jack_in_box_finalProject

June 10, 2021

0.0.1 import libraries

[1]:
```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
from sympy.abc import t
from math import sqrt
```

0.0.2 main set up of constants and functions

[2]:
```
#constants
dim1 = 5 #dim 1 of box
dim2 = dim1 #dim 2 of box (make it square)
lj = 1 #length of jack
mj = 1 #mass of the bulb at each end of jack
mb = 25 #give the walls of the box some large mass relative to jack
g = 9.8
```

[3]:
```
#set up configuration variables for jack and box
#these will be used for deriving KE and PE equations for each component

#config variabls for jack
xj = sym.Function(r'x_j')(t)
yj = sym.Function(r'y_j')(t)
thj = sym.Function(r'\theta_j')(t)
#config vriables for box
xb = sym.Function(r'x_b')(t)
yb = sym.Function(r'y_b')(t)
thb = sym.Function(r'\theta_b')(t)

#consolidate config variables into array q
q = sym.Matrix([xj, yj, thj, xb, yb, thb])
qdot = q.diff(t)
qddot = qdot.diff(t)
```

0.0.3 functions to handle frame transformations

[4]:
```
#rotate a 2x2 matrix
def rot(th):
    return sym.Matrix([[sym.cos(th), -sym.sin(th)], [sym.sin(th), sym.cos(th)]])

#generate g transformation matrix (R is rotation component, t is translation␣
↪component)
def transform_frame(rot, trans):
    g = sym.Matrix([
        [rot[0], rot[1], 0, trans[0]],
        [rot[2], rot[3], 0, trans[1]],
        [0, 0, 1, trans[2]],
        [0, 0, 0, 1]
    ])
    return g


#compute inverse of transformation matrix
def inverse_mat(mat):

    #extract rotation elements from input transform matrix mat
    R = sym.Matrix([
        [mat[0,0], mat[0,1], mat[0,2]],
        [mat[1,0], mat[1,1], mat[1,2]],
        [mat[2,0], mat[2,1], mat[2,2]]
    ])

    #we know it's a square matrix
    R_inv = R.T

    #extract translational elements from input transform matrix mat
    p = sym.Matrix([
        mat[0,3],
        mat[1,3],
        mat[2,3]
    ])

    p_inv = -R_inv*p

    #now compute inverse matrix
    output = sym.Matrix([
        [R_inv[0,0], R_inv[0,1], R_inv[0,2], p_inv[0]],
        [R_inv[1,0], R_inv[1,1], R_inv[1,2], p_inv[1]],
        [R_inv[2,0], R_inv[2,1], R_inv[2,2], p_inv[2]],
        [0, 0, 0, 1]
    ])
```

```
    return output
```

0.0.4 set up transformation matrices g__xx

[5]:
```
#frame transformations

#world to jack
gwj = transform_frame(rot(q[2]), sym.Matrix([q[0], q[1], 0])) #rotation from␣
↪world to jack frame
gjb = transform_frame(rot(0), sym.Matrix([0, -lj/2, 0]))
gjt = transform_frame(rot(0), sym.Matrix([0, lj/2, 0]))
gjl = transform_frame(rot(0), sym.Matrix([-lj/2, 0, 0]))
gjr = transform_frame(rot(0), sym.Matrix([lj/2, 0, 0]))

#jack to box
gwj_b = transform_frame(rot(q[5]), sym.Matrix([q[3], q[4], 0])) #rotation from␣
↪jack to box frame
gjb_b = transform_frame(rot(0), sym.Matrix([0, -dim2/2, 0]))
gjb_t = transform_frame(rot(0), sym.Matrix([0, dim2/2, 0]))
gjb_l = transform_frame(rot(0), sym.Matrix([-dim1/2, 0, 0]))
gjb_r = transform_frame(rot(0), sym.Matrix([dim1/2, 0, 0]))

#transform from world frame to end points of jack
gwb = gwj*gjb
gwt = gwj*gjt
gwl = gwj*gjl
gwr = gwj*gjr

#transform from jack center to wall points of box
gwb_b = gwj_b*gjb_b
gwb_t = gwj_b*gjb_t
gwb_l = gwj_b*gjb_l
gwb_r = gwj_b*gjb_r

#now transform from jack end points to center of box
Gjj_b = inverse_mat(gwj)*gwj_b
Gjb_b = inverse_mat(gwb)*gwj_b
Gjt_b = inverse_mat(gwt)*gwj_b
Gjl_b = inverse_mat(gwl)*gwj_b
Gjr_b = inverse_mat(gwr)*gwj_b
```

[6]:
```
#transform world-to-jack endpoint frames to each box walls
#jack bottom end
Gbb_b = inverse_mat(gwb) * gwb_b #world to jack bottom pt, to box bottom wall
Gbt_b = inverse_mat(gwb) * gwb_t #world to jack bottom pt, to box top wall
Gbl_b = inverse_mat(gwb) * gwb_l #world to jack bottom pt, to box left wall
```

```
Gbr_b = inverse_mat(gwb) * gwb_r #world to jack bottom pt, to box right wall

#jack top end
Gtb_b = inverse_mat(gwt) * gwb_b #world to jack top pt, to box bottom wall
Gtt_b = inverse_mat(gwt) * gwb_t #world to jack top pt, to box top wall
Gtl_b = inverse_mat(gwt) * gwb_l #world to jack top pt, to box left wall
Gtr_b = inverse_mat(gwt) * gwb_r #world to jack top pt, to box right wall

#jack left end
Glb_b = inverse_mat(gwl) * gwb_b #world to jack left pt, to box bottom wall
Glt_b = inverse_mat(gwl) * gwb_t #world to jack left pt, to top bottom wall
Gll_b = inverse_mat(gwl) * gwb_l #world to jack left pt, to left bottom wall
Glr_b = inverse_mat(gwl) * gwb_r #world to jack left pt, to right bottom wall

#jack right end
Grb_b = inverse_mat(gwr) * gwb_b #world to jack right pt, to box bottom wall
Grt_b = inverse_mat(gwr) * gwb_t #world to jack right pt, to top bottom wall
Grl_b = inverse_mat(gwr) * gwb_l #world to jack right pt, to left bottom wall
Grr_b = inverse_mat(gwr) * gwb_r #world to jack right pt, to right bottom wall
```

0.0.5 compute moment of inertia and body velocities to set up for KE and PE

[7]:
```
def unhat(mat):
    return sym.Matrix([
        mat[0,3], mat[1,3], mat[2,3], mat[2,1], -mat[2,0], mat[1,0]
    ])

#Mass config
mb_tot = 4*mb
mj_tot = 4*mj
mb_dist = sqrt(2)*dim1/2
mj_dist = sqrt(2)*dim2/2
Jb = mb_tot*(dim1/2)**2
Jj = mj_tot*(dim2/2)**2

#Mass-Inertia
Ij = sym.Matrix([
        [mj_tot,0.,0.,0.,0.,0.],
        [0.,mj_tot,0.,0.,0.,0.],
        [0.,0.,mj_tot,0.,0.,0.],
        [0.,0.,0.,0.,0.,0.],
        [0.,0.,0.,0.,0.,0.],
        [0.,0.,0.,0.,0.,Jj]
    ])

Ij2 = sym.Matrix([
        [0.,0.,0.],
```

```
        [0.,0.,0.],
        [0.,0.,Jj]
    ])

Ib = sym.Matrix([
        [mb_tot,0.,0.,0.,0.,0.],
        [0.,mb_tot,0.,0.,0.,0.],
        [0.,0.,mb_tot,0.,0.,0.],
        [0.,0.,0.,0.,0.,0.],
        [0.,0.,0.,0.,0.,0.],
        [0.,0.,0.,0.,0.,Jb]
    ])

Ib2 = sym.Matrix([
        [0.,0.,0.],
        [0.,0.,0.],
        [0.,0.,Jb]
    ])

#compute body velocity
Vj = inverse_mat(gwj)*gwj.diff(t)
Vj = unhat(Vj)
Vb = inverse_mat(gwj_b)*gwj_b.diff(t)
Vb = unhat(Vb)
```

0.0.6 compute Euler-Lagrangian equation

[91]:
```
#kinetic energy of system
KEj = ((Vj.T * Ij * Vj)/2)[0] #KE for jack
KEb = ((Vb.T * Ib * Vb)/2)[0] #KE for box
KE = KEj + KEb

#potential energy
PE = 0 #assume jack is on table top, there is no potential energy due to gravity

#Lagrangian
L = sym.simplify(KE- PE)

dldq = L.diff(q)
dldqdot = L.diff(qdot)
ddt_dldqdot = dldqdot.diff(t)

#assume force input only applies to outside of box in XY directions
fx = 500*sym.sin(t)
fy = 0 #-500*sym.cos(t)
f = (sym.Matrix([0,0,0,fx,fy,0]))
```

```
#Euler-Lagrangian equations
el = sym.Eq(ddt_dldqdot - dldq, f)
```

[92]:
```
print('==================================================')
print()
print("Euler Lagrange equations for box and jack with input force on box")
display((el))
```

```
==================================================

Euler Lagrange equations for box and jack with input force on box
```

$$\left[\begin{matrix}4.0\frac{d^2}{dt^2}\operatorname{x_j}(t)\\4.0\frac{d^2}{dt^2}\operatorname{y_j}(t)\\25.0\frac{d^2}{dt^2}\theta_j(t)\\100.0\frac{d^2}{dt^2}\operatorname{x_b}(t)\\100.0\frac{d^2}{dt^2}\operatorname{y_b}(t)\\625.0\frac{d^2}{dt^2}\theta_b(t)\end{matrix}\right] = \left[\begin{matrix}0\\0\\0\\500\sin(t)\\0\\0\end{matrix}\right]$$

1 Now need to solve for trajectory simulation

1.0.1 solve and lambdify

[93]:
```
el_soln = sym.solve(el, [*qddot])
```

[94]:
```
print('==================================================')
print()
print("Equations of motion:")

for key in el_soln:
    display(sym.Eq(key, el_soln[key]))
```

```
==================================================

Equations of motion:
```

$$\frac{d^2}{dt^2}\operatorname{x_j}(t) = 0.0$$

$$\frac{d^2}{dt^2}\operatorname{y_j}(t) = 0.0$$

$$\frac{d^2}{dt^2}\theta_j(t) = 0.0$$

$$\frac{d^2}{dt^2}\operatorname{x_b}(t) = 5.0\sin(t)$$

$$\frac{d^2}{dt^2}\operatorname{y_b}(t) = 0.0$$

$$\frac{d^2}{dt^2}\theta_b(t) = 0.0$$

[95]:
```
#lambdify
xjfunc = sym.lambdify([*q, *qdot, t], el_soln[qddot[0]])
yjfunc = sym.lambdify([*q, *qdot, t], el_soln[qddot[1]])
thjfunc = sym.lambdify([*q, *qdot, t], el_soln[qddot[2]])
xbfunc = sym.lambdify([*q, *qdot, t], el_soln[qddot[3]])
ybfunc = sym.lambdify([*q, *qdot, t], el_soln[qddot[4]])
thbfunc = sym.lambdify([*q, *qdot, t], el_soln[qddot[5]])
```

1.0.2 prep for computing Hamiltonian

[96]:
```
#dummy variables
xjminus, yjminus, thjminus = sym.symbols(r'{x_j}^{-},{y_j}^{-},{\theta_j}^{-}')
xbminus, ybminus, thbminus = sym.symbols(r'{x_b}^{-},{y_b}^{-},{\theta_b}^{-}')
xjdotplus,yjdotplus,thjdotplus,xjdotminus,yjdotminus,thjdotminus = sym.
 ↪symbols(r'\dot{x_j}^{+},\dot{y_j}^{+},\dot{\theta_j}^{+},\dot{x_j}^{-},\dot{y_j}^{-},\dot{\
xbdotplus,ybdotplus,thbdotplus,xbdotminus,ybdotminus,thbdotminus = sym.
 ↪symbols(r'\dot{x_b}^{+},\dot{y_b}^{+},\dot{\theta_b}^{+},\dot{x_b}^{-},\dot{y_b}^{-},\dot{\
lam = sym.symbols(r'\lambda')

#substitution dictionaries
sub_minus = {
    q[0]:xjminus,
    q[1]:yjminus,
    q[2]:thjminus,
    q[3]:xbminus,
    q[4]:ybminus,
    q[5]:thbminus,
    qdot[0]:xjdotminus,
    qdot[1]:yjdotminus,
    qdot[2]:thjdotminus,
    qdot[3]:xbdotminus,
    qdot[4]:ybdotminus,
    qdot[5]:thbdotminus}

sub_plus = {
    q[0]:xjminus,
    q[1]:yjminus,
    q[2]:thjminus,
    q[3]:xbminus,
    q[4]:ybminus,
    q[5]:thbminus,
    qdot[0]:xjdotplus,
    qdot[1]:yjdotplus,
    qdot[2]:thjdotplus,
```

```
    qdot[3]:xbdotplus,
    qdot[4]:ybdotplus,
    qdot[5]:thbdotplus}
```

1.0.3 compute Hamiltonian

[97]:
```
#Lagrangian in matrix form needed for Hamiltonian step
l_mat = sym.Matrix([L])

#compute Hamiltonian. needed for evaluating impacts
H = dldqdot.T * qdot - l_mat
H_minus = H.subs(sub_minus)
H_plus = H.subs(sub_plus)
```

1.0.4 set up for impact update equations

[98]:
```
#we have potentially 16 impact cases so a loop can be used to efficiently␣
↪generate the equations
#first set up the constraint condition phi for each of the 16 possible cases

#possible impact conditions of bottom jack endpoint to four walls of box
phi1 = sym.Matrix([sym.simplify((inverse_mat(Gbb_b)*sym.Matrix([0,0,0,1]))[1])])
phi2 = sym.Matrix([sym.simplify((inverse_mat(Gbt_b)*sym.Matrix([0,0,0,1]))[1])])
phi3 = sym.Matrix([sym.simplify((inverse_mat(Gbr_b)*sym.Matrix([0,0,0,1]))[0])])
phi4 = sym.Matrix([sym.simplify((inverse_mat(Gbl_b)*sym.Matrix([0,0,0,1]))[0])])

#possible impact conditions of top jack endpoint to four walls of box
phi5 = sym.Matrix([sym.simplify((inverse_mat(Gtb_b)*sym.Matrix([0,0,0,1]))[1])])
phi6 = sym.Matrix([sym.simplify((inverse_mat(Gtt_b)*sym.Matrix([0,0,0,1]))[1])])
phi7 = sym.Matrix([sym.simplify((inverse_mat(Gtr_b)*sym.Matrix([0,0,0,1]))[0])])
phi8 = sym.Matrix([sym.simplify((inverse_mat(Gtl_b)*sym.Matrix([0,0,0,1]))[0])])

#possible impact conditions of left jack endpoint to four walls of box
phi9 = sym.Matrix([sym.simplify((inverse_mat(Glb_b)*sym.Matrix([0,0,0,1]))[1])])
phi10 = sym.Matrix([sym.simplify((inverse_mat(Glt_b)*sym.
↪Matrix([0,0,0,1]))[1])])
phi11 = sym.Matrix([sym.simplify((inverse_mat(Glr_b)*sym.
↪Matrix([0,0,0,1]))[0])])
phi12 = sym.Matrix([sym.simplify((inverse_mat(Gll_b)*sym.
↪Matrix([0,0,0,1]))[0])])

#possible impact conditions of right jack endpoint to four walls of box
phi13 = sym.Matrix([sym.simplify((inverse_mat(Grb_b)*sym.
↪Matrix([0,0,0,1]))[1])])
phi14 = sym.Matrix([sym.simplify((inverse_mat(Grt_b)*sym.
↪Matrix([0,0,0,1]))[1])])
```

```
phi15 = sym.Matrix([sym.simplify((inverse_mat(Grr_b)*sym.
↪Matrix([0,0,0,1]))[0])])
phi16 = sym.Matrix([sym.simplify((inverse_mat(Grl_b)*sym.
↪Matrix([0,0,0,1]))[0])])

#contain all phi constraints in one list to be accessed by for loop
phi_mat = sym.
↪Matrix([phi1,phi2,phi3,phi4,phi5,phi6,phi7,phi8,phi9,phi10,phi11,phi12,phi13,phi14,phi15,ph
phi_mat_minus = phi_mat.subs(sub_minus)
phi_list =␣
↪[phi1,phi2,phi3,phi4,phi5,phi6,phi7,phi8,phi9,phi10,phi11,phi12,phi13,phi14,phi15,phi16]
```

1.0.5 generate 16 impact update equations and store in one list

[99]:
```
dldqdot_minus = dldqdot.subs(sub_minus)
dldqdot_plus = dldqdot.subs(sub_plus)

#will be used in generating each impact update eqn
lhs = sym.Matrix([dldqdot_plus-dldqdot_minus, H_plus-H_minus])
# display(lhs)

impacts = []
for i in range(len(phi_list)):
    dpdq = phi_list[i].jacobian(q).T
    dpdq_minus = dpdq.subs(sub_minus)

    rhs = sym.Matrix([lam * dpdq_minus, 0])
    impact = sym.Eq(lhs, rhs)
    impacts.append(impact)
```

[100]:
```
print('==================================================')
print()
print("print out of one impact update equation as an example:")
sym.simplify(impacts[1])
```

```
==================================================

print out of one impact update equation as an example:
```

[100]:

$$\begin{bmatrix} 1.0\lambda \sin\left(\theta_b{}^{-}\right) \\ -1.0\lambda \cos\left(\theta_b{}^{-}\right) \\ 0.5\lambda \sin\left(\theta_b{}^{-} - \theta_j{}^{-}\right) \\ -1.0\lambda \sin\left(\theta_b{}^{-}\right) \\ 1.0\lambda \cos\left(\theta_b{}^{-}\right) \\ -\lambda\left(1.0x_b{}^{-}\cos\left(\theta_b{}^{-}\right) - 1.0x_j{}^{-}\cos\left(\theta_b{}^{-}\right) + 1.0y_b{}^{-}\sin\left(\theta_b{}^{-}\right) - 1.0y_j{}^{-}\sin\left(\theta_b{}^{-}\right) + 0.5\sin\left(\theta_b{}^{-} - \theta_j{}^{-}\right)\right) \\ 0 \end{bmatrix} =$$

$$
\begin{bmatrix}
-4.0\dot{x}_j{}^{+} + 4.0\dot{x}_j{}^{-} \\
-4.0\dot{y}_j{}^{+} + 4.0\dot{y}_j{}^{-} \\
-25.0\dot{\theta}_j{}^{+} + 25.0\dot{\theta}_j{}^{-} \\
-100.0\dot{x}_b{}^{+} + 100.0\dot{x}_b{}^{-} \\
-100.0\dot{y}_b{}^{+} + 100.0\dot{y}_b{}^{-} \\
-625.0\dot{\theta}_b{}^{+} + 625.0\dot{\theta}_b{}^{-} \\
-312.5\left(\dot{\theta}_b{}^{+}\right)^2 + 312.5\left(\dot{\theta}_b{}^{-}\right)^2 - 12.5\left(\dot{\theta}_j{}^{+}\right)^2 + 12.5\left(\dot{\theta}_j{}^{-}\right)^2 - 50.0\left(\dot{x}_b{}^{+}\right)^2 + 50.0\left(\dot{x}_b{}^{-}\right)^2 - 2.0\left(\dot{x}_j{}^{+}\right)^2 + 2.0\left(\dot{x}_j{}^{-}\right)
\end{bmatrix}
$$

1.0.6 simulate and compute trajectory

[101]:
```
#solves constraint equations to determine whether or not impact is occurring
def impact_condition(s,threshold=1e-1):
    i = 0
    for phi in phi_funcs:
        phi_val = phi(s)
        if phi_val < threshold and phi_val > -threshold:
            return i+1 #which condition impacted
        i+=1
    return 0

#solves necessary impact update equations
def impact_update(s,condition):
    sub_vals = {
        xjminus:s[0],
        yjminus:s[1],
        thjminus:s[2],
        xbminus:s[3],
        ybminus:s[4],
        thbminus:s[5],

        xjdotminus:s[6],
        yjdotminus:s[7],
        thjdotminus:s[8],
        xbdotminus:s[9],
        ybdotminus:s[10],
        thbdotminus:s[11]
    }

    cur_ieqs = impacts[condition-1].subs(sub_vals)
    print('')
    i_sols = sym.solve(cur_ieqs,
                       ␣
↪[lam,xjdotplus,yjdotplus,thjdotplus,xbdotplus,ybdotplus,thbdotplus],
                       dict=True)
    if len(i_sols) > 1:
        for i in i_sols:
            if abs(i[lam]) < 1e-06:
```

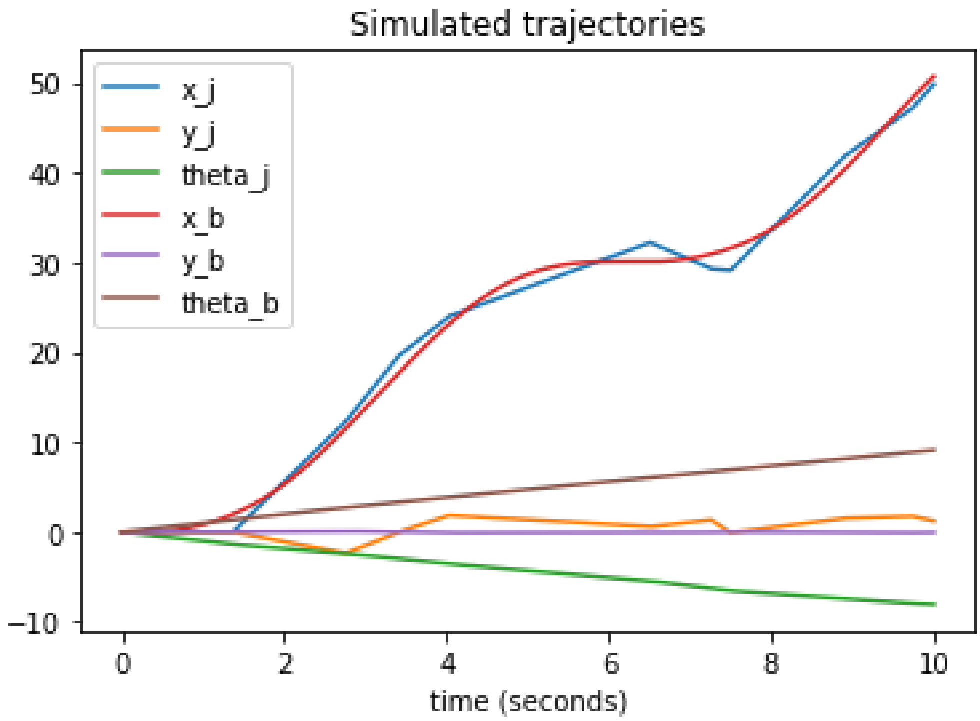
```
                pass
            else:
                return np.array([
                    s[0],s[1],s[2],s[3],s[4],s[5],
                    float(sym.N(i[xjdotplus])),
                    float(sym.N(i[yjdotplus])),
                    float(sym.N(i[thjdotplus])),
                    float(sym.N(i[xbdotplus])),
                    float(sym.N(i[ybdotplus])),
                    float(sym.N(i[thbdotplus]))
                ])
    else:
        return np.array([])

#copied from previous homeworks
def integrate(f, xt, dt, tt):
    """
    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:
    tt: current time
    step size for integration
    Return
    ============
    new_xt:
    value of x(t+dt) integrated from x(t)
    """
    k1 = dt * f(xt,tt)
    k2 = dt * f(xt+k1/2.,tt+dt/2.)
    k3 = dt * f(xt+k2/2.,tt+dt/2.)
    k4 = dt * f(xt+k3,tt+dt)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

#modify simulate function from previous homeworks
def simulate(f,x0,tspan,dt,integrate):
    N = int((max(tspan)-min(tspan))/dt)
    x = np.copy(x0)
```

```python
    tvec = np.linspace(min(tspan),max(tspan),N)
    xtraj = np.zeros((len(x0),N))
    for i in range (N):

        #check for impact with some threshold condition
        condition = impact_condition(x)
        #if impact is detected
        if condition != 0:
            print(f"impact condition {condition} has been detected at t =␣
↪{tvec[i]} sec")
            #update impact conditions
            x = impact_update(x,condition)
            xtraj[:,i] = integrate(f,x,dt,tvec[i])
        else:
            xtraj[:,i] = integrate(f,x,dt,tvec[i])
        x = np.copy(xtraj[:,i])
    return xtraj

#Return jack and box velocities and accelerations
def dyn(s,t):
    return np.array([
        s[6], s[7], s[8], s[9], s[10], s[11],
        xjfunc(*s,t),yjfunc(*s,t),thjfunc(*s,t),
        xbfunc(*s,t),ybfunc(*s,t),thbfunc(*s,t)
    ])
```

[102]:
```python
#lambdify constraint equations
phi_funcs = []
for condition in phi_list:
    phi_funcs.append(sym.lambdify([
        [q[0],q[1],q[2],q[3],q[4],q[5],
         qdot[0],qdot[1],qdot[2],qdot[3],qdot[4],qdot[5]]
    ],condition))



#Run Simulation

#[xj,yj,thj,xb,yb,thb,...]
s0 = [0,0,0,0,0,0,0,0,-1,0,0,1]
tspan = [0,10]
dt = 0.01
N = int((max(tspan)-min(tspan))/dt)
traj = simulate(dyn,s0,tspan,dt,integrate)

print(f"shape of trajectory array: {traj.shape}")
print('')
```

```python
#Plot
t_list = np.linspace(min(tspan), max(tspan), N)
fig1 = plt.figure(0)
plt.plot(t_list,traj[0])
plt.plot(t_list,traj[1])
plt.plot(t_list,traj[2])
plt.plot(t_list,traj[3])
plt.plot(t_list,traj[4])
plt.plot(t_list,traj[5])
plt.xlabel('time (seconds)')
plt.legend(['x_j','y_j','theta_j','x_b','y_b','theta_b'])
plt.title('Simulated trajectories')
plt.show()
```

```
impact condition 2 has been detected at t = 1.3813813813813813 sec

impact condition 14 has been detected at t = 2.7727727727727727 sec

impact condition 12 has been detected at t = 3.4134134134134135 sec

impact condition 4 has been detected at t = 4.044044044044044 sec

impact condition 3 has been detected at t = 6.516516516516517 sec

impact condition 6 has been detected at t = 7.267267267267267 sec

impact condition 4 has been detected at t = 7.5075075075075075 sec

impact condition 5 has been detected at t = 8.90890890890891 sec

impact condition 3 has been detected at t = 9.73973973973974 sec

shape of trajectory array: (12, 1000)
```



[103]:
```
#copy and modify from previous homeworks
def animate(ar, Lj, Lb, Wb, T=10):
    ################################
    # 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 trajectories to form Jack and Box

xx1 = ar[3]#Center of jack
yy1 = ar[4]
xx2 = xx1+(Lj/2)*np.cos(ar[5])#jack legs
yy2 = yy1+(Lj/2)*np.sin(ar[5])
xx3 = xx1-(Lj/2)*np.cos(ar[5])
yy3 = yy1-(Lj/2)*np.sin(ar[5])
xx4 = (xx1+(Lj/2)*np.sin(-ar[5]))
yy4 = yy1+(Lj/2)*np.cos(-ar[5])
xx5 = (xx1-(Lj/2)*np.sin(-ar[5]))
yy5 = yy1-(Lj/2)*np.cos(-ar[5])

xx6 = ar[0]#Center of box
yy6 = ar[1]

phi = np.arctan(Wb/Lb)
z = np.sqrt(Lb**2+Wb**2)/2
#top right
xx7 = z*np.cos(ar[2]+phi)+xx6
yy7 = z*np.sin(ar[2]+phi)+yy6
#bottom right
xx8 = xx7+(Wb*np.sin(ar[2]))
yy8 = yy7-(Wb*np.cos(ar[2]))
#bottom left
xx9 = xx6-(xx7-xx6)
yy9 = yy6-(yy7-yy6)
#top left
xx10 = xx6-(xx8-xx6)
yy10 = yy6-(yy8-yy6)


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

####################################
# Using these to specify axis limits.
xm=-2
xM=2
ym=-11
yM=11

##########################
# Defining data dictionary.
```

```
    # Trajectories are here.
    data=[dict(x=xx1, y=yy1,
               mode='lines', name='Jack',
               line=dict(width=2, color='blue')
              ),
          dict(x=xx6, y=yy6,
               mode='lines', name='Box',
              ),

        ]

    ################################
    # 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='Jack in a Box',
                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(
            #frames for jack
            x=[
                xx2[k],
                xx3[k],
                xx1[k],
                xx4[k],
                xx5[k]],
```

```
            y=[
                yy2[k],
                yy3[k],
                yy1[k],
                yy4[k],
                yy5[k]],

            mode='lines',
            line=dict(color='blue', width=3)),


         dict(
             #frames for box
             x=[xx7[k],
                xx8[k],
                xx9[k],
                xx10[k],
                xx7[k]],


             y=[yy7[k],
                yy8[k],
                yy9[k],
                yy10[k],
                yy7[k]],

             mode='lines',
             line=dict(color='red', width=3)
                        ),]) for k in range(N)]

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

[104]:
```
# Animate
theta_array = np.array([traj[3], traj[4], traj[5], traj[0], traj[1], traj[2]])
animate(theta_array,Lj=lj,Wb=dim2,Lb=dim1)
```

<IPython.core.display.HTML object>

[105]:
```
display(phi_list[15])
```

$\left[-1.0\,\mathrm{x_b}\,(t)\cos\left(\theta_b(t)\right) + 1.0\,\mathrm{x_j}\,(t)\cos\left(\theta_b(t)\right) - 1.0\,\mathrm{y_b}\,(t)\sin\left(\theta_b(t)\right) + 1.0\,\mathrm{y_j}\,(t)\sin\left(\theta_b(t)\right) + 0.5\cos\left(\theta_b(t) - \theta_j(t)\right) + 2.5\right.$

[ ]:
```

```