HW3

October 19, 2023

#HW3

[1]:
```
#IMPORT ALL NECESSARY PACKAGES AT THE TOP OF THE CODE
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

# Custom display helpers
from IPython.display import Markdown

def md_print(md_str: str):
    display(Markdown(md_str))

def lax_eq(equation):
    return sym.latex(equation , mode='inline')
```

[2]:
```
def integrate(f, xt, 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 = dt * f(xt)
    k2 = dt * f(xt+k1/2.)
    k3 = dt * f(xt+k2/2.)
    k4 = dt * f(xt+k3)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

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

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

    Return
    ============
    x_traj:
        simulated trajectory of x(t) 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)
        x = np.copy(xtraj[:,i])
    return tvec , xtraj

def animate_double_pend(theta_array,L1=1,L2=1,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:
        length of the first pendulum
    L2:
        length of the second pendulum
    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])
    N = len(theta_array[0]) # Need this for specifying length of simulation
```

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

    ############################
    # Defining data dictionary.
    # Trajectories are here.
    data=[dict(x=xx1, y=yy1,
               mode='lines', name='Arm',
               line=dict(width=2, color='blue')
              ),
          dict(x=xx1, y=yy1,
               mode='lines', name='Mass 1',
               line=dict(width=2, color='purple')
              ),
          dict(x=xx2, y=yy2,
               mode='lines', name='Mass 2',
               line=dict(width=2, color='green')
              ),
          dict(x=xx1, y=yy1,
               mode='markers', name='Pendulum 1 Traj',
               marker=dict(color="purple", size=2)
              ),
          dict(x=xx2, y=yy2,
               mode='markers', name='Pendulum 2 Traj',
               marker=dict(color="green", size=2)
              ),
        ]

    ################################
    # Preparing simulation layout.
    # Title and axis ranges are here.
    layout=dict(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='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=[dict(x=[0,xx1[k],xx2[k]],
                            y=[0,yy1[k],yy2[k]],
                            mode='lines',
                            line=dict(color='red', 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)),
                      ]) for k in range(N)]

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

[3]:
```
def problem1():
    x , y = sym.symbols('x,y')

    q = sym.Matrix([x,y])
    f1 = sym.sin(x+y)* sym.sin(x-y)

    J = sym.Matrix([f1]).jacobian(q).T
    display(J)

    H = J.jacobian(q).T
    display(H)

    xy_value = [(x,0) , (y,sym.pi / 2)]
```

```
    md_print(f"""
## Problem1 solution
At x,y = 0,pi/2
Jacobian of F is
{lax_eq(J.subs(xy_value))}
Hessian of F is
{lax_eq(H.subs(xy_value))}

Which prove this value is a necessary and sufficient location of local␣
 ↪minimizer
          """)

problem1()
```

$$\begin{bmatrix}\sin(x-y)\cos(x+y)+\sin(x+y)\cos(x-y)\\ \sin(x-y)\cos(x+y)-\sin(x+y)\cos(x-y)\end{bmatrix}$$

$$\begin{bmatrix}-2\sin(x-y)\sin(x+y)+2\cos(x-y)\cos(x+y) & 0\\ 0 & -2\sin(x-y)\sin(x+y)-2\cos(x-y)\cos(x+y)\end{bmatrix}$$

0.1 Problem1 solution

At x,y = 0,pi/2 Jacobian of F is $\begin{bmatrix}0\\0\end{bmatrix}$ Hessian of F is $\begin{bmatrix}2&0\\0&2\end{bmatrix}$

Which prove this value is a necessary and sufficient location of local minimizer

[4]:
```
def problem2():
    t= sym.symbols('t')
    k1 , k2 = sym.symbols(r'k_1,k_2') # spring constants
    m1 , m2 = sym.symbols(r"m_1 , m_2")
    L = sym.symbols("L")

    theta1 = sym.Function(r'\theta_1')(t)
    theta2 = sym.Function(r'\theta_2')(t)
    x1 = sym.tan(theta1) *L
    x2 = sym.tan(theta1 + theta2) *L - x1
    x1dot = x1.diff(t)
    x2dot = x2.diff(t)

    # x1= sym.Function(r'x_1')(t)
    # x2= sym.Function(r'x_2')(t)

    t1_dd = theta1.diff(t).diff(t)
    t2_dd = theta2.diff(t).diff(t)

    KE_x =  0.5 * m1 * (x1dot**2) + 0.5 * m2 *(x2dot**2)
    V_x =  0.5*k1*(x1**2) + 0.5 * k2 * ( (x1-x2)**2 )
```

```
    L = sym.simplify(KE_x - V_x)

    q = sym.Matrix([theta1,theta2])
    q_dot = sym.Matrix([theta1.diff(t) ,theta2.diff(t)])
    q_ddot = sym.Matrix([t1_dd , t2_dd])

    dL_dq = sym.simplify(sym.Matrix([L]).jacobian(q).T)
    dL_dq_dot = sym.simplify(sym.Matrix([L]).jacobian(q_dot).T)

    eu_la = (dL_dq_dot.diff(t) - dL_dq)

    eu_la_eq = sym.Eq(eu_la , sym.Matrix([0,0]) )
    solved_el = sym.solve(eu_la_eq , [t1_dd , t2_dd] , dict=True)[0]

    # Solve for theta_dd
    solved_eq_dict = {}
    md_print("solution:")
    for var in q_ddot:
        # eq_local = sym.simplify(sym.Eq(var , solved_el[var]))
        md_print(f'{lax_eq(var)} = {lax_eq(solved_el[var])}')
        # display(sym.expand(eq_local))
        # solved_eq_dict[var] = solved_el[var]
problem2()
```

solution:

$$\frac{d^2}{dt^2}\theta_1(t) = -k_1\cos^2(\theta_1(t))\tan(\theta_1(t))/m_1 + 2.0k_2\cos^4(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))/m_1 + 2.0k_2\cos^4(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))/m_1 - 4.0k_2\cos^4(\theta_1(t))\tan^3(\theta_1(t))/m_1 - 4.0k_2\cos^4(\theta_1(t))\tan(\theta_1(t))/m_1 - k_2\cos^2(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))/m_1 + 2.0k_2\cos^2(\theta_1(t))\tan(\theta_1(t))/m_1 - \frac{2.0\sin(\theta_1(t))\left(\frac{d}{dt}\theta_1(t)\right)^2}{\cos(\theta_1(t))}$$

$$\frac{d^2}{dt^2}\theta_2(t) = \frac{k_1 m_2\cos^3(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{k_1 m_2\cos^3(\theta_1(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{k_1 m_2\cos(\theta_1(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{k_2 m_1\cos(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{2.0k_2 m_1\cos(\theta_1(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{2.0k_2 m_2\cos^5(\theta_1(t))\tan^3(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{2.0k_2 m_2\cos^5(\theta_1(t))\tan^3(\theta_1(t)+\theta_2(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{4.0k_2 m_2\cos^5(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))\tan^3(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{4.0k_2 m_2\cos^5(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{2.0k_2 m_2\cos^5(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{2.0k_2 m_2\cos^5(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{4.0k_2 m_2\cos^5(\theta_1(t))\tan^3(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{4.0k_2 m_2\cos^5(\theta_1(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{k_2 m_2\cos^3(\theta_1(t))\tan^3(\theta_1(t)+\theta_2(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{2.0k_2 m_2\cos^3(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{2.0k_2 m_2\cos^3(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))\tan^2(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{3.0k_2 m_2\cos^3(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{4.0k_2 m_2\cos^3(\theta_1(t))\tan^3(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{6.0k_2 m_2\cos^3(\theta_1(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} - \frac{k_2 m_2\cos(\theta_1(t))\tan(\theta_1(t)+\theta_2(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{2.0k_2 m_2\cos(\theta_1(t))\tan(\theta_1(t))}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} + \frac{2.0m_1 m_2\sin(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1 m_2\cos(\theta_1(t))\tan^2(\theta_1(t)+\theta_2(t))+m_1 m_2\cos(\theta_1(t))} +$$

$$\frac{2.0m_1m_2\sin\left(\theta_1(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)} - \frac{2.0m_1m_2\cos\left(\theta_1(t)\right)\tan^3\left(\theta_1(t)+\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)} -$$
$$\frac{4.0m_1m_2\cos\left(\theta_1(t)\right)\tan^3\left(\theta_1(t)+\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)} - \frac{2.0m_1m_2\cos\left(\theta_1(t)\right)\tan^3\left(\theta_1(t)+\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)} -$$
$$\frac{2.0m_1m_2\cos\left(\theta_1(t)\right)\tan\left(\theta_1(t)+\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)} - \frac{4.0m_1m_2\cos\left(\theta_1(t)\right)\tan\left(\theta_1(t)+\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)} -$$
$$\frac{2.0m_1m_2\cos\left(\theta_1(t)\right)\tan\left(\theta_1(t)+\theta_2(t)\right)\left(\frac{d}{dt}\theta_2(t)\right)^2}{m_1m_2\cos\left(\theta_1(t)\right)\tan^2\left(\theta_1(t)+\theta_2(t)\right)+m_1m_2\cos\left(\theta_1(t)\right)}$$

[5]:
```
# P3

# Define x1 x2 y1 y2 w.r.t theta
t= sym.symbols('t')
k1 , k2 = sym.symbols(r'k_1,k_2') # spring constants
m1 , m2 = sym.symbols(r"m_1 , m_2")
L = sym.symbols("L")

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

t1_d = theta1.diff(t)
t2_d = theta2.diff(t)
t1_dd = t1_d.diff(t)
t2_dd = t2_d.diff(t)

x1 = sym.tan(theta1) *L
x2 = sym.tan(theta1 + theta2) *L - x1
y1 = sym.tan(theta1) *L
y2 = sym.tan(theta1+ theta2) *L

# Now we use F=ma
F_x  = m1* x1.diff(t).diff(t) +  m2*x2.diff(t).diff(t)
F_y  = m1* y1.diff(t).diff(t) +  m2*y2.diff(t).diff(t)

# We just randomly selecting some values for all variables
subs_list = ([m1 , 1] , [m2 ,2 ] , [L,5])
display(F_x.subs(subs_list))
display (t1_dd)
F_x_lam = sym.lambdify( [theta1 , theta2 , t1_d , t2_d , t1_dd , t2_dd] , F_x.
 ↪subs(subs_list) )
F_y_lam = sym.lambdify( [theta1 , theta2 , t1_d , t2_d , t1_dd , t2_dd] , F_y.
 ↪subs(subs_list) )

fx_val = F_x_lam(0.1 , 0.2 , 1 , 2 ,10, 20)
fy_val = F_y_lam(0.1 , 0.2 , 1 , 2 ,10, 20)

md_print(f"""
## Problem 3 solution
Using F=m1a1 + m2a2 with both x and y.
```

```
After express both x and y in terms of theta.
Plug in some random value for m1 m2 L t1 t2 t1d t2d t1dd t2dd,
Fx = {fx_val}
Fy = {fy_val}
At the same condition, the Force from x and y coordinate are not the same.
      """)
```

$$20 \left(\tan^2\left(\theta_1(t) + \theta_2(t)\right) + 1\right) \left(\frac{d}{dt}\theta_1(t) + \frac{d}{dt}\theta_2(t)\right)^2 \tan\left(\theta_1(t) + \theta_2(t)\right) +$$
$$10 \left(\tan^2\left(\theta_1(t) + \theta_2(t)\right) + 1\right) \left(\frac{d^2}{dt^2}\theta_1(t) + \frac{d^2}{dt^2}\theta_2(t)\right) - 10 \left(\tan^2\left(\theta_1(t)\right) + 1\right) \tan\left(\theta_1(t)\right) \left(\frac{d}{dt}\theta_1(t)\right)^2 -$$
$$5 \left(\tan^2\left(\theta_1(t)\right) + 1\right) \frac{d^2}{dt^2}\theta_1(t)$$

$$\frac{d^2}{dt^2}\theta_1(t)$$

0.2 Problem 3 solution

Using F=m1a1 + m2a2 with both x and y. After express both x and y in terms of theta. Plug in some random value for m1 m2 L t1 t2 t1d t2d t1dd t2dd, Fx = 338.1984087816165 Fy = 441.2320083416084 At the same condition, the Force from x and y coordinate are not the same.

[6]:
```
# P4
t= sym.symbols('t')

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


theta1_dot = theta1.diff(t)
theta2_dot = theta2.diff(t)
theta1_ddot = theta1_dot.diff(t)
theta2_ddot = theta2_dot.diff(t)

R1 = 1
R2 = 1
m1 = 1
m2 = 1
# R1 , R2, m1 , m2  = sym.symbols('R_1,R_2,m_1,m_2')
# subs_list = [(R1, 1), (R2, 1), (m1, 1), (m2, 1)]
g = 9.8 # Define positive G

# Origin is top of the pendulum, the corner of theta1.
# Define x is down, which is also the dir of g.
# Z is defined of the screen.
# wight right hand rule, y is right.

# The XY for m1 and m2 are position wrt origin. not diff between them.
```

```
# For m1
m1_x = R1*sym.cos(theta1)
m1_y = R1*sym.sin(theta1)

# For m2
m2_x = R1*sym.cos(theta1) + R2*sym.cos(theta1+theta2)
m2_y = R1*sym.sin(theta1) + R2*sym.sin(theta1+theta2)

# Now do the energy stuff
total_kinematic = 0.5 * m1 * (m1_x.diff(t)**2 + m1_y.diff(t)**2) + 0.5 * m2 *␣
 ↪(m2_x.diff(t)**2 +
                                                                              ␣
 ↪m2_y.diff(t)**2)
total_potential = -m1*g*m1_x + -m2*g*m2_x

lagrangian = sym.simplify(total_kinematic - total_potential)
# lagrangian = lagrangian.subs(subs_list)

q = sym.Matrix([theta1, theta2])
q_dot = sym.Matrix([theta1_dot, theta2_dot])
q_ddot = sym.Matrix([theta1_ddot, theta2_ddot])

dL_dq = sym.Matrix([lagrangian]).jacobian(q).T
dL_dqdot = sym.Matrix([lagrangian]).jacobian(q_dot).T

euler_lagrange = sym.simplify(dL_dqdot.diff(t) - dL_dq)

el_equation = sym.Eq(euler_lagrange, sym.Matrix([0, 0]))
solved_el = sym.solve(euler_lagrange, [theta1_ddot, theta2_ddot], dict=True)[0]

solved_eq_dict = {}
for var in q_ddot:
    solved_eq_dict[var] = solved_el[var]


# Hamiltonian

p_term = dL_dqdot.T
# matrix math end up give a scaler inside matrix
Ham = (p_term * q_dot)[0] - lagrangian
md_print(f"## Problem 4 solution\n Hamiltonian")
display(sym.simplify(Ham))
```

0.3 Problem 4 solution

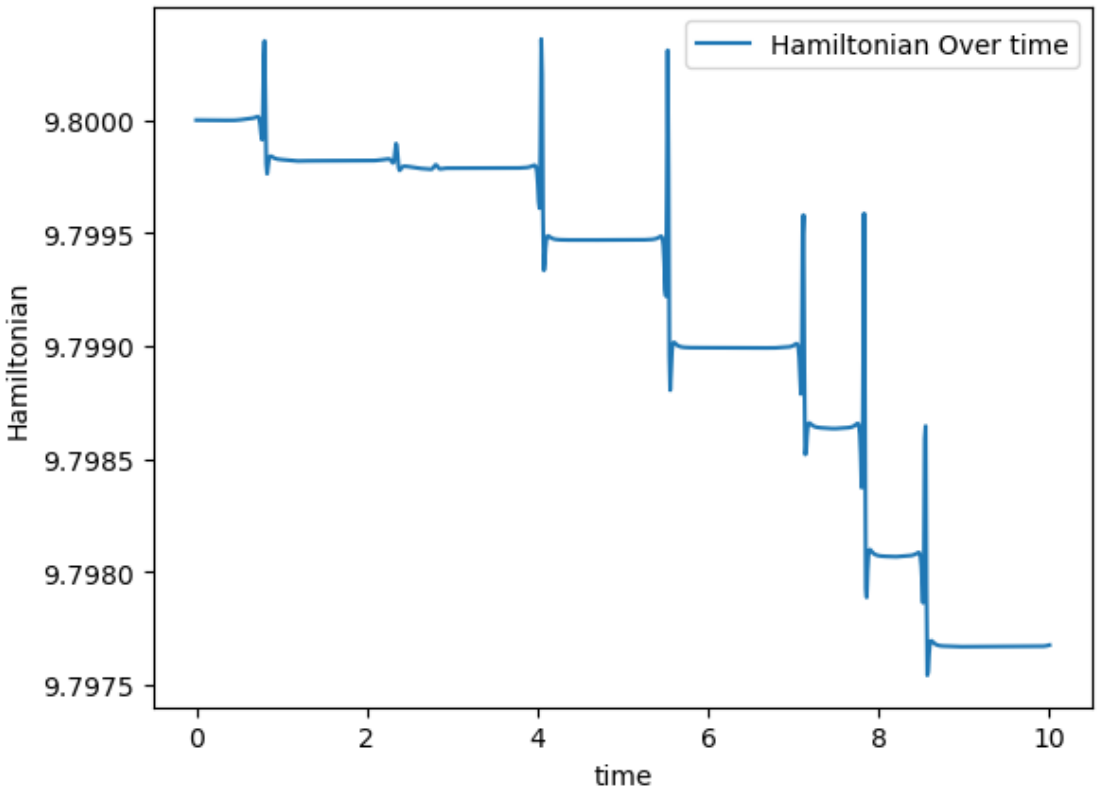
Hamiltonian

$$-9.8\cos\left(\theta_1(t)+\theta_2(t)\right)-19.6\cos\left(\theta_1(t)\right)+1.0\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2+1.0\cos\left(\theta_2(t)\right)\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)+$$
$$1.5\left(\frac{d}{dt}\theta_1(t)\right)^2+1.0\frac{d}{dt}\theta_1(t)\frac{d}{dt}\theta_2(t)+0.5\left(\frac{d}{dt}\theta_2(t)\right)^2$$

[7]:
```
# P5
initial_state = [-np.pi / 2, -np.pi / 2, 0, 0]
t_range = [0,10]
time_step = 0.01
lambda_dict = {}

for var in q_ddot:
    acceleration_function = sym.simplify(solved_eq_dict[var])
    lambda_func = sym.lambdify([theta1, theta2, theta1_dot, theta2_dot],␣
 ↪acceleration_function )
    lambda_dict[var] = lambda_func

# now simulate

def system_equation(state):
    '''
    Takes theta1, theta2 , theta1dot, theta2dot as input
    output theta1dot, theta2dot , theta1ddot, theta2ddot
    '''

    t1_pos = state[0]
    t2_pos = state[1]
    t1_v = state[2]
    t2_v = state[3]
    t1_a = lambda_dict[theta1_ddot](t1_pos,t2_pos , t1_v,t2_v)
    t2_a = lambda_dict[theta2_ddot](t1_pos,t2_pos , t1_v,t2_v)

    return np.array([t1_v,t2_v,t1_a,t2_a])

lam_ham = sym.lambdify([theta1, theta2, theta1_dot, theta2_dot] , Ham)

time_vec , q_traj = simulate(system_equation , initial_state , t_range ,␣
 ↪time_step , integrate)
ham_vec = []
for t1 , t2 , t1d , t2d in zip(q_traj[0],q_traj[1],q_traj[2],q_traj[3]):
    ham_vec.append(lam_ham(t1,t2,t1d,t2d))

md_print("Problem 5 solution ")
plt.plot(time_vec ,ham_vec , label = "Hamiltonian Over time")
plt.xlabel("time")
plt.ylabel("Hamiltonian")
plt.legend()
```

```
plt.show()
```

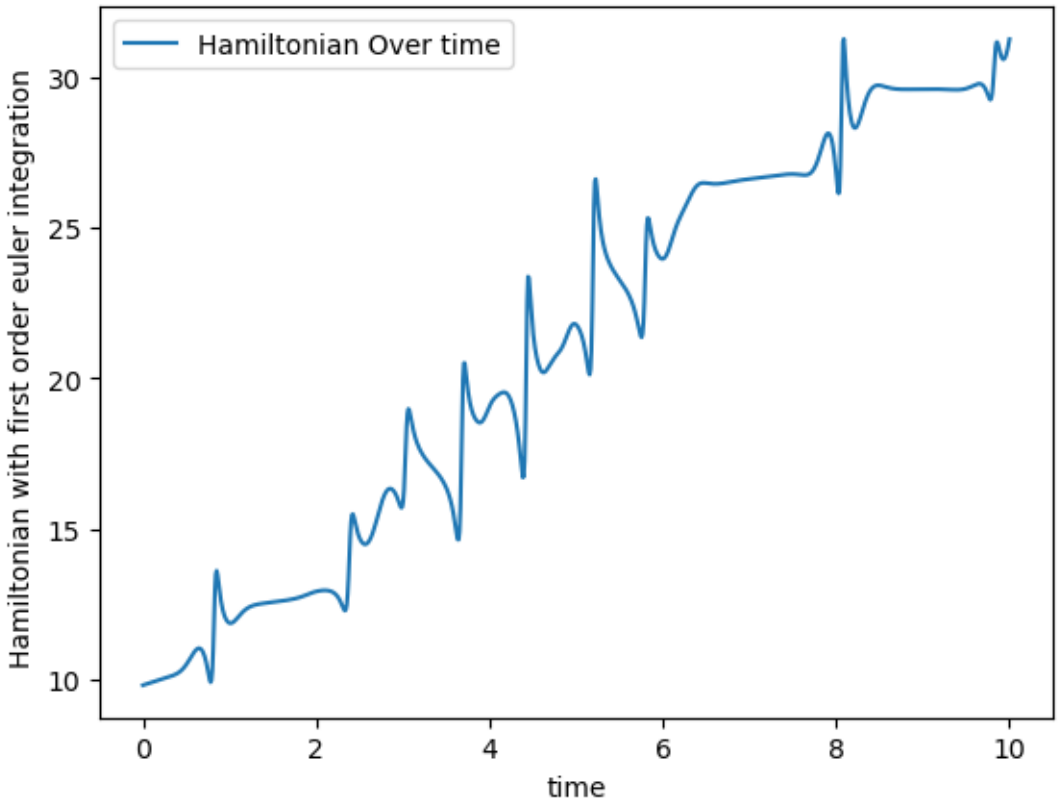
Problem 5 solution



[8]:
```
# P6

def euler_integrate(diff_function, init_state , dt):

    return init_state + diff_function(init_state)*dt


time_vec, q_traj_euler = simulate(system_equation, initial_state, t_range,␣
 ↪time_step, euler_integrate)
ham_vec_euler = []
for t1, t2, t1d, t2d in zip(q_traj_euler[0], q_traj_euler[1], q_traj_euler[2],␣
 ↪q_traj_euler[3]):
    ham_vec_euler.append(lam_ham(t1,t2,t1d,t2d))

md_print("Problem 6 solution ")
plt.plot(time_vec ,ham_vec_euler , label = "Hamiltonian Over time")
plt.xlabel("time")
plt.ylabel("Hamiltonian with first order euler integration ")
```

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

md_print("""
**With the euler integration, the resulting hamiltonian is diverging
rapidly over time. This mean the euler is diverging a lot more then the
Runge-Kutta integration.**
""")
```

Problem 6 solution



With the euler integration, the resulting hamiltonian is diverging rapidly over time. This mean the euler is diverging a lot more then the Runge-Kutta integration.

[9]:
```
# p7
lambda_scaler = sym.symbols('\lambda')

constrain_phi = m2_x**2 + m2_y**2  - 2
phi_dt1 = sym.simplify(constrain_phi.diff(theta1))
phi_dt2 = sym.simplify(constrain_phi.diff(theta2))

constrain_ddt = sym.simplify(constrain_phi.diff(t).diff(t))
```

```
constrained_system = sym.Eq(
    sym.Matrix([euler_lagrange[0] - lambda_scaler * phi_dt1 ,
     euler_lagrange[1] - lambda_scaler * phi_dt2 ,
      constrain_ddt]),
    sym.Matrix( [0,0,0])
)

p7_vars = [theta1_ddot, theta2_ddot,lambda_scaler]
p7_solution = sym.solve(constrained_system , p7_vars , dict=True)

p7_lambdafied = {}
for var in  p7_vars:
    sol = sym.simplify(p7_solution[0][var])
    display(sol)
    p7_lambdafied[var] = sym.lambdify([theta1, theta2, theta1_dot, theta2_dot],␣
 ↪sol)


def system_equation_7(state):
    '''
    Takes theta1, theta2 , theta1dot, theta2dot as input
    output theta1dot, theta2dot , theta1ddot, theta2ddot
    '''

    t1_pos = state[0]
    t2_pos = state[1]
    t1_v = state[2]
    t2_v = state[3]
    t1_a = p7_lambdafied[theta1_ddot](t1_pos,t2_pos , t1_v,t2_v)
    t2_a = p7_lambdafied[theta2_ddot](t1_pos,t2_pos , t1_v,t2_v)
    return np.array([t1_v,t2_v,t1_a,t2_a])


time_vec_p7, q_traj_p7 = simulate(system_equation_7,
                            initial_state,
                            t_range,
                            time_step,
                            integrate)

display(q_traj_p7)
animate_double_pend(q_traj_p7[0:2,:] , L1=1,L2=1,T=10)
```

$$\frac{-3.26666666666667 \sin\left(\theta_1(t) + \theta_2(t)\right) - 6.53333333333333 \sin\left(\theta_1(t)\right) + 0.666666666666667 \sin\left(\theta_2(t)\right) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t)}{0.666666666666667 \cos\left(\theta_2(t)\right) + 1.0}$$

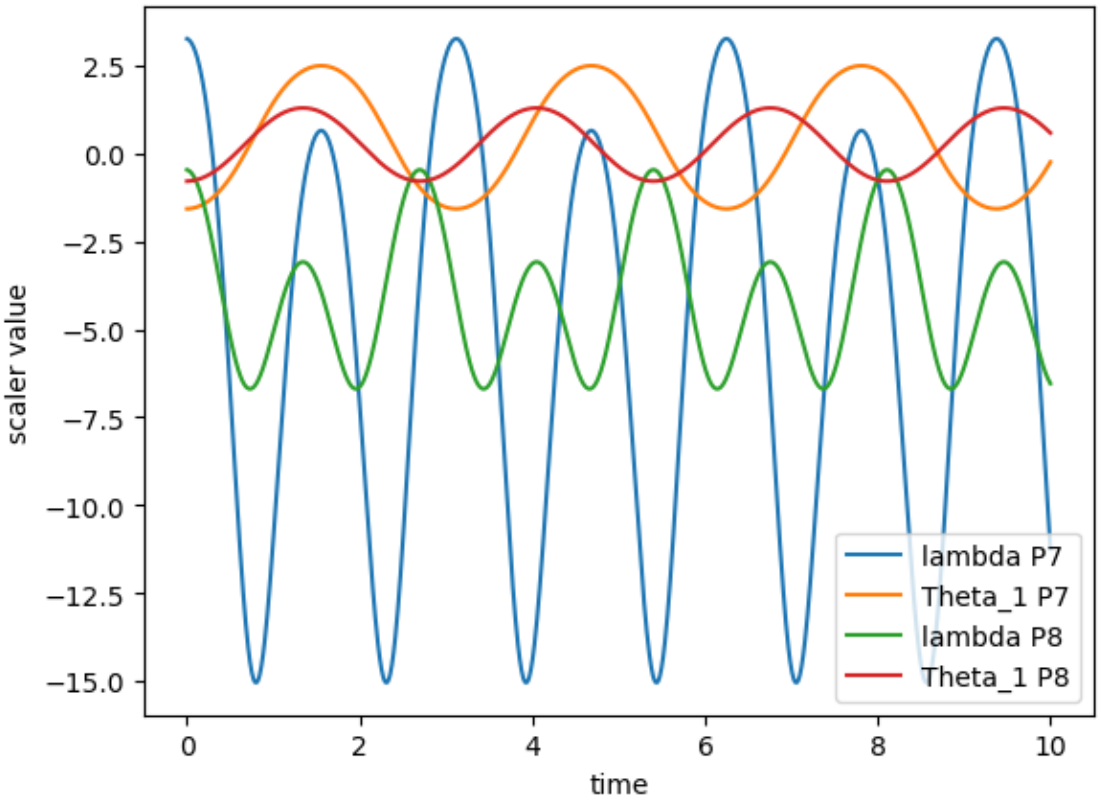
$$-\frac{\left(\frac{d}{dt}\theta_2(t)\right)^2}{\tan\left(\theta_2(t)\right)}$$

$$\frac{2.85833333333333\sin\left(\theta_1(t)\right)}{\sin\left(\theta_2(t)\right)} - 3.26666666666667\cos\left(\theta_1(t)\right) - 0.333333333333333\cos\left(\theta_2(t)\right)\left(\frac{d}{dt}\theta_1(t)\right)^2 - 0.33333333333$$

<IPython.core.display.HTML object>

[10]:
```
time_vec_p8, q_traj_p8 = simulate(system_equation_7,
                            [-np.pi / 4, -np.pi / 4, 0, 0],
                            t_range,
                            time_step,
                            integrate)


animate_double_pend(q_traj_p8[0:2,:])

lambda_over_time = []


lambda_values_p7 = p7_lambdafied[lambda_scaler](q_traj_p7[0], q_traj_p7[1],␣
 ↪q_traj_p7[2], q_traj_p7[3])
lambda_values_p8 = p7_lambdafied[lambda_scaler](q_traj_p8[0], q_traj_p8[1],␣
 ↪q_traj_p8[2], q_traj_p8[3])
plt.plot(time_vec, lambda_values_p7 , label = "lambda P7")
plt.plot(time_vec, q_traj_p7[0] , label = 'Theta_1 P7')
plt.plot(time_vec, lambda_values_p8 , label = "lambda P8")
plt.plot(time_vec, q_traj_p8[0] , label = 'Theta_1 P8')
plt.legend()
plt.xlabel("time")
plt.ylabel("scaler value")
plt.show()
```

<IPython.core.display.HTML object>



0.4 Problem 8 solution.

With the initial condition at imposable locations, the simulation still gives a similar motion just with joints at a different location.

I think the reason can be explained by looking at the plot of lambda value between both question.

In P8, the magnitude of lambda value never reaches 0. In the constrained euler-Lagrange equation, lambda loosely represented the forces needed. The amount of force needed will simply scale as the joints becomes running out of the constraint.

0.5 Collaboration list

- Srikanth Schelbert
- Graham Clifford
- Shail Dalal
- Carter DiOrio

The notebook is generated locally. Thus no google collab is available.