Two pendulum on moving platform

September 28, 2023

0.0.1 Importing the python packages

[1]:
```
import sympy as sm
import sympy.physics.mechanics as me
me.init_vprinting(use_latex='mathjax')
```

0.0.2 Initializing the symbols

[2]:
```
x, theta1, theta2 = me.dynamicsymbols('x theta_1 theta_2')

l1, l2 = sm.symbols('l_1 l_2')
m1, m2, m3 = sm.symbols('m_1 m_2 m_3')
I1, I2, I3 = sm.symbols('I_1 I_2 I_3')

g = sm.symbols('g')
```

[3]:
```
x
```

[3]: x

[4]:
```
l1
```

[4]: l_1

0.0.3 Defining the reference frames and CG position of each rigid body

[5]:
```
N = me.ReferenceFrame('N')
O_N = me.Point('O_N')
O_N.set_vel(N, 0)

B1 = N.orientnew('B_1','Axis',[N.z,0])
O_B1 = O_N.locatenew('O_B1', x*N.x)
O_B1.set_vel(B1, 0)

B2 = B1.orientnew('B_2','Axis',[B1.z,theta1])
O_B2 = O_B1.locatenew('O_B2', -(l1/2)*B2.y)
O_B2.set_vel(B2, 0)
```

```python
B3 = B2.orientnew('B_3','Axis',[B2.z,theta2])
O_B3 = O_B2.locatenew('O_B3', -(l1/2)*B2.y-(l2/2)*B3.y)
O_B3.set_vel(B3, 0)

B3.dcm(N)
```

[5]:

$$\begin{bmatrix} -\sin(\theta_1)\sin(\theta_2)+\cos(\theta_1)\cos(\theta_2) & \sin(\theta_1)\cos(\theta_2)+\sin(\theta_2)\cos(\theta_1) & 0 \\ -\sin(\theta_1)\cos(\theta_2)-\sin(\theta_2)\cos(\theta_1) & -\sin(\theta_1)\sin(\theta_2)+\cos(\theta_1)\cos(\theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

0.0.4 Computing the positions of the CG of the bodies

[6]:
```python
O_B1.pos_from(O_N)
```

[6]:

$$x\hat{\mathbf{n}}_{\mathbf{x}}$$

[7]:
```python
O_B2.pos_from(O_N)
```

[7]:

$$-\frac{l_1}{2}\hat{\mathbf{b}}_{\mathbf{2y}} + x\hat{\mathbf{n}}_{\mathbf{x}}$$

[8]:
```python
O_B3.pos_from(O_N)
```

[8]:

$$-l_1\hat{\mathbf{b}}_{\mathbf{2y}} - \frac{l_2}{2}\hat{\mathbf{b}}_{\mathbf{3y}} + x\hat{\mathbf{n}}_{\mathbf{x}}$$

[9]:
```python
((O_B3.pos_from(O_N)).express(N)).simplify()
```

[9]:

$$(l_1\sin(\theta_1) + \frac{l_2\sin(\theta_1+\theta_2)}{2} + x)\hat{\mathbf{n}}_{\mathbf{x}} + (-l_1\cos(\theta_1) - \frac{l_2\cos(\theta_1+\theta_2)}{2})\hat{\mathbf{n}}_{\mathbf{y}}$$

0.0.5 Computing the angular velocities of the bodies

[10]:
```python
w_B1 = B1.ang_vel_in(N)
w_B1
```

[10]:

$$0$$

[11]:
```python
w_B2 = B2.ang_vel_in(N).express(N)
w_B2
```

[11]:

$$\dot{\theta}_1\hat{\mathbf{n}}_{\mathbf{z}}$$

[12]:
```python
w_B3 = B3.ang_vel_in(N).express(N)
w_B3
```

[12]:

$$(\dot{\theta}_1 + \dot{\theta}_2)\hat{\mathbf{n}}_{\mathbf{z}}$$

0.0.6 Computing the linear velocities of the CGs of the bodies

[13]:
```
v_B1 = O_B1.vel(N)
v_B1
```

[13]:

$$\dot{x}\hat{\mathbf{n}}_{\mathbf{x}}$$

[14]:
```
v_B2 = O_B2.vel(N)
v_B2
```

[14]:

$$\frac{l_1\dot{\theta}_1}{2}\hat{\mathbf{b}}_{\mathbf{2}_\mathbf{x}} + \dot{x}\hat{\mathbf{n}}_{\mathbf{x}}$$

[15]:
```
v_B3 = O_B3.vel(N)
v_B3
```

[15]:

$$l_1\dot{\theta}_1\hat{\mathbf{b}}_{\mathbf{2}_\mathbf{x}} + \frac{l_2\left(\dot{\theta}_1 + \dot{\theta}_2\right)}{2}\hat{\mathbf{b}}_{\mathbf{3}_\mathbf{x}} + \dot{x}\hat{\mathbf{n}}_{\mathbf{x}}$$

[16]:
```
(v_B3.express(N)).simplify()
```

[16]:

$$(l_1\cos(\theta_1)\dot{\theta}_1 + \frac{l_2\left(\dot{\theta}_1 + \dot{\theta}_2\right)\cos(\theta_1 + \theta_2)}{2} + \dot{x})\hat{\mathbf{n}}_{\mathbf{x}} + (l_1\sin(\theta_1)\dot{\theta}_1 + \frac{l_2\left(\dot{\theta}_1 + \dot{\theta}_2\right)\sin(\theta_1 + \theta_2)}{2})\hat{\mathbf{n}}_{\mathbf{y}}$$

0.0.7 Defining the rigid bodies

[17]:
```
BodyB1=me.RigidBody('BodyB1', O_B1, B1, m1, (me.inertia(B1, I1, I1, I1), O_B1))
```

[18]:
```
BodyB2=me.RigidBody('BodyB2', O_B2, B2, m2, (me.inertia(B2, I2, 0, I2), O_B2))
```

[19]:
```
BodyB3=me.RigidBody('BodyB3', O_B3, B3, m3, (me.inertia(B3, I3, 0, I3), O_B3))
```

0.0.8 Computation of kinetic energy of the system

[20]:
```
KE = me.kinetic_energy(N, BodyB1, BodyB2, BodyB3)
KE
```

[20]:

$$\frac{I_2\dot{\theta}_1^2}{2} + \frac{I_3\left(\dot{\theta}_1 + \dot{\theta}_2\right)\dot{\theta}_1}{2} + \frac{I_3\left(\dot{\theta}_1 + \dot{\theta}_2\right)\dot{\theta}_2}{2} + \frac{m_1\dot{x}^2}{2} + \frac{m_2\left(\frac{l_1^2\dot{\theta}_1^2}{4} + l_1\cos(\theta_1)\dot{\theta}_1\dot{x} + \dot{x}^2\right)}{2} + \frac{m_3\left(l_1^2\dot{\theta}_1^2 + l_1 l_2\left(\dot{\theta}_1 + \dot{\theta}_2\right)\cos(\theta_2)\dot{\theta}_1 + 2l_1\cos(\theta_1)\dot{\theta}_1\dot{x} + \frac{l_2^2\left(\dot{\theta}_1 + \dot{\theta}_2\right)^2}{4} + l_2\left(-\sin(\theta_1)\sin(\theta_2) + \cos(\theta_1)\cos(\theta_2)\right)\left(\dot{\theta}_1 + \dot{\theta}_2\right.\right.}{2}$$

0.0.9 Computing the potential energy of each body

[21]:
```
g_vec = -g*N.y
g_vec
```

[21]:

$$-g\hat{\mathbf{n}}_{\mathbf{y}}$$

[22]:
```
BodyB1.potential_energy = -m1*(O_B1.pos_from(O_N)).dot(g_vec)
BodyB1.potential_energy
```

[22]:

$$0$$

[23]:
```
BodyB2.potential_energy = -m2*(O_B2.pos_from(O_N)).dot(g_vec)
BodyB2.potential_energy
```

[23]:

$$-\frac{gl_1 m_2 \cos(\theta_1)}{2}$$

[24]:
```
BodyB3.potential_energy = -m3*(O_B3.pos_from(O_N)).dot(g_vec)
BodyB3.potential_energy.simplify()
```

[24]:

$$-\frac{gm_3 \cdot (2l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2))}{2}$$

0.0.10 Computation of potential energy of the system

[25]:
```
PE = me.potential_energy(BodyB1, BodyB2, BodyB3).simplify()
PE
```

[25]:

$$-\frac{g(l_1 m_2 \cos(\theta_1) + 2l_1 m_3 \cos(\theta_1) + l_2 m_3 \cos(\theta_1 + \theta_2))}{2}$$

0.0.11 Computing the Lagrangian

[26]:
```
L = me.Lagrangian(N, BodyB1, BodyB2, BodyB3)

L.simplify()
```

[26]:

$$\frac{I_2\dot{\theta}_1^2}{2} + \frac{I_3\left(\dot{\theta}_1 + \dot{\theta}_2\right)\dot{\theta}_1}{2} + \frac{I_3\left(\dot{\theta}_1 + \dot{\theta}_2\right)\dot{\theta}_2}{2} + \frac{gl_1 m_2 \cos(\theta_1)}{2} + \frac{gm_3 \cdot (2l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2))}{2} +$$
$$\frac{m_1\dot{x}^2}{2} + \frac{m_2\left(l_1^2\dot{\theta}_1^2 + 4l_1\cos(\theta_1)\dot{\theta}_1\dot{x} + 4\dot{x}^2\right)}{8} + \frac{m_3 \cdot \left(4l_1^2\dot{\theta}_1^2 + 4l_1 l_2\left(\dot{\theta}_1 + \dot{\theta}_2\right)\cos(\theta_2)\dot{\theta}_1 + 8l_1\cos(\theta_1)\dot{\theta}_1\dot{x} + l_2^2\left(\dot{\theta}_1 + \dot{\theta}_2\right)^2 \right.}{8}$$

0.0.12 Deriving the Euler-Lagrange equations

[27]:
```
q = [x, theta1, theta2]

l = me.LagrangesMethod(L, q)

le = l.form_lagranges_equations()

le.simplify(); le
```

[27]:

$$\left[\begin{array}{r} m_1\ddot{x} + \frac{m_2\left(-l_1\sin(\theta_1)\dot{\theta}_1^2 + l_1\cos(\theta_1)\ddot{\theta}_1 + 2\ddot{x}\right)}{2} - \frac{m_3\cdot\left(2l_1\sin(\theta_1)\dot{\theta}_1^2 - \right.}{} \\ I_2\ddot{\theta}_1 + I_3\ddot{\theta}_1 + I_3\ddot{\theta}_2 + \frac{gl_1 m_2\sin(\theta_1)}{2} + gl_1 m_3\sin(\theta_1) + \frac{gl_2 m_3\sin(\theta_1+\theta_2)}{2} + \frac{l_1^2 m_2\ddot{\theta}_1}{4} + l_1^2 m_3\ddot{\theta}_1 - l_1 l_2 m_3\sin(\theta_2)\dot{\theta}_1\dot{\theta}_2 - \frac{l_1 l_2 m}{} \\ I_3\ddot{\theta}_1 + I_3\ddot{\theta}_2 + \frac{gl_2 m_3\sin(\theta_1+\theta_2)}{2} + \frac{l_1 l_2 m_3\sin(\theta_2)\dot{\theta}_1^2}{2} + \frac{l_1 l}{} \end{array}\right.$$

Individual equations

[28]:
```
le[0]
```

[28]:

$$m_1\ddot{x}+\frac{m_2\left(-l_1\sin(\theta_1)\dot{\theta}_1^2+l_1\cos(\theta_1)\ddot{\theta}_1+2\ddot{x}\right)}{2}-\frac{m_3\cdot\left(2l_1\sin(\theta_1)\dot{\theta}_1^2-2l_1\cos(\theta_1)\ddot{\theta}_1+l_2\left(\dot{\theta}_1+\dot{\theta}_2\right)^2\sin(\theta_1+\theta_2)-l_2}{2}$$

[29]:
```
le[1]
```

[29]:

$$I_2\ddot{\theta}_1+I_3\ddot{\theta}_1+I_3\ddot{\theta}_2+\frac{gl_1m_2\sin(\theta_1)}{2}+gl_1m_3\sin(\theta_1)+\frac{gl_2m_3\sin(\theta_1+\theta_2)}{2}+\frac{l_1^2m_2\ddot{\theta}_1}{4}+l_1^2m_3\ddot{\theta}_1-l_1l_2m_3\sin(\theta_2)\dot{\theta}_1\dot{\theta}_2-\frac{l_1l_2m_3\sin(\theta_2)\dot{\theta}_2^2}{2}+l_1l_2m_3\cos(\theta_2)\ddot{\theta}_1+\frac{l_1l_2m_3\cos(\theta_2)\ddot{\theta}_2}{2}+\frac{l_1m_2\cos(\theta_1)\ddot{x}}{2}+l_1m_3\cos(\theta_1)\ddot{x}+\frac{l_2^2m_3\ddot{\theta}_1}{4}+\frac{l_2^2m_3\ddot{\theta}_2}{4}+\frac{l_2m_3\cos(\theta_1+\theta_2)\ddot{x}}{2}$$

[30]:
```
le[2]
```

[30]:

$$I_3\ddot{\theta}_1+I_3\ddot{\theta}_2+\frac{gl_2m_3\sin(\theta_1+\theta_2)}{2}+\frac{l_1l_2m_3\sin(\theta_2)\dot{\theta}_1^2}{2}+\frac{l_1l_2m_3\cos(\theta_2)\ddot{\theta}_1}{2}+\frac{l_2^2m_3\ddot{\theta}_1}{4}+\frac{l_2^2m_3\ddot{\theta}_2}{4}+\frac{l_2m_3\cos(\theta_1+\theta_2)\ddot{x}}{2}$$

0.0.13 Equations in $AX = B$ form

Generalized mass matrix

[31]:
```
Md = l.mass_matrix
Md

Md.simplify()
Md
```

[31]:

$$\left[\begin{matrix}m_1+m_2+m_3 & \frac{l_1m_2\cos(\theta_1)}{2}+l_1m_3\cos(\theta_1)+\frac{l_2m_3\cos(\theta_1+\theta_2)}{2} & \frac{l_2m_3\cos(\theta_1+\theta_2)}{2}\\ \frac{l_1m_2\cos(\theta_1)}{2}+l_1m_3\cos(\theta_1)+\frac{l_2m_3\cos(\theta_1+\theta_2)}{2} & I_2+I_3+\frac{l_1^2m_2}{4}+l_1^2m_3+l_1l_2m_3\cos(\theta_2)+\frac{l_2^2m_3}{4} & I_3+\frac{l_2m_3\cdot(2l_1\cos(\theta_2)+l}{4}\\ \frac{l_2m_3\cos(\theta_1+\theta_2)}{2} & I_3+\frac{l_2m_3\cdot(2l_1\cos(\theta_2)+l_2)}{4} & I_3+\frac{l_2^2m_3}{4}\end{matrix}\right.$$

Generalized force vector

[32]:
```
fd = l.forcing
fd

fd.simplify()
fd
```

[32]:

$$\left[\begin{matrix}\frac{l_1m_2\sin(\theta_1)\dot{\theta}_1^2}{2}+\frac{m_3\cdot\left(2l_1\sin(\theta_1)\dot{\theta}_1^2+l_2\left(\dot{\theta}_1+\dot{\theta}_2\right)^2\sin(\theta_1+\theta_2)\right)}{2}\\ -\frac{gl_1m_2\sin(\theta_1)}{2}-gl_1m_3\sin(\theta_1)-\frac{gl_2m_3\sin(\theta_1+\theta_2)}{2}+l_1l_2m_3\sin(\theta_2)\dot{\theta}_1\dot{\theta}_2+\frac{l_1l_2m_3\sin(\theta_2)\dot{\theta}_2^2}{2}\\ -\frac{l_2m_3\left(g\sin(\theta_1+\theta_2)+l_1\sin(\theta_2)\dot{\theta}_1^2\right)}{2}\end{matrix}\right]$$

1 Simulations

1.0.1 Setup

[33]:
```
import numpy as np
```

[34]:
```
t = me.dynamicsymbols._t
```

[35]:
```
l.q
```

[35]:

$$\begin{bmatrix} x \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

[36]:
```
l.u
```

[36]:

$$\begin{bmatrix} \dot{x} \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix}$$

[37]:
```
le.free_symbols
```

[37]:

$$\{I_2, I_3, g, l_1, l_2, m_1, m_2, m_3, t\}$$

System parameters

[38]:
```
p = sm.Matrix([g,l1,l2,m1,m2,m3,I2,I3])
p
```

[38]:

$$\begin{bmatrix} g \\ l_1 \\ l_2 \\ m_1 \\ m_2 \\ m_3 \\ I_2 \\ I_3 \end{bmatrix}$$

Generalized coordinates

[39]:
```
q = l.q
q
```

[39]:

$$\begin{bmatrix} x \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

Generalized speed

[40]:
```
u = l.u
u
```

[40]:

$$\begin{bmatrix}\dot{x}\\\dot{\theta}_1\\\dot{\theta}_2\end{bmatrix}$$

2 Simulation

[41]:

```
eval_eom = sm.lambdify((q, u, p), [Md, fd])
```

Initial conditions

[42]:

```
q_vals = np.array([
    0.0,
    np.deg2rad(30.0),   # q1, rad
    np.deg2rad(-60.0),   # q2, rad
])
u_vals = np.array([
    0,   # u1, rad/s
    0,   # u2, rad/s
    0,   # u3, m/s
])
p_vals = np.array([
    9.81,    # g
    1,       # l1
    0.5,     # l2
    2,       # m1
    1,       # m2
    1,       # m3
    (1/12)*1*1**2,   # I2
    (1/12)*1*0.5**2, # I3
])
```

[43]:

```
Md_vals, gd_vals = eval_eom(q_vals, u_vals, p_vals)
Md_vals, gd_vals
```

[43]:

```
(array([[4.        , 1.51554446, 0.21650635],
        [1.51554446, 1.66666667, 0.20833333],
        [0.21650635, 0.20833333, 0.08333333]]),
 array([[ 0.     ],
        [-6.13125],
        [ 1.22625]]))
```

[44]:

```
ud_vals = np.linalg.solve(Md_vals, np.squeeze(gd_vals))
ud_vals
```

[44]:

```
array([ 1.77610646, -9.53655052, 33.94191638])
```

Integration Algorithm Setup

[45]:
```python
def rk4_integrate(rhs_func, tspan, x0_vals, p_vals, delt=0.01):
    """Returns state trajectory and corresponding values of time resulting
    from integrating the ordinary differential equations with Euler's
    Method.

    Parameters
    ==========
    rhs_func : function
       Python function that evaluates the derivative of the state and takes
       this form ``dxdt = f(t, x, p)``.
    tspan : 2-tuple of floats
       The initial time and final time values: (t0, tf).
    x0_vals : array_like, shape(2*n,)
       Values of the state x at t0.
    p_vals : array_like, shape(o,)
       Values of constant parameters.
    delt : float
       Integration time step in seconds/step.

    Returns
    =======
    ts : ndarray(m, )
       Monotonically increasing values of time.
    xs : ndarray(m, 2*n)
       State values at each time in ts.

    """
    # generate monotonically increasing values of time.
    duration = tspan[1] - tspan[0]
    num_samples = round(duration/delt) + 1
    ts = np.arange(tspan[0], tspan[0] + delt*num_samples, delt)

    # create an empty array to hold the state values.
    x = np.empty((len(ts), len(x0_vals)))

    # set the initial conditions to the first element.
    x[0, :] = x0_vals

    # use a for loop to sequentially calculate each new x.
    for i, ti in enumerate(ts[:-1]):

        # step 1
        tstep = ti
        xstep = x[i, :]
        k1 = rhs_func(tstep, xstep, p_vals)

        # step 2
```

```python
        tstep = ti + delt/2.0
        xstep = x[i, :] + (delt/2.0)*k1
        k2 = rhs_func(tstep, xstep, p_vals)

        # step 3
        tstep = ti + delt/2.0
        xstep = x[i, :] + (delt/2.0)*k2
        k3 = rhs_func(tstep, xstep, p_vals)

        # step 4
        tstep = ti + delt
        xstep = x[i, :] + (delt)*k3
        k4 = rhs_func(tstep, xstep, p_vals)

        x[i + 1, :] = x[i, :] + (delt/6.0)*(k1 + 2.0*k2 + 2.0*k3 + k4)

    return ts, x
```

[46]:
```python
def eval_derivative(t, x, p):
    """Return the right hand side of the explicit ordinary differential
    equations which evaluates the time derivative of the state ``x`` at time
    ``t``.

    Parameters
    ==========
    t : float
       Time in seconds.
    x : array_like, shape(6,)
       State at time t: [q1, q2, q3, u1, u2, u3]
    p : array_like, shape(5,)
       Constant parameters: [g, kl, kt, l, m]

    Returns
    =======
    xd : ndarray, shape(6,)
        Derivative of the state with respect to time at time ``t``.

    """

    # unpack the q and u vectors from x
    q = x[:3]
    u = x[3:]

    # evaluate the equations of motion matrices with the values of q, u, p
    Md, gd = eval_eom(q, u, p)

    # solve for q' and u'
```

```
    qd = u
    ud = np.linalg.solve(Md, np.squeeze(gd))

    # pack dq/dt and du/dt into a new state time derivative vector dx/dt
    xd = np.empty_like(x)
    xd[:3] = qd
    xd[3:] = ud

    return xd
```

[47]:
```
x0 = np.empty(6)
x0[:3] = q_vals
x0[3:] = u_vals
```

Running the Integration

[48]:
```
t0 = 0.0
tf = 30.0
dt = 0.001
ts1, states1 = rk4_integrate(eval_derivative, (t0, tf), x0, p_vals, dt)
```

[49]:
```
ts = ts1[::20]
states = states1[::20,:]

ts
```

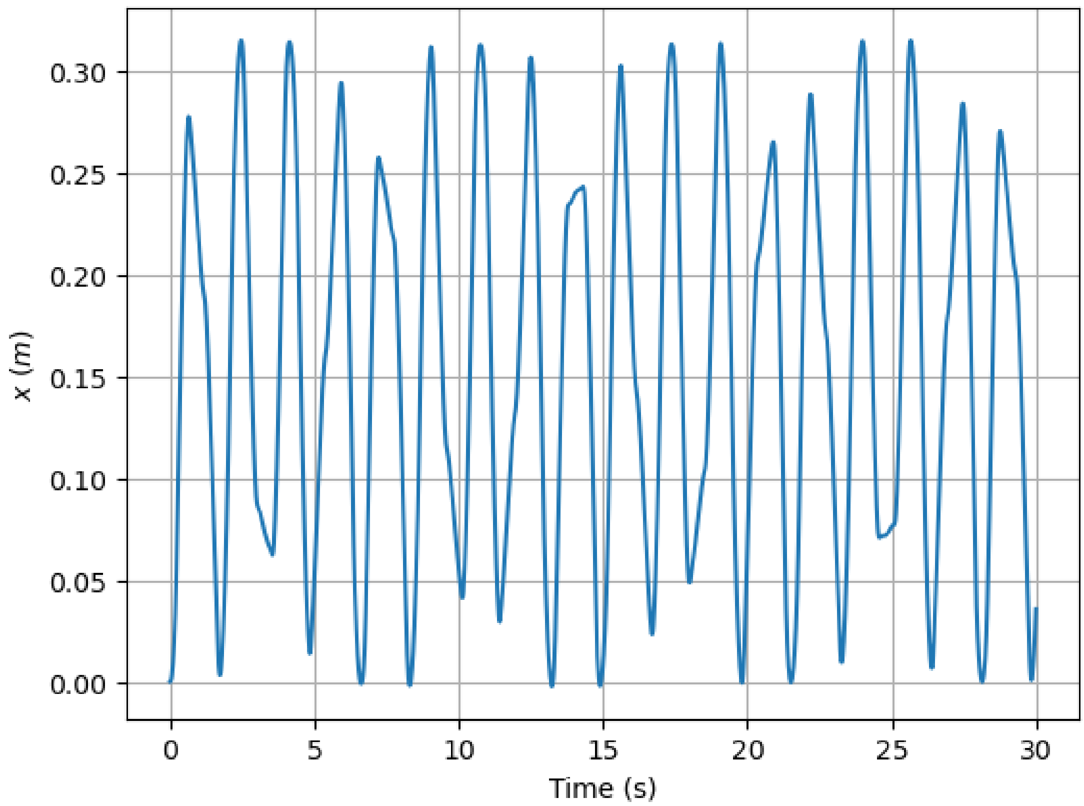
[49]:
```
array([0.000e+00, 2.000e-02, 4.000e-02, …, 2.996e+01, 2.998e+01,
       3.000e+01])
```

3 Plotting results

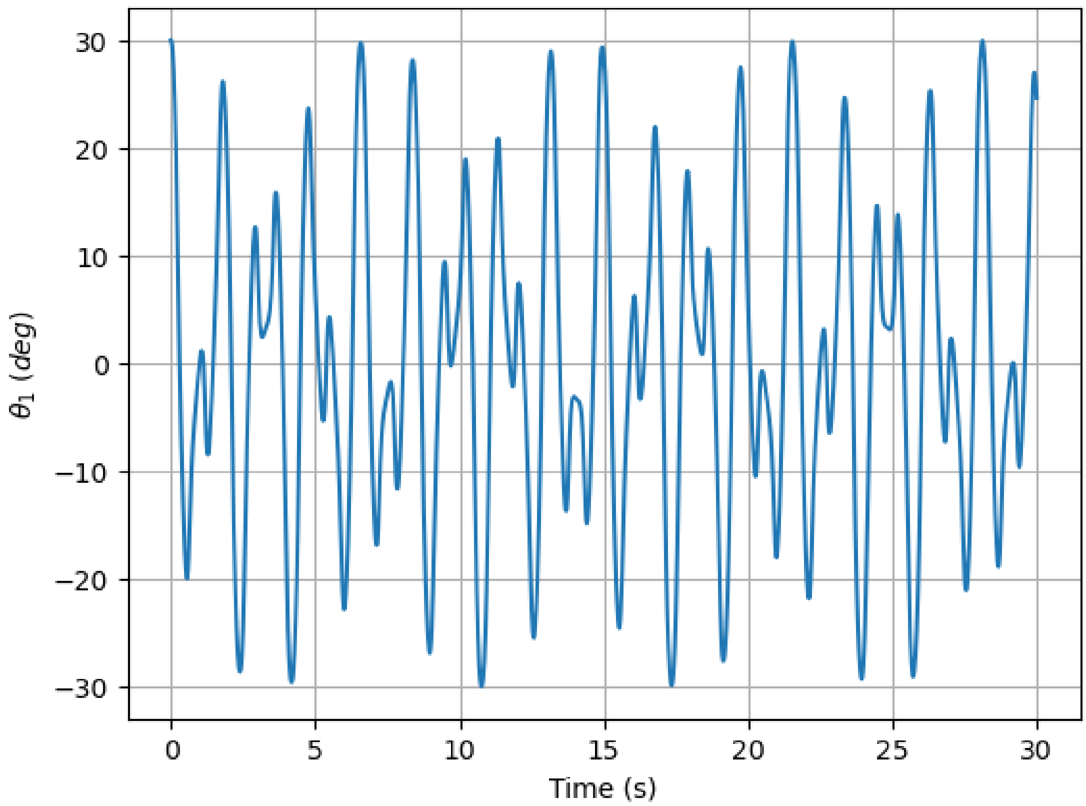
[50]:
```
import matplotlib.pyplot as plt
```

[51]:
```
xs = states[:,0]
theta1s = np.rad2deg(states[:,1])
theta2s = np.rad2deg(states[:,2])
xds = states[:,3]
theta1ds = np.rad2deg(states[:,4])
theta2ds = np.rad2deg(states[:,5])
```

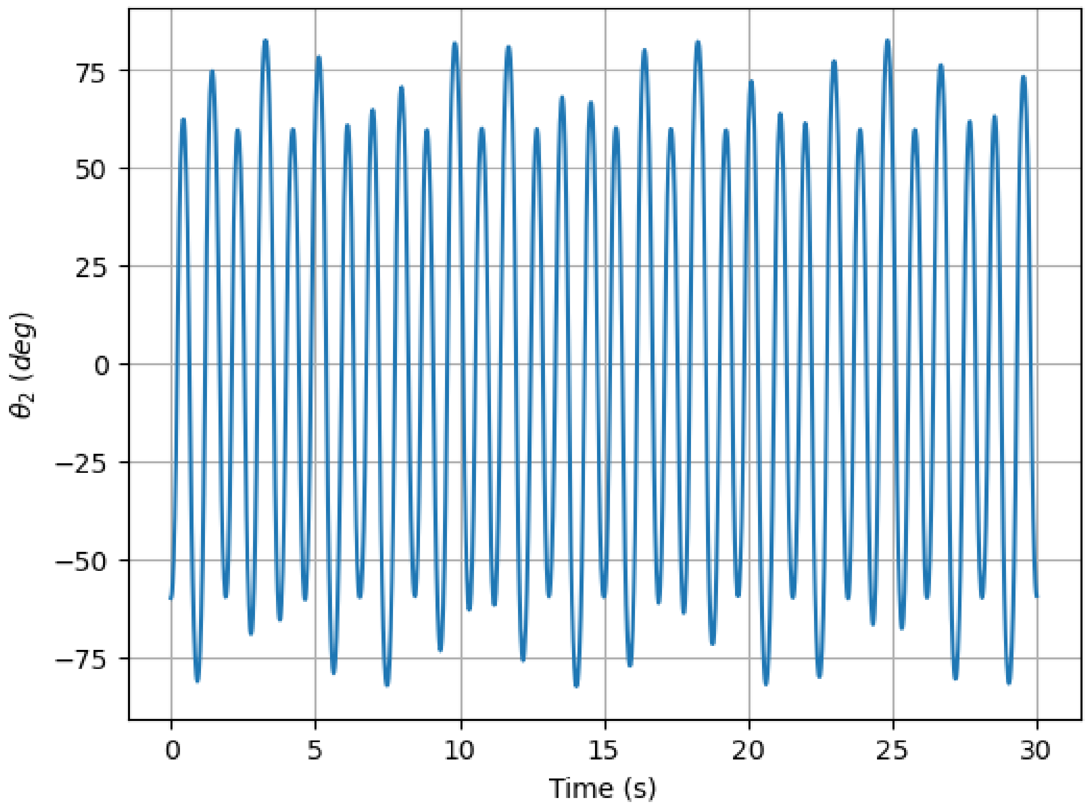
[52]:
```
plt.plot(ts,xs);
plt.xlabel("Time (s)")
plt.ylabel("$x~(m)$")
plt.grid()
```



[53]:
```
plt.plot(ts,theta1s);
plt.xlabel("Time (s)")
plt.ylabel("$\\theta_1~(deg)$")
plt.grid()
```



[54]:
```
plt.plot(ts,theta2s);
plt.xlabel("Time (s)")
plt.ylabel("$\\theta_2~(deg)$")
plt.grid()
```



4 Validation

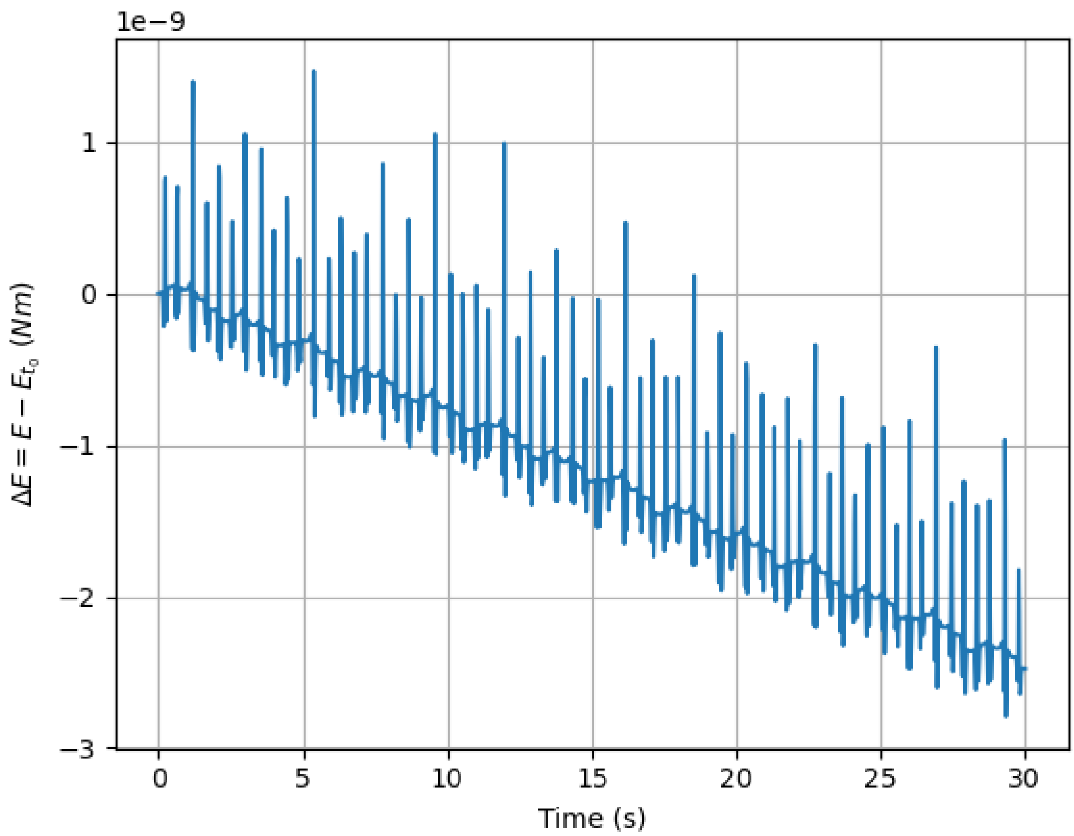
4.0.1 Verifying conservation of mechanical energy

[55]:
```
eval_me = sm.lambdify((q, u, p), [KE, PE])
```

[56]:
```
kes = np.zeros((len(ts), len([1])))
pes = np.zeros((len(ts), len([1])))
tes = np.zeros((len(ts), len([1])))

for i, ti in enumerate(ts[:]):
    q_vals = np.array([
        xs[i],
        np.deg2rad(theta1s[i]),
        np.deg2rad(theta2s[i]),
    ])

    u_vals = np.array([
        xds[i],
        np.deg2rad(theta1ds[i]),
```

```
        np.deg2rad(theta2ds[i]),
    ])

    kes[i], pes[i] = eval_me(q_vals, u_vals, p_vals)
    tes[i] = kes[i] + pes[i]
```

[57]:
```
plt.plot(ts, tes - tes[0]);
plt.xlabel("Time (s)")
plt.ylabel("$\\Delta E = E - E_{t_0}~(Nm)$")
plt.grid()
```



5 Animation

[58]:
```
import matplotlib.animation as animation
```

Joint positions

[59]:
```
# Joints
J1 = O_B1.locatenew('J1', 0*B1.x)
```

```python
J1.set_vel(B1, 0)

J2 = O_B2.locatenew('J2', -(l1/2)*B2.y)
J2.set_vel(B2, 0)

J3 = O_B3.locatenew('J3', -(l2/2)*B3.y)
J3.set_vel(B3, 0)
```

[60]:
```python
j1_pos = J1.pos_from(O_N).to_matrix(N)
j1_pos
```

[60]:

$$\begin{bmatrix} x \\ 0 \\ 0 \end{bmatrix}$$

[61]:
```python
j2_pos = J2.pos_from(O_N).to_matrix(N)
j2_pos
```

[61]:

$$\begin{bmatrix} l_1 \sin(\theta_1) + x \\ -l_1 \cos(\theta_1) \\ 0 \end{bmatrix}$$

[62]:
```python
j3_pos = J3.pos_from(O_N).to_matrix(N)
j3_pos.simplify()
```

[62]:

$$\begin{bmatrix} l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) + x \\ -l_1 \cos(\theta_1) - l_2 \cos(\theta_1 + \theta_2) \\ 0 \end{bmatrix}$$

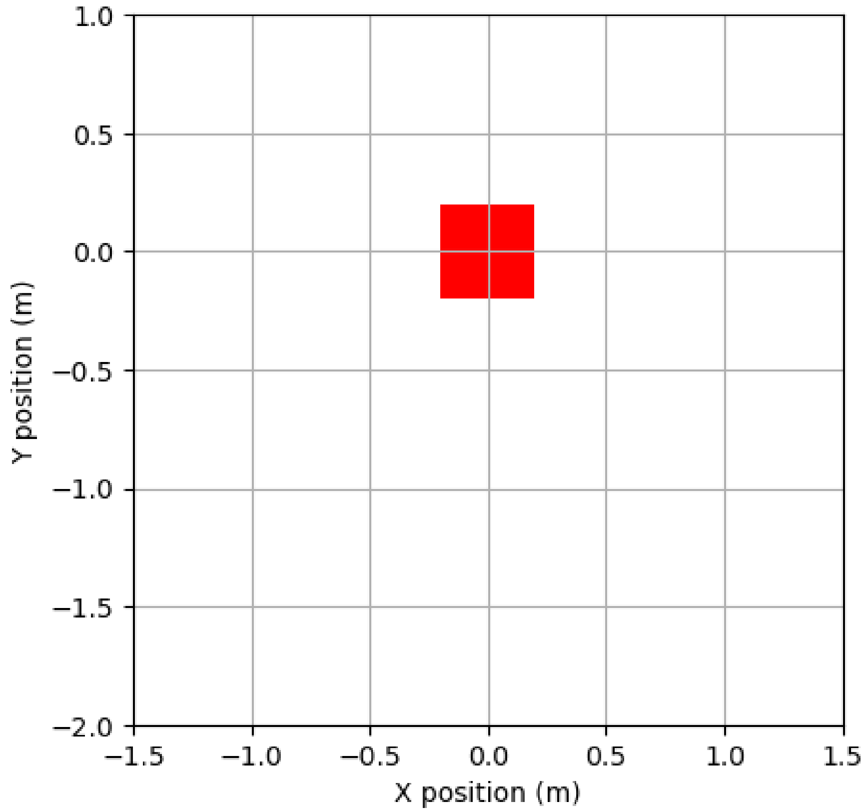
[63]:
```python
eval_j1 = sm.lambdify((q, u, p), [j1_pos])
eval_j2 = sm.lambdify((q, u, p), [j2_pos])
eval_j3 = sm.lambdify((q, u, p), [j3_pos])
```

[64]:
```python
j1x = np.zeros(len(ts))
j1y = np.zeros(len(ts))
j2x = np.zeros(len(ts))
j2y = np.zeros(len(ts))
j3x = np.zeros(len(ts))
j3y = np.zeros(len(ts))


for i, ti in enumerate(ts[:]):
    q_vals = np.array([
        xs[i],
        np.deg2rad(theta1s[i]),
        np.deg2rad(theta2s[i]),
    ])
```

```
    u_vals = np.array([
        xds[i],
        np.deg2rad(theta1ds[i]),
        np.deg2rad(theta2ds[i]),
    ])

    j1_poss = eval_j1(q_vals, u_vals, p_vals)
    j2_poss = eval_j2(q_vals, u_vals, p_vals)
    j3_poss = eval_j3(q_vals, u_vals, p_vals)

    j1x[i] = j1_poss[0][0][0]
    j1y[i] = j1_poss[0][1][0]

    j2x[i] = j2_poss[0][0][0]
    j2y[i] = j2_poss[0][1][0]

    j3x[i] = j3_poss[0][0][0]
    j3y[i] = j3_poss[0][1][0]
```

Setting up animation

[65]:
```
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-1.5, 1.5), ylim=(-2, 1),
                     xlabel="X position (m)", ylabel="Y position (m)")
ax.set_aspect('equal')
ax.grid()

line, = ax.plot([], [], 'o-', lw=4)
# line2, = ax.plot([], [], '*', lw=1)

rectangle = plt.Rectangle((-0.2, -0.2), 0.4, 0.4, fc='r')
plt.gca().add_patch(rectangle)

time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
```



[66]:
```
def init():
    line.set_data([], [])
#     line2.set_data([], [])

    rectangle.set_x([])
    rectangle.set_y([])
    time_text.set_text('')
    return line, time_text
```

[67]:
```
def animate(i):
    thisx = [j1x[i], j2x[i], j3x[i]]
    thisy = [j1y[i], j2y[i], j3y[i]]

    line.set_data(thisx, thisy)
    rectangle.set_x(j1x[i]-0.2)
    rectangle.set_y(j1y[i]-0.2)

#     thisx = [0, j1x[i]]
#     thisy = [0, j1y[i]]

#     line2.set_data(thisx, thisy)
```

```python
    time_text.set_text(time_template % (i*0.02))
    return line, time_text
```

[68]:
```python
ani = animation.FuncAnimation(fig, animate, np.arange(1, len(ts)),
                              interval=25, blit=True, init_func=init)

ani.save('moving_double_pendulum.mp4', fps=50) #'__double_pendulum.mp4', fps=15)
#plt.show()
```

[ ]:
```python

```