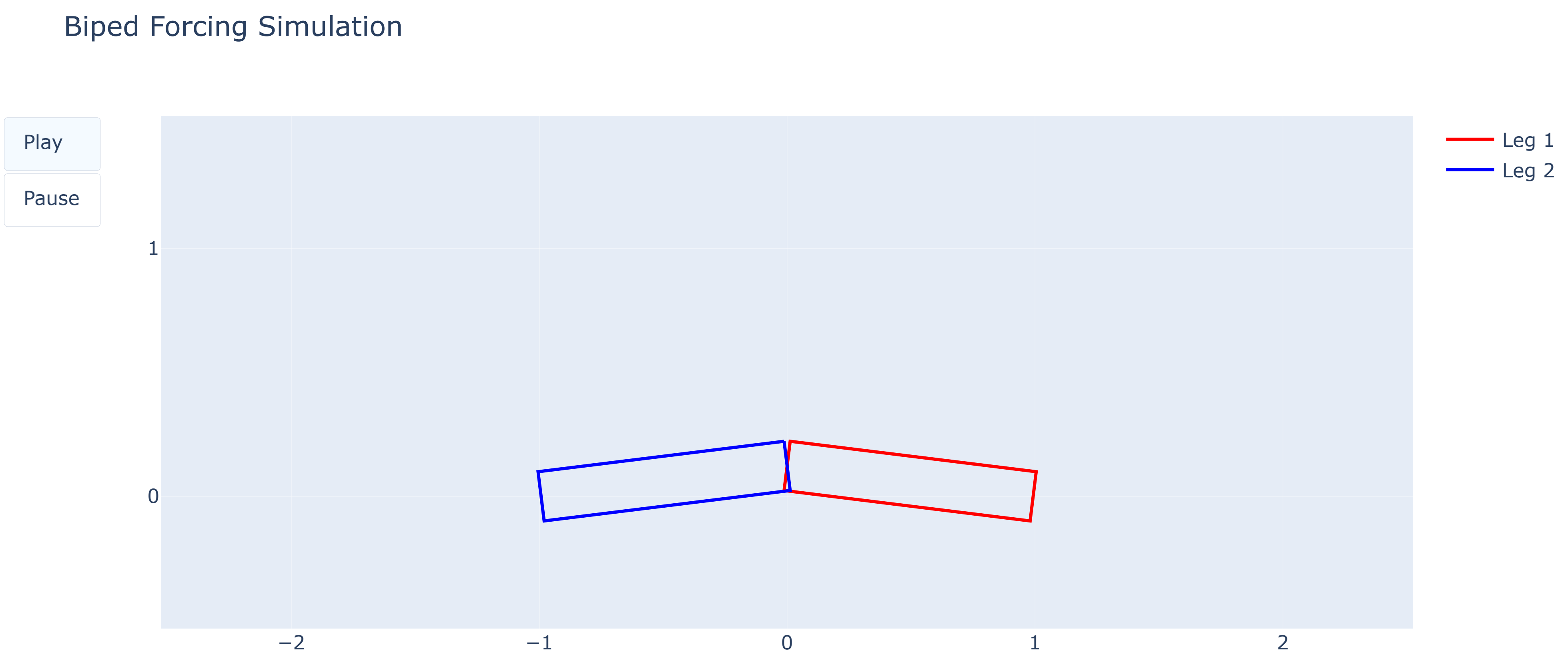
In [72]:
```python
def animate_biped(q_array, L1=1, L2=1, w=0.2, T=10):
    """
    Function to generate web-based animation of biped with two legs.

    Parameters:
    ================================================
    q_array:
        trajectory of x, y, theta1, theta2
    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 pendulum angle trajectories.
    x_array      = q_array[0]
    y_array      = q_array[1]
    theta1_array = q_array[2]
    theta2_array = q_array[3]

    #this matrix contains vectors that correspond to the locations
    #of all 4 vertices of the leg rectangles in space. needs to be
    #multiplied by the transf. mat. for the top of each leg to get world posns.
    vertices_mat_L1 = np.matrix([
        [-w/2, w/2, w/2, -w/2],
        [   0,   0, -L1,  -L1],
        [   0,   0,   0,    0],
        [   1,   1,   1,    1]
    ])

    vertices_mat_L2 = np.matrix([
```

```python
    [-w/2, w/2, w/2, -w/2],
    [   0,   0, -L2,  -L2],
    [   0,   0,   0,    0],
    [   1,   1,   1,    1]
])

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

###############################################
# Define arrays containing data for plotting
vertices1x = np.zeros((4,N))
vertices1y = np.zeros((4,N))
vertices2x = np.zeros((4,N))
vertices2y = np.zeros((4,N))

# evaluate homogeneous transformations to get data to plot
for i in range(N): # iteration through each time step

    #transformation matrices we need: Tsa, Tab, Tac, Tbd, Tce,
    x      = x_array[i]
    y      = y_array[i]
    theta1 = theta1_array[i]
    theta2 = theta2_array[i]

    Raa1 = np.matrix([
        [np.cos(theta1), -np.sin(theta1), 0],
        [np.sin(theta1),  np.cos(theta1), 0],
        [              0,               0, 1]
    ])

    Raa2 = np.matrix([
        [np.cos(theta2), -np.sin(theta2), 0],
        [np.sin(theta2),  np.cos(theta2), 0],
        [              0,               0,   1]

    ])

    Gaa1 = SOnAndRnToSEn(Raa1, [0,0,0])
    Gaa2 = SOnAndRnToSEn(Raa2, [0,0,0])

    #---------------------#

    #combine transformation matrices
    Gsa = SOnAndRnToSEn(np.matrix(np.eye(3)),[x,y,0])
    vertices1 = Gsa @ Gaa1 @ vertices_mat_L1
    vertices2 = Gsa @ Gaa2 @ vertices_mat_L2

    vertices1x[:,i] = vertices1[0,:]
    vertices1y[:,i] = vertices1[1,:]
    vertices2x[:,i] = vertices2[0,:]
    vertices2y[:,i] = vertices2[1,:]

#all the stuff below here is for plotting

####################################
# Using these to specify axis limits.
xm = np.min(x_array)-L1
xM = np.max(x_array)+L1
ym = np.min(y_array)- 0.5*L1
yM = np.max(y_array)+ 0.5*L1
```

```python
##########################
# Defining data dictionary.
# Trajectories are here.
data=[
    dict(name='Leg 1'),
    dict(name='Leg 2'),
    ]

###############################
# Preparing simulation layout.
# Title and axis ranges are here.
layout=dict(autosize=False, width=1000, height=500,
            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='Biped Forcing 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=[vertices1x[0,k], vertices1x[1,k], vertices1x[2,k], vertices1x[3,k], vertices1x[0,k] ],
                        y=[vertices1y[0,k], vertices1y[1,k], vertices1y[2,k], vertices1y[3,k], vertices1y[0,k] ],
                        mode='lines',
                        line=dict(color='red', width=2),
                        ), #right leg

                   dict(x=[vertices2x[0,k], vertices2x[1,k], vertices2x[2,k], vertices2x[3,k], vertices2x[0,k] ],
                        y=[vertices2y[0,k], vertices2y[1,k], vertices2y[2,k], vertices2y[3,k], vertices2y[0,k] ],
                        mode='lines',
                        line=dict(color='blue', width=2),
                        ), #left leg

                  ]) for k in range(N)]

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

In [101]:

```
animate_biped(q_array, L1=1, L2=1, w=0.2, T=10)
```



In [ ]: