hw7-FA2024

November 30, 2024

1 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.

[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
#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)
```

1.1 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.

To show that A is skew symmetric, we need to show that $A^T = -A$.

We have $A = \frac{d}{dt}(R)R^{-1}$.

Taking the transpose of A gives us $A^T = \left(\frac{d}{dt}(R)R^{-1}\right)^T$.

We know that $(AB)^T = B^T A^T$ for any matrices A and B.

Therefore, $A^T = \left(\frac{d}{dt}(R)R^{-1}\right)^T = \left(R^{-1}\right)^T \left(\frac{d}{dt}(R)\right)^T$.

Since $R \in SO(n)$, we have $R^T R = I$.

Taking the derivative of both sides gives us $\frac{d}{dt}(R^T R) = 0$.

Therefore, $\frac{d}{dt}(R^T)R + R^T \frac{d}{dt}(R) = 0$.

Multiplying both sides by R^{-1} on the right gives us $\frac{d}{dt}(R^T)RR^{-1} + R^T \frac{d}{dt}(R)R^{-1} = 0$.

Therefore, $\frac{d}{dt}(R^T)R^{-1} + R^T \frac{d}{dt}(R)R^{-1} = 0$.

Therefore, $\left(\frac{d}{dt}(R)R^{-1}\right)^T = -\frac{d}{dt}(R)R^{-1}$.

This gives that , $A^T = -A$. Therefore, A is skew symmetric.

1.2 Problem 2 (20pts)

Show that

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

where ω and r_b are both 3×1 vectors.

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

The left hand side of the equation $\hat{\omega}\, r_b = \omega \times r_b$.

The right hand side of the equation $-\hat{r}_b\, \omega = -(r_b \times \omega)$.

Cross product is anti-commutative, i.e., $a \times b = -b \times a$.

Therefore, $r_b \times \omega = -(\omega \times r_b)$.

Subbing this in

$-\hat{r}_b\, \omega = -(-(\omega \times r_b)) = \omega \times r_b$.

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

Summarizing this we show that the left hand side of the equation is equal to the right hand side of the equation.

[55]:
```
from IPython.core.display import HTML, Markdown
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/
  ↪raw/master/biped_simplified.jpg' width=600' height='350'></table>"))
```

```
<IPython.core.display.HTML object>
```

1.3 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.

[102]:
```
## Your code goes here
import numpy as np
```

[108]:
```
def integrate(f,xt,t,dt):
    """
    This function takes in an initial condition x(t) and a timestep dt,
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x(t). It outputs a vector x(t+dt) at the future
    time step.

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

```
        step size for integration

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

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

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

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

```
    for i in range(N):
        xtraj[:,i]=integrate(f,x,tvec[i],dt)
        x = np.copy(xtraj[:,i])
    return xtraj
```

```
<>:2: SyntaxWarning: invalid escape sequence '\d'
<>:31: SyntaxWarning: invalid escape sequence '\d'
<>:2: SyntaxWarning: invalid escape sequence '\d'
<>:31: SyntaxWarning: invalid escape sequence '\d'
/tmp/ipykernel_125351/24789098.py:2: SyntaxWarning: invalid escape sequence '\d'
  """
/tmp/ipykernel_125351/24789098.py:31: SyntaxWarning: invalid escape sequence
'\d'
  """
```

[84]:
```
# Define symbols
m1, m2, g, L1, L2, J1, J2, t, k = sym.symbols('m1 m2 g L1 L2 J1 J2 t k')
lambda_1, lambda_2, phi_1, phi_2 = sym.symbols('lambda_1 lambda_2 phi_1 phi_2')
theta1, theta2 = sym.Function('theta1')(t), sym.Function('theta2')(t)
x, y = sym.Function('x')(t), sym.Function('y')(t)
q = sym.Matrix([x, y, theta1, theta2])
qd = q.diff(t)
qdd = qd.diff(t)
```

[58]:
```
# Define helper for generating SE(3) matrix

# rotational
def Rotz(theta):
    return sym.Matrix([
        # 4x4 rotation matrix
        [sym.cos(theta), -sym.sin(theta), 0, 0],
        [sym.sin(theta), sym.cos(theta), 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

# translational se(3)
def T(x, y):
    return sym.Matrix([
        # 4x4 translation matrix
        [1, 0, 0, x],
        [0, 1, 0, y],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])
```

[59]:
```
# Define the helper for hatting a 3x1 vector
def hat(v: sym.Matrix) -> sym.Matrix:
    return sym.Matrix([[0, -v[2], v[1]],
                       [v[2], 0, -v[0]],
                       [-v[1], v[0], 0]])
```

[60]:
```
# Define the helper for unhating a 3x3 skew-symmetric matrix
def unhat(v: sym.Matrix) -> sym.Matrix:
    return sym.Matrix([v[2, 1], v[0, 2], v[1, 0]])
```

[61]:
```
# Define the helper for getting the inverse of a g matrix
def inverse_g(g: sym.Matrix) -> sym.Matrix:
    R = g[:3, :3]
    p = g[:3, 3]
    return sym.Matrix([[R.T, -R.T * p], [0, 0, 0, 1]])
```

[62]:
```
# use q to define all the g matrices
g_wa = T(q[0], q[1])
g_ab = Rotz(q[2]) * T(0, -L1 / 2)
g_ac = Rotz(q[3]) * T(0, -L2 / 2)
g_bd = T(0, -L1 / 2)
g_ce = T(0, -L2 / 2)
```

[63]:
```
display(g_wa, g_ab, g_ac, g_bd, g_ce)
```

$$\begin{bmatrix} 1 & 0 & 0 & x(t) \\ 0 & 1 & 0 & y(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(\theta_1(t)) & -\sin(\theta_1(t)) & 0 & \frac{L_1 \sin(\theta_1(t))}{2} \\ \sin(\theta_1(t)) & \cos(\theta_1(t)) & 0 & -\frac{L_1 \cos(\theta_1(t))}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(\theta_2(t)) & -\sin(\theta_2(t)) & 0 & \frac{L_2 \sin(\theta_2(t))}{2} \\ \sin(\theta_2(t)) & \cos(\theta_2(t)) & 0 & -\frac{L_2 \cos(\theta_2(t))}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{L_1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{L_2}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[64]:
```
origin = sym.Matrix([0, 0, 0, 1])
r_b = g_wa * g_ab * origin
r_c = g_wa * g_ac * origin
r_d = g_wa * g_ab * g_bd * origin
r_e = g_wa * g_ac * g_ce * origin

display(r_b, r_c, r_d, r_e)
```

$$\begin{bmatrix} \frac{L_1 \sin(\theta_1(t))}{2} + x(t) \\ -\frac{L_1 \cos(\theta_1(t))}{2} + y(t) \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{L_2 \sin(\theta_2(t))}{2} + x(t) \\ -\frac{L_2 \cos(\theta_2(t))}{2} + y(t) \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} L_1 \sin(\theta_1(t)) + x(t) \\ -L_1 \cos(\theta_1(t)) + y(t) \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} L_2 \sin(\theta_2(t)) + x(t) \\ -L_2 \cos(\theta_2(t)) + y(t) \\ 0 \\ 1 \end{bmatrix}$$

[65]:
```
def g_dot_SE3(transform, angular_velocity):
    """
    Computes the time derivative of an SE(3) transformation matrix.

    The SE(3) matrix `transform` is composed of a rotation matrix `R` and a position vector `p`.
    The derivative is given as:
        g_dot = [R_dot   p_dot]
                [  0       0 1],

    where:
        - R_dot = R * hat(w),
          `w` is the angular velocity vector.
        - p_dot = derivative of the position vector `p`.

    Parameters
    ----------
    transform : sympy.Matrix
        The SE(3) homogeneous transformation matrix (3x3 in planar case).
    angular_velocity : sympy.Symbol
```

```
        Angular velocity (derivative of the rotation angle, theta).

    Returns
    -------
    sympy.Matrix
        Time derivative of the SE(3) homogeneous transformation matrix (g_dot).

    Notes
    -----
    This assumes a 2D planar system with rotation about the z-axis (right-hand␣
↪rule). The system uses:
        - A 3x3 matrix for rotation and translation.
        - Angular velocity vector w = [0, 0, d(theta)/dt], where z is the axis␣
↪of rotation.
    """
    # Compute the angular velocity vector (in matrix form for SE(3))
    w = sym.Matrix([0, 0, angular_velocity.diff(t)])

    # Extract Components
    R = transform[:3, :3]  # Rotation matrix
    p = transform[:3, 3]   # Translation vector

    # Derivative of rotation: R_dot = R * hat(w)
    R_dot = R * hat(w)

    # Derivative of translation: p_dot
    p_dot = p.diff(t)

    # Construct g_dot
    # use T and Rotz to construct the g_dot matrix
    g_dot_matrix = sym.Matrix([[R_dot, p_dot], [0, 0, 0, 1]])
    return g_dot_matrix
```

[66]:
```
def body_velocity(transform, angular_velocity):
    """
    Computes the body velocity from the SE(3) transformation matrix.

    The body velocity is given as a 6D spatial velocity vector (in planar␣
↪motion, reduced to 3D):
        V_body = [v  ] -> Linear velocity from R_inv * (p_dot)
                 [w  ] -> Angular velocity

    Parameters
    ----------
    transform : sympy.Matrix
        The SE(3) homogeneous transformation matrix for a frame (3x3 in planar␣
↪case).
```

```
    angular_velocity : sympy.Symbol
        Angular velocity (derivative of the rotation angle, theta).

    Returns
    -------
    sympy.Matrix
        The spatial body velocity vector for the specified frame.

    Raises
    ------
    ValueError
        If an invalid output type is specified.

    Notes
    -----
    - Body velocity is computed as:
          V_body = g^(-1) * g_dot
      where g^(-1) is the inverse of the SE(3) matrix.
    - This function is planar, so angular velocity is scalar (about z-axis).
    """
    # Compute g^-1 (the inverse of SE(3))
    transform_inverse = inverse_g(transform)

    # Compute g_dot
    g_dot_matrix = g_dot_SE3(transform, angular_velocity)

    # Multiply to get body velocity: V_body = g^(-1) * g_dot
    V_body_matrix = transform_inverse * g_dot_matrix

    # Extract body velocity: w = angular component, v = linear component
    w_hat = V_body_matrix[:3, :3]
    v = V_body_matrix[:3, 3]
    w = unhat(w_hat)

    return v.col_join(w)
```

[67]:
```
# Define the angular velocities
body_velocity_b = sym.simplify(body_velocity(g_wa * g_ab, q[2]))
body_velocity_c = sym.simplify(body_velocity(g_wa * g_ac, q[3]))
```

[68]:
```
display(body_velocity_b, body_velocity_c)
```

$$\begin{bmatrix} \frac{L_1 \frac{d}{dt}\theta_1(t)}{2} - x(t)\cos(\theta_1(t)) - y(t)\sin(\theta_1(t)) + \sin(\theta_1(t))\frac{d}{dt}y(t) + \cos(\theta_1(t))\frac{d}{dt}x(t) \\ \frac{L_1}{2} + x(t)\sin(\theta_1(t)) - y(t)\cos(\theta_1(t)) - \sin(\theta_1(t))\frac{d}{dt}x(t) + \cos(\theta_1(t))\frac{d}{dt}y(t) \\ 0 \\ 0 \\ 0 \\ \frac{d}{dt}\theta_1(t) \end{bmatrix}$$

$$\begin{bmatrix} \frac{L_2 \frac{d}{dt}\theta_2(t)}{2} - x(t)\cos(\theta_2(t)) - y(t)\sin(\theta_2(t)) + \sin(\theta_2(t))\frac{d}{dt}y(t) + \cos(\theta_2(t))\frac{d}{dt}x(t) \\ \frac{L_2}{2} + x(t)\sin(\theta_2(t)) - y(t)\cos(\theta_2(t)) - \sin(\theta_2(t))\frac{d}{dt}x(t) + \cos(\theta_2(t))\frac{d}{dt}y(t) \\ 0 \\ 0 \\ 0 \\ \frac{d}{dt}\theta_2(t) \end{bmatrix}$$

[70]:
```
# Define inertia matrices
I1 = sym.Matrix([
    [m1, 0, 0, 0, 0, 0],
    [0, m1, 0, 0, 0, 0],
    [0, 0, m1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, J1]
])

I2 = sym.Matrix([
    [m2, 0, 0, 0, 0, 0],
    [0, m2, 0, 0, 0, 0],
    [0, 0, m2, 0, 0, 0],
    [0, 0, 0, 2, 0, 0],
    [0, 0, 0, 0, 2, 0],
    [0, 0, 0, 0, 0, J2]
])

KE = 0.5 * body_velocity_b.T * I1 * body_velocity_b + 0.5 * body_velocity_c.T *␣
↪I2 * body_velocity_c
KE = sym.simplify(KE[0])

KE_sym = sym.symbols('KE')
display(sym.Eq(KE_sym, KE))
```

$$KE = 0.5J_1\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5J_2\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.125L_1^2m_1\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.125L_1^2m_1 + 0.5L_1m_1x(t)\sin(\theta_1(t)) - 0.5L_1m_1x(t)\cos(\theta_1(t))\frac{d}{dt}\theta_1(t) - 0.5L_1m_1y(t)\sin(\theta_1(t))\frac{d}{dt}\theta_1(t) - 0.5L_1m_1y(t)\cos(\theta_1(t)) + 0.5L_1m_1\sin(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}y(t) - 0.5L_1m_1\sin(\theta_1(t))\frac{d}{dt}x(t) + 0.5L_1m_1\cos(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}x(t) + 0.5L_1m_1\cos(\theta_1(t))\frac{d}{dt}y(t) + 0.125L_2^2m_2\left(\frac{d}{dt}\theta_2(t)\right)^2 +$$

$$0.125L_2^2m_2 + 0.5L_2m_2x(t)\sin(\theta_2(t)) - 0.5L_2m_2x(t)\cos(\theta_2(t))\frac{d}{dt}\theta_2(t) - 0.5L_2m_2y(t)\sin(\theta_2(t))\frac{d}{dt}\theta_2(t) - 0.5L_2m_2y(t)\cos(\theta_2(t)) + 0.5L_2m_2\sin(\theta_2(t))\frac{d}{dt}\theta_2(t)\frac{d}{dt}y(t) - 0.5L_2m_2\sin(\theta_2(t))\frac{d}{dt}x(t) + 0.5L_2m_2\cos(\theta_2(t))\frac{d}{dt}\theta_2(t)\frac{d}{dt}x(t) + 0.5L_2m_2\cos(\theta_2(t))\frac{d}{dt}y(t) + 0.5m_1x^2(t) - 1.0m_1x(t)\frac{d}{dt}x(t) + 0.5m_1y^2(t) - 1.0m_1y(t)\frac{d}{dt}y(t) + 0.5m_1\left(\frac{d}{dt}x(t)\right)^2 + 0.5m_1\left(\frac{d}{dt}y(t)\right)^2 + 0.5m_2x^2(t) - 1.0m_2x(t)\frac{d}{dt}x(t) + 0.5m_2y^2(t) - 1.0m_2y(t)\frac{d}{dt}y(t) + 0.5m_2\left(\frac{d}{dt}x(t)\right)^2 + 0.5m_2\left(\frac{d}{dt}y(t)\right)^2$$

[71]:
```
PE = m1 * g * r_b[1] + m2 * g * r_c[1]
PE_sym = sym.symbols('PE')
display(sym.Eq(PE_sym, PE))
```

$$PE = gm_1\left(-\frac{L_1\cos(\theta_1(t))}{2} + y(t)\right) + gm_2\left(-\frac{L_2\cos(\theta_2(t))}{2} + y(t)\right)$$

[74]:
```
L_sym = sym.symbols('L')
L = KE - PE
display(sym.Eq(L_sym, L))
```

$$L = 0.5J_1\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5J_2\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.125L_1^2m_1\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.125L_1^2m_1 + 0.5L_1m_1x(t)\sin(\theta_1(t)) - 0.5L_1m_1x(t)\cos(\theta_1(t))\frac{d}{dt}\theta_1(t) - 0.5L_1m_1y(t)\sin(\theta_1(t))\frac{d}{dt}\theta_1(t) - 0.5L_1m_1y(t)\cos(\theta_1(t)) + 0.5L_1m_1\sin(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}y(t) - 0.5L_1m_1\sin(\theta_1(t))\frac{d}{dt}x(t) + 0.5L_1m_1\cos(\theta_1(t))\frac{d}{dt}\theta_1(t)\frac{d}{dt}x(t) + 0.5L_1m_1\cos(\theta_1(t))\frac{d}{dt}y(t) + 0.125L_2^2m_2\left(\frac{d}{dt}\theta_2(t)\right)^2 + 0.125L_2^2m_2 + 0.5L_2m_2x(t)\sin(\theta_2(t)) - 0.5L_2m_2x(t)\cos(\theta_2(t))\frac{d}{dt}\theta_2(t) - 0.5L_2m_2y(t)\sin(\theta_2(t))\frac{d}{dt}\theta_2(t) - 0.5L_2m_2y(t)\cos(\theta_2(t)) + 0.5L_2m_2\sin(\theta_2(t))\frac{d}{dt}\theta_2(t)\frac{d}{dt}y(t) - 0.5L_2m_2\sin(\theta_2(t))\frac{d}{dt}x(t) + 0.5L_2m_2\cos(\theta_2(t))\frac{d}{dt}\theta_2(t)\frac{d}{dt}x(t) + 0.5L_2m_2\cos(\theta_2(t))\frac{d}{dt}y(t) - gm_1\left(-\frac{L_1\cos(\theta_1(t))}{2} + y(t)\right) - gm_2\left(-\frac{L_2\cos(\theta_2(t))}{2} + y(t)\right) + 0.5m_1x^2(t) - 1.0m_1x(t)\frac{d}{dt}x(t) + 0.5m_1y^2(t) - 1.0m_1y(t)\frac{d}{dt}y(t) + 0.5m_1\left(\frac{d}{dt}x(t)\right)^2 + 0.5m_1\left(\frac{d}{dt}y(t)\right)^2 + 0.5m_2x^2(t) - 1.0m_2x(t)\frac{d}{dt}x(t) + 0.5m_2y^2(t) - 1.0m_2y(t)\frac{d}{dt}y(t) + 0.5m_2\left(\frac{d}{dt}x(t)\right)^2 + 0.5m_2\left(\frac{d}{dt}y(t)\right)^2$$

[89]:
```
# Calculate E-L equations
dL_dq = L.diff(q)
dL_dq_dot = L.diff(qd)

EL_eq = dL_dq - dL_dq_dot.diff(t)
```

```
EL_eq = EL_eq.col_join(sym.zeros(2, 1))
display(EL_eq)
```

$$\begin{bmatrix} 0.5L_1m_1\sin\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5L_1m_1\sin\left(\theta_1(t)\right) - 0.5L_1m_1\cos\left(\theta_1(t)\right)\frac{d^2}{dt^2}\theta_1(t) + 0.5L_2m_2\sin\left(\theta_2(t)\right)\left(\frac{d}{dt} \\ -0.5L_1m_1\sin\left(\theta_1(t)\right)\frac{d^2}{dt^2}\theta_1(t) - 0.5L_1m_1\cos\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 0.5L_1m_1\cos\left(\theta_1(t)\right) - 0.5L_2m_2\sin\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t) - \\ -1.0J_1\frac{d^2}{dt^2}\theta_1(t) - 0.25L_1^2m_1\frac{d^2}{dt^2}\theta_1(t) - \frac{L_1gm_1\sin\left(\theta_1(t)\right)}{2} + 0.5L_1m_1x(t)\cos\left(\theta_1(\\ -1.0J_2\frac{d^2}{dt^2}\theta_2(t) - 0.25L_2^2m_2\frac{d^2}{dt^2}\theta_2(t) - \frac{L_2gm_2\sin\left(\theta_2(t)\right)}{2} + 0.5L_2m_2x(t)\cos\left(\theta_2(\\ \end{bmatrix}$$

[ ]:
```
phi_1, phi_2 = r_d[1], r_e[1]

# calculate dphi dq
dphi_dq_1, dphi_dq_2 = phi_1.diff(q), phi_2.diff(q)
ddphi_dt_1, ddphi_dt_2 = phi_1.diff(t, 2), phi_2.diff(t, 2)

k = 20
theta1_torque = np.pi / 15 + ((np.pi / 3) * sym.sin(t / 2)) ** 2
theta2_torque = -np.pi / 15 - ((np.pi / 3) * sym.sin(t / 2)) ** 2
torques = sym.Matrix([
    0, 0, k * (q[2] - theta1_torque), k * (q[3] - theta2_torque)
])
display(torques)
```

$$\begin{bmatrix} 0 \\ 0 \\ k\left(\theta_1(t) - 1.09662271123215\sin^2\left(\frac{t}{2}\right) - 0.20943951023932\right) \\ k\left(\theta_2(t) + 1.09662271123215\sin^2\left(\frac{t}{2}\right) + 0.20943951023932\right) \end{bmatrix}$$

[85]:
```
rhs = lambda_1 * dphi_dq_1 + lambda_2 * dphi_dq_2 + torques

# Assemble
rhs_top = lambda_1 * dphi_dq_1 + lambda_2 * dphi_dq_2 + torques
rhs_combined = rhs_top.col_join(sym.Matrix([ddphi_dt_1, ddphi_dt_2]))
display(rhs_combined)
```

$$\begin{bmatrix} 0 \\ \lambda_1 + \lambda_2 \\ L_1\lambda_1\sin\left(\theta_1(t)\right) + 25\theta_1(t) - 27.4155677808038\sin^2\left(\frac{t}{2}\right) - 5.23598775598299 \\ L_2\lambda_2\sin\left(\theta_2(t)\right) + 25\theta_2(t) + 27.4155677808038\sin^2\left(\frac{t}{2}\right) + 5.23598775598299 \\ L_1\sin\left(\theta_1(t)\right)\frac{d^2}{dt^2}\theta_1(t) + L_1\cos\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 + \frac{d^2}{dt^2}y(t) \\ L_2\sin\left(\theta_2(t)\right)\frac{d^2}{dt^2}\theta_2(t) + L_2\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2 + \frac{d^2}{dt^2}y(t) \end{bmatrix}$$

[90]:
```
# Now construct the equations
eq = sym.Eq(EL_eq, rhs_combined)
display(eq)
```

$$\begin{bmatrix} 0.5L_1m_1\sin(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5L_1m_1\sin(\theta_1(t)) - 0.5L_1m_1\cos(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t) + 0.5L_2m_2\sin(\theta_2(t))\left(\frac{d}{d} \\ -0.5L_1m_1\sin(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t) - 0.5L_1m_1\cos(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 - 0.5L_1m_1\cos(\theta_1(t)) - 0.5L_2m_2\sin(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t) - \\ -1.0J_1\frac{d^2}{dt^2}\theta_1(t) - 0.25L_1^2m_1\frac{d^2}{dt^2}\theta_1(t) - \frac{L_1gm_1\sin(\theta_1(t))}{2} + 0.5L_1m_1x(t)\cos(\theta_1(\\ -1.0J_2\frac{d^2}{dt^2}\theta_2(t) - 0.25L_2^2m_2\frac{d^2}{dt^2}\theta_2(t) - \frac{L_2gm_2\sin(\theta_2(t))}{2} + 0.5L_2m_2x(t)\cos(\theta_2(\end{bmatrix}$$

$$\begin{bmatrix} 0 \\ \lambda_1 + \lambda_2 \\ L_1\lambda_1\sin(\theta_1(t)) + 25\theta_1(t) - 27.4155677808038\sin^2\left(\frac{t}{2}\right) - 5.23598775598299 \\ L_2\lambda_2\sin(\theta_2(t)) + 25\theta_2(t) + 27.4155677808038\sin^2\left(\frac{t}{2}\right) + 5.23598775598299 \\ L_1\sin(\theta_1(t))\frac{d^2}{dt^2}\theta_1(t) + L_1\cos(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + \frac{d^2}{dt^2}y(t) \\ L_2\sin(\theta_2(t))\frac{d^2}{dt^2}\theta_2(t) + L_2\cos(\theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 + \frac{d^2}{dt^2}y(t) \end{bmatrix}$$

[96]:
```
# Sub and solve
sub = {
    g: 9.8,
    L1: 1,
    L2: 1,
    m1: 1,
    m2: 1,
    J1: 1,
    J2: 1
}
# Add lambdas to qddot
variables_to_solve_for = list(qdd) + [lambda_1, lambda_2]

soln = sym.solve(eq.subs(sub), variables_to_solve_for, dict=True)
```

[99]:
```
solns = []
for sol in soln:
    for v in variables_to_solve_for:
        solns.append(sym.simplify(sol[v]))
```

[ ]:
```
# lambdify
# q = sym.Matrix([x, y, theta1, theta2, lambda_1, lambda_2])
lambdified = []

for index, v in enumerate(variables_to_solve_for):
    this_lmbd = sym.lambdify([*q, *qd, t], solns[index])
    lambdified.append(this_lmbd)
```

[107]:
```
def dyn(s,t):
    """
    System dynamics function (extended)

    Parameters
```

```
    ============
    s: NumPy array
        s = [x, xdot] is the extended system
        state vector, includng the position and
        the velocity of the particle

    Return
    ============
    sdot: NumPy array
        time derivative of input state vector,
        sdot = [xdot, xddot]
    """
    return np.
↪array([s[4],s[5],s[6],s[7],lambdified[0](*s,t),lambdified[1](*s,t),lambdified[2](*s,t),lamb
```

[109]:
```
# define initial state
initial = np.array([0, np.cos(np.pi / 15), np.pi/15, -np.pi/15, 0, 0, 0, 0])

#simulate trajectory
traj = simulate(dyn, initial, [0, 10], 0.01, integrate)
print('shape of traj: ', traj.shape)
```

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

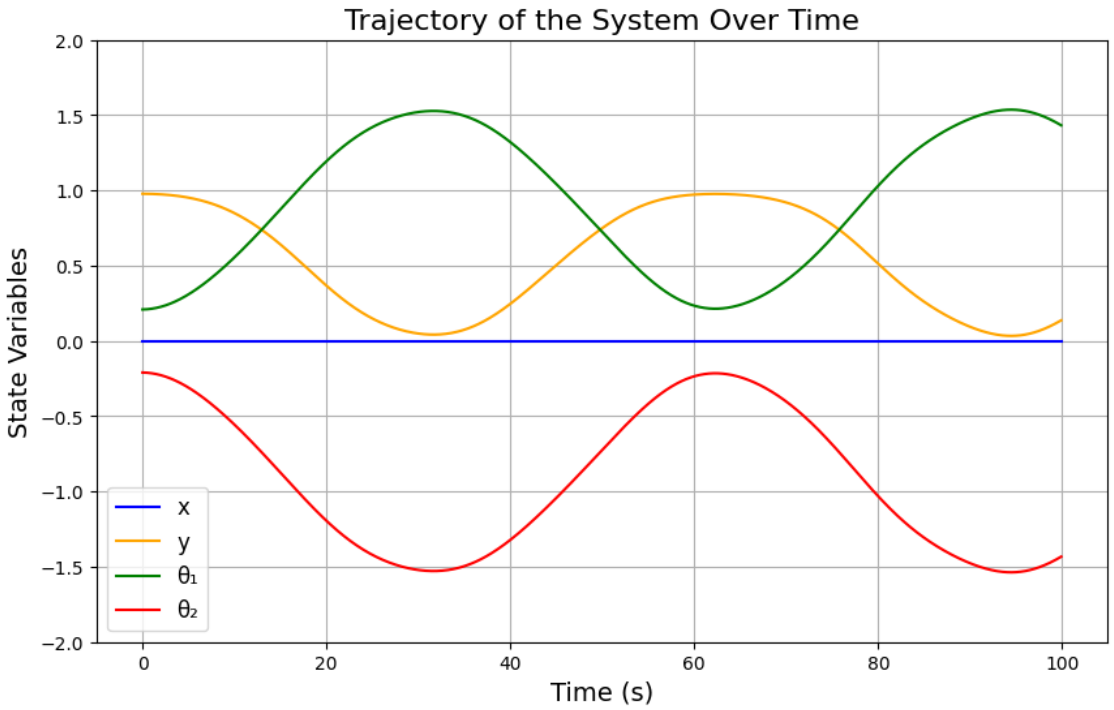
[119]:
```
### plotting...
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))

# Generate time points (assuming 1000 steps and 0.1 time interval)
time_points = np.arange(1000) * 0.1

# Assuming `traj` has 4 rows: [x, y, theta1, theta2]
plt.plot(time_points, traj[0].T, label='x', color='blue')
plt.plot(time_points, traj[1].T, label='y', color='orange')
plt.plot(time_points, traj[2].T, label=' ', color='green')
plt.plot(time_points, traj[3].T, label=' ', color='red')

# Add labels, legend, and set axis limits
plt.title("Trajectory of the System Over Time", fontsize=16)
plt.xlabel("Time (s)", fontsize=14)
plt.ylabel("State Variables", fontsize=14)
plt.ylim(-2, 2)
plt.grid(True)
plt.legend(fontsize=12)

# Show plot
plt.show()
```



[122]:
```
def animate(q_array,L1=1,L2=1,W1=0.2,W2=0.2,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ===============================================
    theta_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1, L2:
        length of the first and second legs
    W1, W2:
        width of the first and second legs
    T:
        length/seconds of animation duration

    Returns: None
    """

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

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

    ################################################
    # Getting data from pendulum angle trajectories.
#     xx1=L1*np.sin(theta_array[0])
#     yy1=-L1*np.cos(theta_array[0])
#     xx2=xx1+L2*np.sin(theta_array[0]+theta_array[1])
#     yy2=yy1-L2*np.cos(theta_array[0]+theta_array[1])
    xxA = q_array[0] #TODO: WHAT DO I NEED THIS FOR?
    yyA = q_array[1]
    N = len(q_array[0]) # Need this for specifying length of simulation

    ################################################
    # Define arrays containing data for frame axes
    # In each frame, the x and y axis are always fixed
    x_axis = np.array([0.05, 0.0])
    y_axis = np.array([0.0, 0.05])
    origin = np.array([0.0, 0.0])

    #rectangle
    rec1_bl = np.array([-W1/2., 0.0])
    rec1_br = np.array([W1/2., 0.0])
    rec1_tl = np.array([-W1/2., L1])
    rec1_tr = np.array([W1/2., L1])

    rec2_bl = np.array([-W2/2., 0.0])
    rec2_br = np.array([W2/2., 0.0])
    rec2_tl = np.array([-W2/2., L2])
    rec2_tr = np.array([W2/2., L2])
```

```
    # Use homogeneous tranformation to transfer these two axes/points
    # back to the fixed frame
    frame_a_x_axis = np.zeros((2,N))
    frame_a_y_axis = np.zeros((2,N))
    frame_a_origin = np.zeros((2,N))

    frame_b_x_axis = np.zeros((2,N))
    frame_b_y_axis = np.zeros((2,N))
    frame_b_origin = np.zeros((2,N))

    frame_c_x_axis = np.zeros((2,N))
    frame_c_y_axis = np.zeros((2,N))
    frame_c_origin = np.zeros((2,N))

    frame_d_x_axis = np.zeros((2,N))
    frame_d_y_axis = np.zeros((2,N))
    frame_d_origin = np.zeros((2,N))

    frame_e_x_axis = np.zeros((2,N))
    frame_e_y_axis = np.zeros((2,N))
    frame_e_origin = np.zeros((2,N))

    rec1_bl_w = np.zeros((2,N))
    rec1_br_w = np.zeros((2,N))
    rec1_tl_w = np.zeros((2,N))
    rec1_tr_w = np.zeros((2,N))

    rec2_bl_w = np.zeros((2,N))
    rec2_br_w = np.zeros((2,N))
    rec2_tl_w = np.zeros((2,N))
    rec2_tr_w = np.zeros((2,N))

    for i in range(N): # iteration through each time step
        # evaluate homogeneous transformations
        t_wa = np.array([[1,0,q_array[0][i]],
                         [0,1,q_array[1][i]],
                         [0,0,1]])
        t_ab = np.array([[np.cos(q_array[2][i]), -np.sin(q_array[2][i]), (L1/
↪2)*np.sin(q_array[2][i])],
                         [np.sin(q_array[2][i]),  np.cos(q_array[2][i]), -(L1/
↪2)*np.cos(q_array[2][i])],
                         [                    0,                     ␣
↪0, 1]])
        t_bd = np.array([[1,0,0],
                         [0,1,-L1/2],
                         [0,0,1]])
```

```
        t_ac = np.array([[np.cos(q_array[3][i]), -np.sin(q_array[3][i]), (L2/
↪2)*np.sin(q_array[3][i])],
                         [np.sin(q_array[3][i]),  np.cos(q_array[3][i]), -(L2/
↪2)*np.cos(q_array[3][i])],
                         [                    0,                      0,  
↪0, 1]])
        t_ce = np.array([[1,0,0],
                         [0,1,-L2/2],
                         [0,0,1]])

        #multiplying to get specific frame transforms...
        t_wb = np.matmul(t_wa,t_ab)
        t_wc = np.matmul(t_wa,t_ac)
        t_wd = np.matmul(t_wb,t_bd)
        t_we = np.matmul(t_wc,t_ce)

        # 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]
        frame_a_origin[:,i] = t_wa.dot([origin[0], origin[1], 1])[0:2]

        frame_b_x_axis[:,i] = t_wb.dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_b_y_axis[:,i] = t_wb.dot([y_axis[0], y_axis[1], 1])[0:2]
        frame_b_origin[:,i] = t_wb.dot([origin[0], origin[1], 1])[0:2]

        frame_c_x_axis[:,i] = t_wc.dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_c_y_axis[:,i] = t_wc.dot([y_axis[0], y_axis[1], 1])[0:2]
        frame_c_origin[:,i] = t_wc.dot([origin[0], origin[1], 1])[0:2]

        frame_d_x_axis[:,i] = t_wd.dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_d_y_axis[:,i] = t_wd.dot([y_axis[0], y_axis[1], 1])[0:2]
        frame_d_origin[:,i] = t_wd.dot([origin[0], origin[1], 1])[0:2]

        frame_e_x_axis[:,i] = t_we.dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_e_y_axis[:,i] = t_we.dot([y_axis[0], y_axis[1], 1])[0:2]
        frame_e_origin[:,i] = t_we.dot([origin[0], origin[1], 1])[0:2]

        #Setting up the rectangle
        rec1_bl_w[:,i] = t_wd.dot([rec1_bl[0], rec1_bl[1], 1])[0:2]
        rec1_br_w[:,i] = t_wd.dot([rec1_br[0], rec1_br[1], 1])[0:2]
        rec1_tl_w[:,i] = t_wd.dot([rec1_tl[0], rec1_tl[1], 1])[0:2]
        rec1_tr_w[:,i] = t_wd.dot([rec1_tr[0], rec1_tr[1], 1])[0:2]

        rec2_bl_w[:,i] = t_we.dot([rec2_bl[0], rec2_bl[1], 1])[0:2]
        rec2_br_w[:,i] = t_we.dot([rec2_br[0], rec2_br[1], 1])[0:2]
        rec2_tl_w[:,i] = t_we.dot([rec2_tl[0], rec2_tl[1], 1])[0:2]
```

```
        rec2_tr_w[:,i] = t_we.dot([rec2_tr[0], rec2_tr[1], 1])[0:2]

    ###################################
    # 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
        dict(name='World Frame X'),
        dict(name='World Frame Y'),
        dict(name='Rectangle1'),
        dict(name='Rectangle1'),
        dict(name='Rectangle1'),
        dict(name='Rectangle1'),
        dict(name='Rectangle2'),
        dict(name='Rectangle2'),
        dict(name='Rectangle2'),
        dict(name='Rectangle2'),

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

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

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

   ########################################
   # Defining the frames of the simulation.
   # This is what draws the lines from
   # joint to joint of the pendulum.
   frames=[dict(data=[# 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,x_axis[0]],
                           y=[0,x_axis[1]],
                           mode='lines',
                           line=dict(color='orange', width=3),
                           ),
                      dict(x=[0,y_axis[0]],
                           y=[0,y_axis[1]],
                           mode='lines',
                           line=dict(color='orange', width=3),
                           ),
                       # Leg 1
                      dict(x=[rec1_bl_w[0][k], rec1_br_w[0][k]],
                           y=[rec1_bl_w[1][k], rec1_br_w[1][k]],
                           mode='lines',
                           line=dict(color='red', width=3),
```

```
                    ),
                dict(x=[rec1_br_w[0][k], rec1_tr_w[0][k]],
                    y=[rec1_br_w[1][k], rec1_tr_w[1][k]],
                    mode='lines',
                    line=dict(color='red', width=3),
                    ),
                dict(x=[rec1_tl_w[0][k], rec1_tr_w[0][k]],
                    y=[rec1_tl_w[1][k], rec1_tr_w[1][k]],
                    mode='lines',
                    line=dict(color='red', width=3),
                    ),
                dict(x=[rec1_bl_w[0][k], rec1_tl_w[0][k]],
                    y=[rec1_bl_w[1][k], rec1_tl_w[1][k]],
                    mode='lines',
                    line=dict(color='red', width=3),
                    ),
                 # Leg 2
                dict(x=[rec2_bl_w[0][k], rec2_br_w[0][k]],
                    y=[rec2_bl_w[1][k], rec2_br_w[1][k]],
                    mode='lines',
                    line=dict(color='blue', width=3),
                    ),
                dict(x=[rec2_br_w[0][k], rec2_tr_w[0][k]],
                    y=[rec2_br_w[1][k], rec2_tr_w[1][k]],
                    mode='lines',
                    line=dict(color='blue', width=3),
                    ),
                dict(x=[rec2_tl_w[0][k], rec2_tr_w[0][k]],
                    y=[rec2_tl_w[1][k], rec2_tr_w[1][k]],
                    mode='lines',
                    line=dict(color='blue', width=3),
                    ),
                dict(x=[rec2_bl_w[0][k], rec2_tl_w[0][k]],
                    y=[rec2_bl_w[1][k], rec2_tl_w[1][k]],
                    mode='lines',
                    line=dict(color='blue', width=3),
                    )
                ]) for k in range(N)]


# add green ground at x = 0 as a line
data.append(dict(
    x=[xm, xM], y=[0, 0],
    mode='lines', name='Ground',
    line=dict(color='green', width=4)
))
########################################
```

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

[123]:
```
animate(traj[0:4],L1=1,L2=1,W1=0.2,W2=0.2,T=10)
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```