Ilan Mayer
ME314
Professor Todd Murphey
Due 06/08/2023

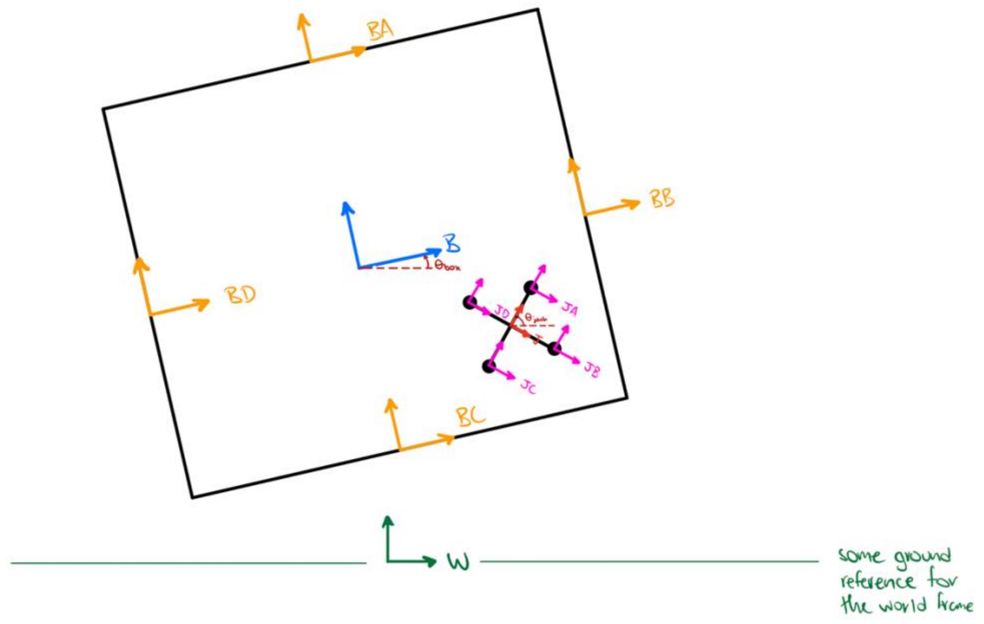
ME314 Final Project

Description

For my final project in Mechanical Engineering 314, I decided to choose the default project. This consists of a planar representation of the behavior of a jack or dice in a box. The jack has 4 corners, which can come into contact with any of the 4 walls of the box (impact). The idea behind this project is to simulate and animate the trajectory of the jack as it moves around in the box and comes in contact with the walls. It is important to mention that the box also rotates due to external forces acting on it. The project requires a complete incorporation of several things we have studied throughout the year; configuration variables, Euler Lagrange equations, impact update laws, constraints, homogenous rigid body transformations; rotational inertia, amongst several others. The image below is a drawing that models the several frames used to describe the translations and rotations of both the jack and the box. These frames allow us to map coordinate points from anybody frame to the world frame. More specifically in the jack in the box, we can determine rotational and translational kinetic energy for any body (4 walls of box and 4 edges of jack) in terms of the world frame, allowing us to determine kinetic energy, potential energy, and eventually the Euler Lagrange equation.

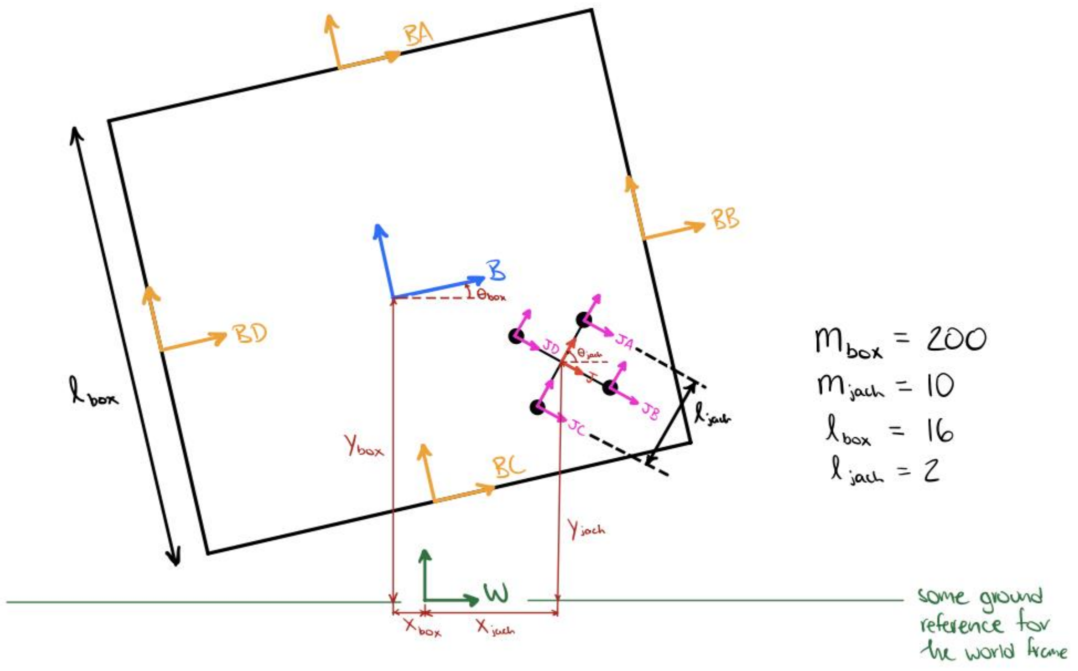
System Drawing

$$q = [x_{box}, y_{box}, \theta_{box}, x_{jack}, y_{jack}, \theta_{jack}]$$



The frames used are B, BA, BB, BC, BD, J, JA, JB, JC, JD. The B frame is located at the center of mass of the box, which happens to be its central position. The B-frame is rotated by an amount theta_box (which is an angle between the x-axis of the B frame and the x-axis of the world frame), and translated by an amount of x_box, y_box. Therefore, B represents the location (x,y) and rotation of the box. BA, BB, BC, and BD are located at the center of mass of each wall of the box (which happens to be the midpoint for each wall). The homogenous rigid body transformation from the B frame to any frame BA, BB, BC, or BD is simply a translation by an amount of l_box/2 in either positive or negative x, positive or negative y, or both, depending on which BA, BB, BC, BD you are moving to. These body frames work to determine the KE and PE for each wall with respect to the world frame and allow us to calculate the lagrangian to see the acceleration of the configuration variables for the box.

The J frame is located at the center of mass of the jack, which happens to be its central position. The J frame is rotated by an angle of theta_jack (which is the angle between the x-axis of the J frame and the x-axis of the world frame), and translated by an amount of x_jack, y_jack. Therefore, J represents the location (x,y) and rotation of the jack. JA, JB, JC, JD are located at the edges of the jack. The homogenous rigid body transformation from the J frame to any frame JA, JB, JC, JD is simply a translation by an amount of l_jack/2 in either positive or negative x, positive or negative y, or both, depending on which JA, JB, JC, JD you are moving to. These body frames are used to find the kinetic energy and potential energy of the jack with respect to the world frame.



The configuration variables of the system can be illustrated above in the wine-red color. The configuration q is given by the following variables: the x-position of the center of mass of the box (the center) relative to the world frame, the y-position of the center of mass of the box (the center) relative to the world frame, the rotation of the box as an angle measured from the x-axis of the world frame to the x-axis of the B frame, the x-position of the center of mass of the jack (the center) relative to the world frame, the y-position of the center of mass of the jack (the center) relative to the world frame, and the rotation of the jack as an angle measured from the x-axis of the world frame to the x-axis of the J frame.

The dimensions of the system are also illustrated above. The length of the box (which is a square) is given by l_box and has a magnitude of 16 units. The length of the jack, from one edge to another, is given by l_jack and has a magnitude of 2 units. Furthermore, the mass of the box is given by m_box and has a magnitude of 200 units. The mass of the jack is given by m_jack and has a magnitude of 10 units.

Code Breakdown

The homogeneous rigid body transformations were defined using the figure shown above. First, the transformations from the world frame to B, BA, BB, BC, BD framed were computed. This was done by using the SE3 helper function I created in HW7. The transformations from the world to the frame at the COM of the jack (J frame) and all the frames at the edges of the jack were computed in the same manner. The purpose of these frames is so that we can find the KE and PE of the jack and box in the J and B frames and use the transformations to get KE and PE in terms of the world frame. This is what allows us to compute the lagrangian of the entire system. To get KE, the rotational inertia tensors for both the jack and box were done so just like HW7, knowing that there is only rotational inertia about the z-axis. Once the tensors were computed, the dynamic matrix for both jack and box were computed by making a 6x6 like in HW7. The place in where the rigid body transformations came into play was in computing the body velocities of the jack and box. The body velocity of the box was done by taking the inverse of the g_W_B matrix and multiplying it by its derivative. g_W_B is the transformation from the world frame to the box COM frame, which we had defined earlier. With the body velocity and inertia tensor, we are able to compute the kinetic energy of the box in the world frame. This shows how rigid body transformations make something like computing the KE of a moving object wrt the world frame much simpler. The exact same is done for the jack's KE. For the PE, it was simply obtaining the y-coordinate of the position of the origin of J and B frames wrt world frame. This gave us the height of J and B frames wrt world frame, which we could then plug into PE formula. With KE and PE, getting the lagrangian was simply finding the difference between them, and the left-hand side was simply found by using the regular EL formula. L.diff(qdot).diff(t) – L.diff(q).

The external forces were created in a way that there would be rotation for theta_box, but the box would stay in the same x, y coordinates. This was done by placing a force in the y-direction to counteract the force of gravity on the box, no force along x, and a force along theta that was used by plugging in a desired theta trajectory (inspired from HW7). This composed the rhs of the EL equations. A force vector F, was created, were the only non-zero components were the 2 and 3 elements (corresponding to y-box and theta-box configurations)… as the force only acts on those configurations. With the lhs and rhs of the EL, it could be then solved to find the acceleration of all configurations. Solutions were then lambdified.

The heart of the code lied within impacts, which used rigid body transformations to determine the impact constraints (phi), of which there were 16… for the 16 different possible impacts. I needed to find the transformation from each wall to each edge (16 different transformations). This is because I knew that when, for example, the y-coordinate of the transformation from the BA frame (wall A) to the JA frame (A edge of jack) was 0, there was an impact between wall A and edge A. Why so? Because this meant that the y-coordinate position of the origin of the JA frame wrt BA was 0. And since the wall A is at y=0 on the BA frame, this meant an impact was occurring. This was one of my impact constraints (where impact occurs). I multiplied the inverse of the rigid body transformation from the world to a wall frame by the rigid body transformation of the world frame to a jack edge frame in order to give me the homogeneous rigid body transformation from that wall frame to that edge frame. I repeated this process for each different combination of wall and edge (which represented each unique impact… 16 of them). Once I had all these transforms, I could begin writing my impact constraints. As I mentioned before, for the BA frame, impact occurred when the y-coordinate was 0, but for the BB frame impact was when the x-coordinate was 0. This is bc of how the wall aligns with the frame (aka wall B is at x=0 on BB frame). So I got either the x or y coordinate of each transformation (depending on which I needed) and set that equal to the constraint phi. I now had 16 of them.

To get the impact update laws, I computed the conserved quantity (P) and the Hamiltonian (H) of the system. P was computed by taking the derivative of the lagrange with respect to the previously defined derivative of configuration matrix q_dot. H was done so by multiplying the transpose of P and qdot and subtracting the lagrangian from that. I then needed to define a dummy dictionary for the configurations and configuration derivatives before impact (tau plus) and configurations and configuration derivatives after impact (tau minus). I substituted the matrix P for tp dummies and P for tm dummies, and did the same for the scalar H. I then subtracted P tau plus from P tau minus and the same for H. I created matrices for both to store the values. One of my impact update laws was simple, setting H_tau_plus minus H_tau_minus equal to zero. The challenge arose for the other 16 equations, which are defined by setting P_tau_plus minus P_tau_minus equal to the constraint wrt q (configuration) times a variable lambda. I set up a matrix for all phi I had previously determined, and then took the derivative of each one wrt to all 6 configurations. This gave me a 16x7 matrix. The first column corresponded to all derivatives taken wrt the first configuration variable q[0], the second column to all derivatives taken wrt to second configuration q[1], and so on and so forth. The Ptp – Ptm matrix worked in a way where the first row was the derivatives taken wrt to the first configuration variable q[0] the second for the second configuration, and so on. The first impact law was essentially the first element of the Ptp – Ptm matrix set equal to the element in the first row and first column of the 16x7 dphi_dq matrix. The second equation was the second element of P set equal to the element in the first row but second column of the 16x7 matrix. As you can see, the each side of the equation was a derivative wrt the same configuration variable. The way this worked, is that I generated a for loop of the length of elements in phi (16) and iterated a value "i" over this list. I took the first row of the 16x7 multiplied it by lambda and set it equal to the corresponding P element. Then the second row, then so on. The equations were appended to an impact_eqns list where I stored all my impact update laws.

Simulation Explanation

The first simulation graph was the position of the configurations corresponding to the box over time. As I wanted, the x_box and y_box configurations barely changed over time, meanwhile the theta_box rotated as it was being forced by the external force I defined to act theta_box. The second graph was the positions of the jack configuration variables over time. As seen in the plot, the x-jack, y-jack, and theta-jack are all changing value. However, what shows impact is the sharp edges in the plots. As we can see, these sharp points in x_jack, y_jack, and theta_jack align at the same time stamp. By counting these sharp edges on the graph, we can see about 8 impacts. Another way of seeing it, is that if you look at the derivatives (line tangent) to each plot, at each sharp edge, the tangent line changes direction instantaneously. These instantaneous changes in velocity represent impact! Therefore, this simulation seems to represent impact of the jack. The third graph is the velocities of the box configuration variables. Here, we can see the theta_box velocity follows the forcing trajectory it was subjected to. The x_box and y_box velocities have sharp changes (but very small). These are impacts with the jack and the box, because since the box is floating (standstill) along x and y, when it is hit by the jack, it has a sharp change in velocity. Lastly, where impacts can be most clearly seen, is in the graph of velocities of the jack configuration variables. For the velocity of the jack along x, y, and theta, there are instantenous changes in velocity that lign up at certain timestamps. These are clearly impacts, as the x, y, and theta velocities change and are updated by the impact update laws at that instant.

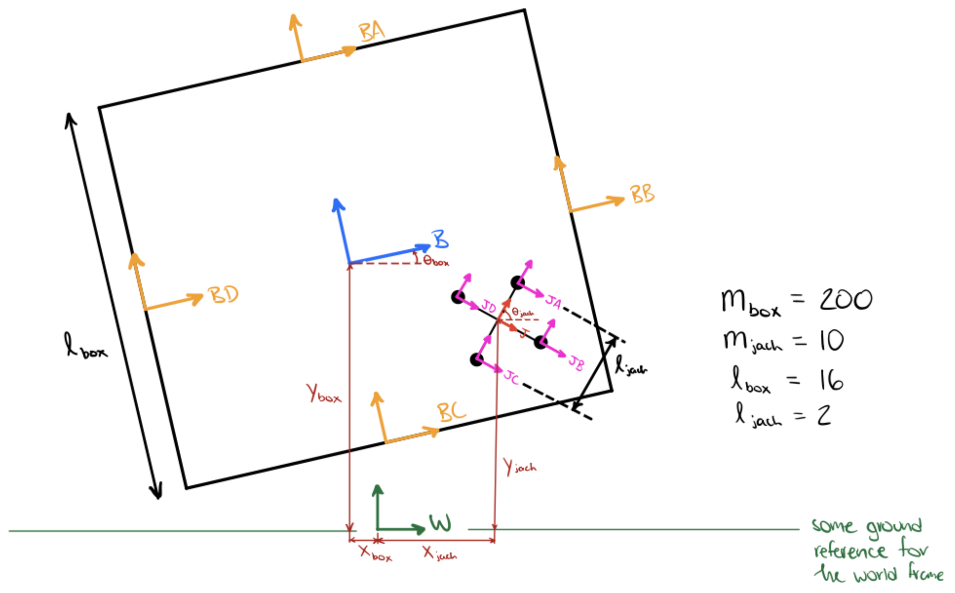
ME314 FINAL PROJECT

For this final project, I decided to go with the default project. This consists of a planar representation of the behavior of a jack or dice in a box. The jack has 4 corners, which can come into contact with any of the 4 walls of the box (impact). The idea behind this project is to simulate and animate the trajectory of the jack as it moves around the box and comes into contact with the walls

- collaboration w/ Valeria Yzaga

```
# importing necessary libraries
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
```

```
from IPython.display import Image
Image('Jack-Box_System.png')
```



```
#### ME314 FINAL PROJECT CODE ####

t = sym.symbols('t')
l_box, l_jack, m_box, m_jack, g = sym.symbols('l_box, l_jack, m_box, m_jack, g')
from sympy import sin, cos

# define constants for subs
consts = {l_box: 16, l_jack: 2, m_box: 200, m_jack: 10, g: 9.81}
```

```

# -------------- defining all necssary helper functions (taken from HW7) -------------

# Defining Function to compute SE(3) given rotation angle and translation.
def SE3(theta, p):
  assert len(p) == 3
  g = sym.Matrix([ [cos(theta), -sin(theta), 0, p[0]], [sin(theta), cos(theta), 0, p[1]], [0, 0, 0, 0], [0,
  return g

# Defining function to compute the derivative of an SE3
def derivate(m):
  return m.diff(t)

# Defining a function to compute the inverse of an SE3
def inverse(m):
  temp = sym.zeros(4, 4)
  temp[0:3, 0:3] = m[0:3, 0:3].T
  temp[0:3, 3] = -temp[0:3, 0:3] * m[0:3, 3]
  temp[3, 0:3] = sym.zeros(1, 3)
  temp[3, 3] = 1
  return temp

# Defining a function that converst a hatted 4x4 matrix into an unhatted 6x1 vector
def unhat(m):
    assert m.shape == (4, 4)
    w_hat = m[0:3, 0:3]
    w = sym.Matrix([w_hat[2, 1], -w_hat[2, 0], w_hat[1, 0]])
    v = m[0:3, 3]  # Select the first three elements of the fourth column
    unhat = sym.Matrix.vstack(v, w)
    return unhat

# Defining a function that converst an unhatted 6x1 vector into a hatted 4x4 matrix
def hat(m):
    assert m.shape == (6, 1)
    v = m[0:3, 0]
    w = m[3:6, 0]
    w_hat = sym.Matrix([[0, -w[2], w[1]], [w[2], 0, -w[0]], [-w[1], w[0], 0]])
    hat = sym.BlockMatrix([[w_hat, v], [sym.zeros(1, 3), sym.zeros(1, 1)]])
    hat = hat.as_explicit()
    return hat


# -------------  begin by defining necessary variables and functions -------------

# configuration variables
x_box = sym.Function('x_box')(t)
y_box = sym.Function('y_box')(t)
theta_box = sym.Function('theta_box')(t)
x_jack = sym.Function('x_jack')(t)
y_jack = sym.Function('y_jack')(t)
theta_jack = sym.Function('theta_jack')(t)

q = sym.Matrix([x_box, y_box, theta_box, x_jack, y_jack, theta_jack])

# derivatives of configuration variables
x_boxdot = x_box.diff(t)
y_boxdot = y_box.diff(t)
theta_boxdot = theta_box.diff(t)
x_jackdot = x_jack.diff(t)
y_jackdot = y_jack.diff(t)
theta_jackdot = theta_jack.diff(t)

x_boxddot = x_boxdot.diff(t)
y_boxddot = y_boxdot.diff(t)
theta_boxddot = theta_boxdot.diff(t)
```

```
x_jackddot = x_jackdot.diff(t)
y_jackddot = y_jackdot.diff(t)
theta_jackddot = theta_jackdot.diff(t)

# ------------- defining the homogeneous rigid body transformations -------------
# while computing transformations, sub for constants

# world frame to B frame
g_W_B = (SE3(theta_box, sym.Matrix([x_box, y_box, 0]))).subs(consts)
# B frame to BA frame
g_B_BA = (SE3(0, sym.Matrix([0, l_box/2, 0]))).subs(consts)
# B frame to BB frame
g_B_BB = (SE3(0, sym.Matrix([l_box/2, 0, 0]))).subs(consts)
# B frame to BC frame
g_B_BC = (SE3(0, sym.Matrix([0, -l_box/2, 0]))).subs(consts)
# B frame to BD frame
g_B_BD = (SE3(0, sym.Matrix([-l_box/2, 0, 0]))).subs(consts)

# world frame to J frame
g_W_J = (SE3(theta_jack, sym.Matrix([x_jack, y_jack, 0]))).subs(consts)
# J frame to JA frame
g_J_JA = (SE3(0, sym.Matrix([0, l_jack/2, 0]))).subs(consts)
# J frame to JB frame
g_J_JB = (SE3(0, sym.Matrix([l_jack/2, 0, 0]))).subs(consts)
# J frame to JC frame
g_J_JC = (SE3(0, sym.Matrix([0, -l_jack/2, 0]))).subs(consts)
# J frame to JD frame
g_J_JD = (SE3(0, sym.Matrix([-l_jack/2, 0, 0]))).subs(consts)

# define transforms, use SE3 function, then define multiplications
# world frame to BA frame
g_W_BA = g_W_B*g_B_BA
# world frame to BB frame
g_W_BB = g_W_B*g_B_BB
# world frame to BC frame
g_W_BC = g_W_B*g_B_BC
# world frame to BD frame
g_W_BD = g_W_B*g_B_BD

# world frame to JA frame
g_W_JA = g_W_J*g_J_JA
# world frame to JB frame
g_W_JB = g_W_J*g_J_JB
# world frame to JC frame
g_W_JC = g_W_J*g_J_JC
# world frame to JD frame
g_W_JD = g_W_J*g_J_JD


# the code above defines all of the frames with respect to the world frame. Now, we have to determine the t
# This is necessary because we need to find when the position of the origin of any of the J frames is 0 wr
# The following lines of code describe these transformations from box walls to jack edges, there should be

# transformation to from the BA (box wall A) frame to all J frames (edges of the jack)
g_BA_JA = inverse(g_W_BA) * g_W_JA
g_BA_JB = inverse(g_W_BA) * g_W_JB
g_BA_JC = inverse(g_W_BA) * g_W_JC
g_BA_JD = inverse(g_W_BA) * g_W_JD

# transformation to from the BB (box wall B) frame to all J frames (edges of the jack)
g_BB_JA = inverse(g_W_BB) * g_W_JA
g_BB_JB = inverse(g_W_BB) * g_W_JB
g_BB_JC = inverse(g_W_BB) * g_W_JC
g_BB_JD = inverse(g_W_BB) * g_W_JD

```

```
# transformation to from the BC (box wall C) frame to all J frames (edges of the jack)
g_BC_JA = inverse(g_W_BC) * g_W_JA
g_BC_JB = inverse(g_W_BC) * g_W_JB
g_BC_JC = inverse(g_W_BC) * g_W_JC
g_BC_JD = inverse(g_W_BC) * g_W_JD

# transformation to from the BD (box wall D) frame to all J frames (edges of the jack)
g_BD_JA = inverse(g_W_BD) * g_W_JA
g_BD_JB = inverse(g_W_BD) * g_W_JB
g_BD_JC = inverse(g_W_BD) * g_W_JC
g_BD_JD = inverse(g_W_BD) * g_W_JD

# ------------ calculate the body inertia matrices for the box and the jack -----------
# --> concepts here are taken from HW7

## rotational inertia of box
# the rotational inertia of a wall of the box can be given by the following inside the parenthesis. The ti
J_box = 4*(m_box*(l_box/2)**2) # l_box/2 is the distance from the center to the COM of any wall
## rotational inertia of jack
# from homework 7 we saw that a stick (with no width) of length 1 and mass 1 has rotational inertia of 1, w
J_jack = 4*(m_jack*(l_jack/2)**2) # l_jack/2 is the distance from the center to an edge

# Defining the rotational Inertia Tensors
# there is only rotational inertia in the z-direction
I_rot_box = sym.Matrix([[0, 0, 0], [0, 0, 0], [0, 0, J_box]])
# for the rectangle with COM at b, it has the same rotational inertia
I_rot_jack = sym.Matrix([[0, 0, 0], [0, 0, 0], [0, 0, J_jack]])

# defining the dynamic matrix of the box
dyn_mat_box = sym.BlockMatrix( [ [4*m_box*sym.eye(3), sym.zeros(3,3)], [sym.zeros(3,3), I_rot_box] ] )
dyn_mat_box = dyn_mat_box.as_explicit()

# defining the dynamic matrix of the jack
dyn_mat_jack = sym.BlockMatrix( [ [4*m_jack*sym.eye(3), sym.zeros(3,3)], [sym.zeros(3,3), I_rot_jack] ] )
dyn_mat_jack = dyn_mat_jack.as_explicit()


# ------------ body velocities for the box and the jack -----------
# body velocity of box
V_hat_box = inverse(g_W_B) * derivate(g_W_B)
V_box = unhat(V_hat_box)

# body velocity of jack
V_hat_jack = inverse(g_W_J) * derivate(g_W_J)
V_jack = unhat(V_hat_jack)

# ------------ kinetic energy of system -----------
KE = 0.5*V_box.T*dyn_mat_box*V_box + 0.5*V_jack.T*dyn_mat_jack*V_jack

# ------------ external forcing on box -----------
# the following external forces act on the box in the x, y, and theta (to make oscillation)

F_y_box = 4*m_box*g + 4*m_jack*g
F_theta_box = 20000*(sin(2*np.pi*t/5))
F = sym.Matrix([0, F_y_box, F_theta_box, 0, 0, 0])

# ------------ potential energy of system ----------
# define the height of COM of box and jack
rB_bar = sym.Matrix([0, 0, 0, 1])
rWB_bar = g_W_B*rB_bar
h_box = sym.Matrix([0, 1, 0, 0]).T*rWB_bar

rJ_bar = sym.Matrix([0, 0, 0, 1])
rWJ_bar = g_W_J*rJ_bar
h_jack = sym.Matrix([0, 1, 0, 0]).T*rWJ_bar
```

```
PE = 4*m_box*g*h_box + 4*m_jack*g*h_jack

# ------------ lagrangian of system ----------

# substitute constants
L = (KE[0] - PE[0]).subs(consts).simplify()

# ------------ euler-lagrange equations ----------
qdot = q.diff(t)

# define solutions and substitute in constants

# calculating the lhs of the EL equations
lhs = (L.diff(qdot)).diff(t) - (L.diff(q)).simplify()
lhs = sym.Matrix([ (lhs[0]).simplify(), (lhs[1]).simplify(), (lhs[2]).simplify(), (lhs[3]).simplify(), (lhs

# the rhs of the EL equations is the forcing term and sub in the constants
rhs = F.subs(consts)

# settig up equation in sympy
Eqs = sym.Eq(lhs, rhs)

# define what we are solving for
c = [x_boxddot, y_boxddot, theta_boxddot, x_jackddot, y_jackddot, theta_jackddot]

# solving
soln = sym.solve(Eqs, c)

# defining solutions
x_boxdd = soln[x_boxddot].subs(consts).simplify()
y_boxdd = soln[y_boxddot].subs(consts).simplify()
theta_boxdd = soln[theta_boxddot].subs(consts).simplify()
x_jackdd = soln[x_jackddot].subs(consts).simplify()
y_jackdd = soln[y_jackddot].subs(consts).simplify()
theta_jackdd = soln[theta_jackddot].subs(consts).simplify()


# lambdify solutions
l1 = sym.lambdify([x_box, y_box, theta_box, x_jack, y_jack, theta_jack, x_boxdot, y_boxdot, theta_boxdot, x
l2 = sym.lambdify([x_box, y_box, theta_box, x_jack, y_jack, theta_jack, x_boxdot, y_boxdot, theta_boxdot, x
l3 = sym.lambdify([x_box, y_box, theta_box, x_jack, y_jack, theta_jack, x_boxdot, y_boxdot, theta_boxdot, x
l4 = sym.lambdify([x_box, y_box, theta_box, x_jack, y_jack, theta_jack, x_boxdot, y_boxdot, theta_boxdot, x
l5 = sym.lambdify([x_box, y_box, theta_box, x_jack, y_jack, theta_jack, x_boxdot, y_boxdot, theta_boxdot, x
l6 = sym.lambdify([x_box, y_box, theta_box, x_jack, y_jack, theta_jack, x_boxdot, y_boxdot, theta_boxdot, x
```

```
# --------------------- IMPACTS --------------------- #


# defining impact constraint for wall A. This impact occurs when the y-value of the location of the origin

# the rorigin_hat represents the coordinate position x, y, z of any origin in any frame.
rorigin_hat = sym.Matrix([0, 0, 0, 1])
# determening the position of origin JA wrt to frame BA
r_BA_JA = g_BA_JA*rorigin_hat
# impact occurs when the y-coordinate of r_BA_JA is equal to zero, bc that is when the origin JA is at the
phi_BA_JA = sym.Matrix([0, 1, 0, 0]).T*r_BA_JA
# for the following impact constraints, the process is the same
r_BA_JB = g_BA_JB*rorigin_hat
phi_BA_JB = sym.Matrix([0, 1, 0, 0]).T*r_BA_JB
r_BA_JC = g_BA_JC*rorigin_hat
phi_BA_JC = sym.Matrix([0, 1, 0, 0]).T*r_BA_JC
r_BA_JD = g_BA_JD*rorigin_hat
phi_BA_JD = sym.Matrix([0, 1, 0, 0]).T*r_BA_JD

```

```
# defining impact constraint for wall B. This impact occurs when the x-value of the location of the origin
r_BB_JA = g_BB_JA*rorigin_hat
# the difference now is that the impact occurs when the x-value of the origin of JA wrt to BB is equal to 
phi_BB_JA = sym.Matrix([1, 0, 0, 0]).T*r_BB_JA
r_BB_JB = g_BB_JB*rorigin_hat
phi_BB_JB = sym.Matrix([1, 0, 0, 0]).T*r_BB_JB
r_BB_JC = g_BB_JC*rorigin_hat
phi_BB_JC = sym.Matrix([1, 0, 0, 0]).T*r_BB_JC
r_BB_JD = g_BB_JD*rorigin_hat
phi_BB_JD = sym.Matrix([1, 0, 0, 0]).T*r_BB_JD

# defining impact constraint for wall C. This impact occurs when the y-value of the location of the origin
r_BC_JA = g_BC_JA*rorigin_hat
# the difference now is that the impact occurs when the y-value of the origin of JA wrt to BC is equal to 
phi_BC_JA = sym.Matrix([0, 1, 0, 0]).T*r_BC_JA
r_BC_JB = g_BC_JB*rorigin_hat
phi_BC_JB = sym.Matrix([0, 1, 0, 0]).T*r_BC_JB
r_BC_JC = g_BC_JC*rorigin_hat
phi_BC_JC = sym.Matrix([0, 1, 0, 0]).T*r_BC_JC
r_BC_JD = g_BC_JD*rorigin_hat
phi_BC_JD = sym.Matrix([0, 1, 0, 0]).T*r_BC_JD

# defining impact constraint for wall D. This impact occurs when the y-value of the location of the origin
r_BD_JA = g_BD_JA*rorigin_hat
# the difference now is that the impact occurs when the x-value of the origin of JA wrt to BD is equal to 
phi_BD_JA = sym.Matrix([1, 0, 0, 0]).T*r_BD_JA
r_BD_JB = g_BD_JB*rorigin_hat
phi_BD_JB = sym.Matrix([1, 0, 0, 0]).T*r_BD_JB
r_BD_JC = g_BD_JC*rorigin_hat
phi_BD_JC = sym.Matrix([1, 0, 0, 0]).T*r_BD_JC
r_BD_JD = g_BD_JD*rorigin_hat
phi_BD_JD = sym.Matrix([1, 0, 0, 0]).T*r_BD_JD


# now we can make a dummy variables to replace for the EL solutions and the impact constraints
x_box_s, y_box_s, theta_box_s, x_jack_s, y_jack_s, theta_jack_s, x_boxdot_s, y_boxdot_s, theta_boxdot_s, x_

# make dummy dictionary for substituion
dummy = {x_box:x_box_s, y_box:y_box_s, theta_box:theta_box_s, x_jack:x_jack_s, y_jack:y_jack_s, theta_jack

# make a matrix q_dd that represents the accelerations and subsitute the dummy variables for the EL soluti
q_dd = sym.Matrix([x_boxdd, y_boxdd, theta_boxdd, x_jackdd, y_jackdd, theta_jackdd]) # no need to sub for 

# make a matrix for the impact constraints
phi = sym.Matrix([phi_BA_JA, phi_BA_JB, phi_BA_JC, phi_BA_JD,
                  phi_BB_JA, phi_BB_JB, phi_BB_JC, phi_BB_JD,
                  phi_BC_JA, phi_BC_JB, phi_BC_JC, phi_BC_JD,
                  phi_BD_JA, phi_BD_JB, phi_BD_JC, phi_BD_JD])


# defining the impact update equations,
P = (L.diff(qdot)) # momentum of the system
H = ((P.T*qdot)[0] - L) # Hamiltonian of the system
phi_dot = phi.jacobian(q) # jacobian of the impact constraints wrt to q (impact constraints)


# substitute with dummy variables
P = P.subs(dummy)
H = H.subs(dummy)
phi_d = phi_dot.subs(dummy)

phi_sub = phi.subs(dummy)


# define dummy symbols for q(Tau-), q_d(Tau-), q(Tau+), q_d(Tau+)
```

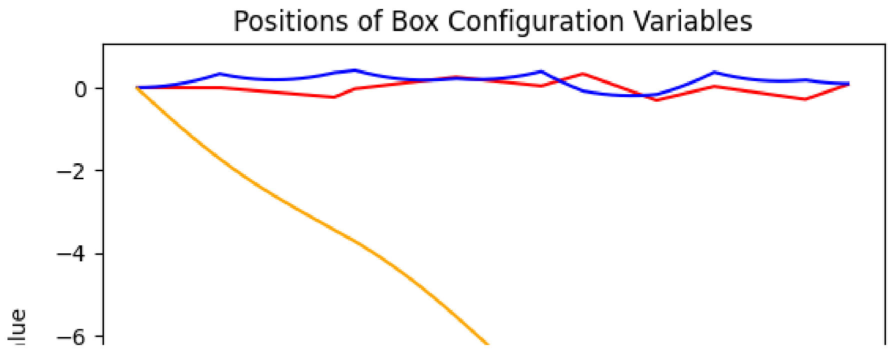
```
x_box_tm = sym.symbols('x_box^-')
y_box_tm = sym.symbols('y_box^-')
theta_box_tm = sym.symbols('theta_box^-')
x_jack_tm = sym.symbols('x_jack^-')
y_jack_tm = sym.symbols('y_jack^-')
theta_jack_tm = sym.symbols('theta_jack^-')
x_box_d_tm = sym.symbols('xdot_box^-')
y_box_d_tm = sym.symbols('ydot_box^-')
theta_box_d_tm = sym.symbols('thetadot_box^-')
x_jack_d_tm = sym.symbols('xdot_jack^-')
y_jack_d_tm = sym.symbols('ydot_jack^-')
theta_jack_d_tm = sym.symbols('thetadot_jack^-')

## The configuration does not change after impact so tp = tm
x_box_tp = x_box_tm
y_box_tp = y_box_tm
theta_box_tp = theta_box_tm
x_jack_tp = x_jack_tm
y_jack_tp = y_jack_tm
theta_jack_tp = theta_jack_tm

x_box_d_tp = sym.symbols('xdot_box^+')
y_box_d_tp = sym.symbols('ydot_box^+')
theta_box_d_tp = sym.symbols('thetadot_box^+')
x_jack_d_tp = sym.symbols('xdot_jack^+')
y_jack_d_tp = sym.symbols('ydot_jack^+')
theta_jack_d_tp = sym.symbols('thetadot_jack^+')

# define dummy dictionary for tau neg
tm = {x_box_s: x_box_tm, y_box_s: y_box_tm, theta_box_s: theta_box_tm, x_jack_s: x_jack_tm, y_jack_s: y_ja
tp = {x_box_s: x_box_tp, y_box_s: y_box_tp, theta_box_s: theta_box_tp, x_jack_s: x_jack_tp, y_jack_s: y_ja

P_tp = P.subs(tp)
P_tm = P.subs(tm)
H_tp = H.subs(tp)
H_tm = H.subs(tm)

# defining the left hand side of the impact update equations
eq1_lhs = P_tp[0] - P_tm[0]
eq2_lhs = P_tp[1] - P_tm[1]
eq3_lhs = P_tp[2] - P_tm[2]
eq4_lhs = P_tp[3] - P_tm[3]
eq5_lhs = P_tp[4] - P_tm[4]
eq6_lhs = P_tp[5] - P_tm[5]
eq7_lhs = H_tp - H_tm

# lhs of the impact equations
lhs = sym.Matrix([eq1_lhs.simplify(), eq2_lhs.simplify(), eq3_lhs.simplify(), eq4_lhs.simplify(), eq5_lhs.s


# define lambda
lam = sym.symbols('lambda')

# make empty array for impact_eqns
impact_eqns = []

for i in range(phi_sub.shape[0]):
    rhs = sym.Matrix([lam*phi_d[i,0],
                      lam*phi_d[i,1],
                      lam*phi_d[i,2],
                      lam*phi_d[i,3],
                      lam*phi_d[i,4],
                      lam*phi_d[i,5],
                      0])
    impact_eqns.append(sym.Eq(lhs, rhs.subs(tm)))
```

```


# Lambdify the impact constraints
phi_lamb = sym.lambdify([x_box_s, y_box_s, theta_box_s, x_jack_s, y_jack_s, theta_jack_s, x_boxdot_s, y_box

# create an impact update function that takes in the state(s) and solves
def impact_update(s, impact):
    subs = {x_box_tm: s[0], y_box_tm: s[1], theta_box_tm: s[2], x_jack_tm: s[3], y_jack_tm: s[4], theta_jac
    # subsitute the tm with the current values of the state (s)
    impact_sub = impact.subs(subs)
    # Solve for impact update variables
    impact_solns = sym.solve(impact_sub, [x_box_d_tp, y_box_d_tp, theta_box_d_tp, x_jack_d_tp, y_jack_d_tp,
    for soln in impact_solns:
        lambda_solution = soln[6]
        # ensuring that the solution lambda is zero is not used. Using the other sln
        if abs(lambda_solution) < 1e-7:
            pass
        else:
            # return updated impact state
            return np.array( [s[0], s[1], s[2], s[3], s[4], s[5], float(sym.N(soln[0])), float(sym.N(soln[


# Create a function that checks for impact!
def impact_condition(s, phi_lamb, threshold = 0.1):
    # calculating the phi value
    phi_val = phi_lamb(*s)
    # checking through all 16 phi_values (16 impact update eqs)
    for i in range(phi_val.shape[0]):
        # if phi value is inside the threshold
        if (phi_val[i] > -0.1) and (phi_val[i] < 0.1):
            # then there is impact. i marks the phi corresponding to that impact (i.e. which one of the 16
            return (True, i)
    # else, there is no impact
    return (False, None)
# print('term impact condition function:', impact_condition(s_test, phi_func, 0.01))
```

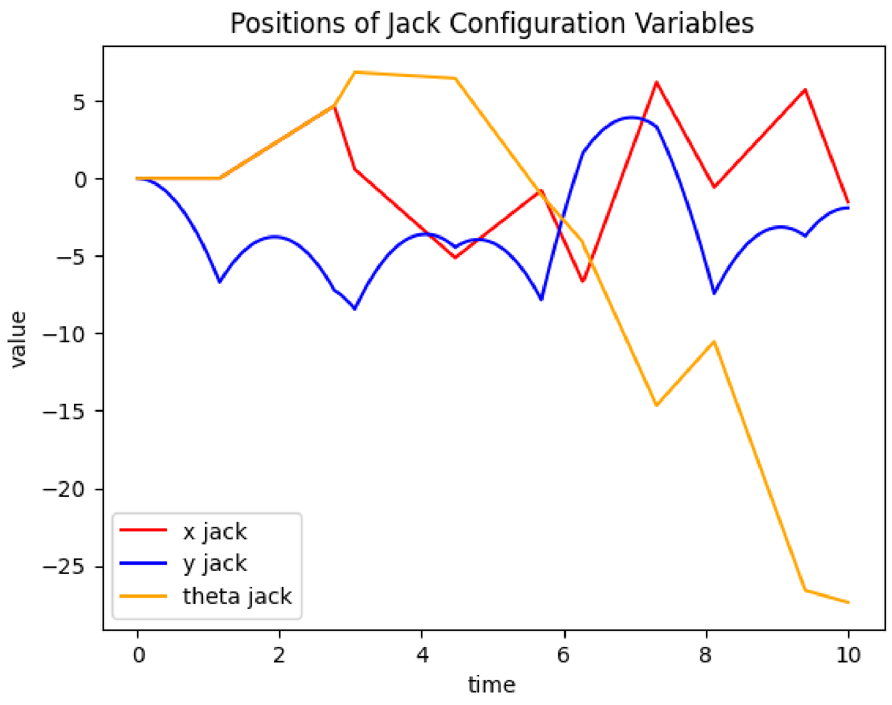
```
def dyn(s, t):
    # This function computes the derivative of the state vector
    # s is the state vector
    # t is the current time
    # The output is the time derivative of the state vector
    sdot = np.array([
        s[6],
        s[7],
        s[8],
        s[9],
        s[10],
        s[11],
        l1(*s, t),
        l2(*s, t),
        l3(*s, t),
        l4(*s, t),
        l5(*s, t),
        l6(*s, t)
    ])
    return sdot

def integrate(f, xt, dt, time):
    # This function integrates the derivative of the state vector over time
    # f is the function that computes the derivative of the state vector
    # xt is the current state vector
    # dt is the time step size
    # time is the current time
    # The output is the new state vector after time dt
```

```python
    k1 = dt * f(xt, time)
    k2 = dt * f(xt+k1/2., time)
    k3 = dt * f(xt+k2/2., time)
    k4 = dt * f(xt+k3, time)
    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

def simulate_impact(f, x0, tspan, dt, integrate):
    # This function simulates the system over time
    # f is the function that computes the derivative of the state vector
    # x0 is the initial state vector
    # tspan is the time span of the simulation
    # dt is the time step size
    # integrate is the function used to integrate the state vector over time
    # The output is the state vector at each time step and the corresponding times
    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))
    time = 0
    for i in range(N):
        (impact, impact_num) = impact_condition(x, phi_lamb, 1e-1)
        if impact is True:
            x = impact_update(x, impact_eqns[impact_num])
            xtraj[:, i]=integrate(f, x, dt, time)
        else:
            xtraj[:, i]=integrate(f, x, dt, time)
        x = np.copy(xtraj[:,i])
        time += dt
    return xtraj, tvec

```

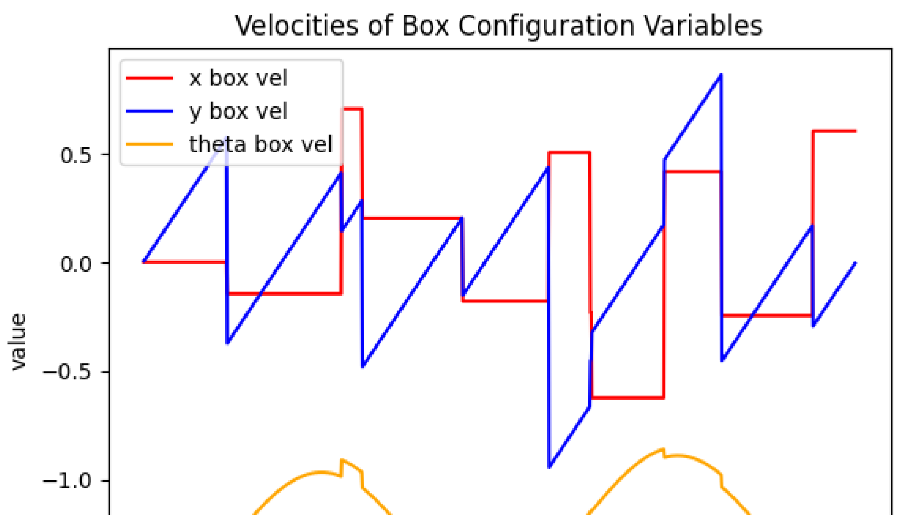
```python
# Simulate the motion:
tspan = [0, 10]
dt = 0.01
s0 = np.array([0, 0, 0, 0, 0, 0, 0, 0, -np.pi/2, 0, 0, 0])

N = int((max(tspan) - min(tspan))/dt)
tvec = np.linspace(min(tspan), max(tspan), N)
traj = simulate_impact(dyn, s0, tspan, dt, integrate)


plt.figure()
plt.plot(tvec, traj[0][0], label='x box', color = 'red')
plt.plot(tvec, traj[0][1], label='y box', color = 'blue')
plt.plot(tvec, traj[0][2], label='theta box', color = 'orange')
plt.title('Positions of Box Configuration Variables')
plt.xlabel('time')
plt.ylabel('value')
plt.legend(loc="best")
plt.show()
```



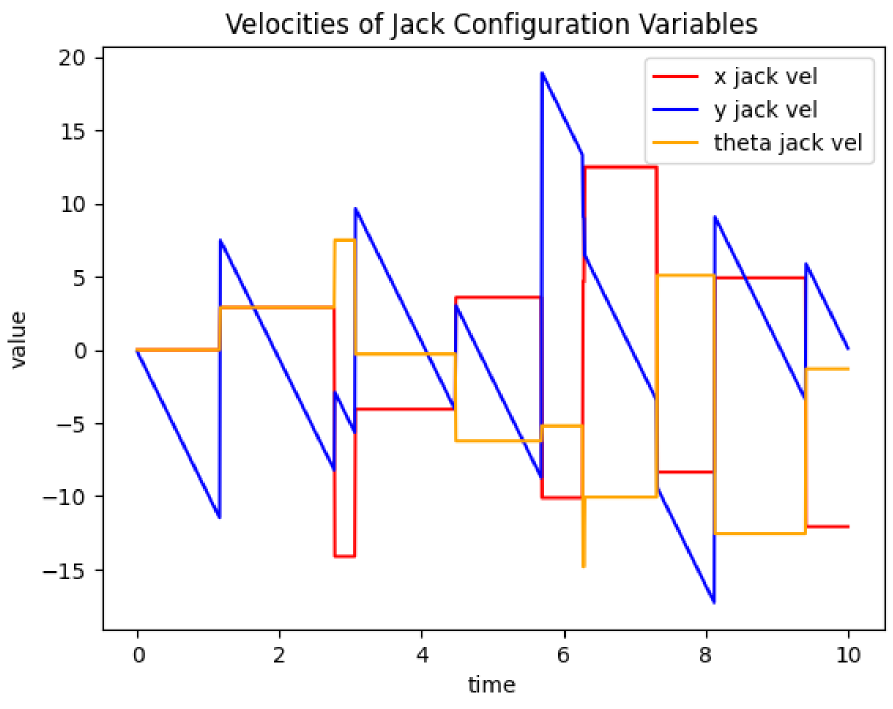
```
plt.figure()
plt.plot(tvec, traj[0][3], label='x jack', color = 'red')
plt.plot(tvec, traj[0][4], label='y jack', color = 'blue')
plt.plot(tvec, traj[0][5], label='theta jack', color = 'orange')
plt.title('Positions of Jack Configuration Variables')
plt.xlabel('time')
plt.ylabel('value')
plt.legend(loc="best")
plt.show()
```



```
plt.figure()
plt.plot(tvec, traj[0][6], label='x box vel', color = 'red')
plt.plot(tvec, traj[0][7], label='y box vel', color = 'blue')
plt.plot(tvec, traj[0][8], label='theta box vel', color = 'orange')
plt.title('Velocities of Box Configuration Variables')
plt.xlabel('time')
plt.ylabel('value')
plt.legend(loc="best")
plt.show()
```



```
plt.figure()
plt.plot(tvec, traj[0][9], label='x jack vel', color = 'red')
plt.plot(tvec, traj[0][10], label='y jack vel', color = 'blue')
plt.plot(tvec, traj[0][11], label='theta jack vel', color = 'orange')
plt.title('Velocities of Jack Configuration Variables')
plt.xlabel('time')
plt.ylabel('value')
plt.legend(loc="best")
plt.show()
```



```
def animate(config_array, T=10):
    # Import required libraries for animation
    from plotly.offline import init_notebook_mode, iplot
    from IPython.display import display, HTML
    import plotly.graph_objects as go

    # Configure the browser for Plotly
    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)

    # Extract trajectory data for the box and the jack
    N = len(config_array[0])
    x_box_array = config_array[0]
    y_box_array = config_array[1]
    theta_box_array = config_array[2]
    x_jack_array = config_array[3]
    y_jack_array = config_array[4]
    theta_jack_array = config_array[5]

    # Initialize arrays to store x and y coordinates for box and jack at each time step
    # The 'BA', 'BB', 'BC', 'BD' variables represent the center of mass of the box's walls
    # The 'J', 'JA', 'JB', 'JC', 'JD' variables represent the center and arms of the jack
    BA_x_array = np.zeros(N, dtype=np.float32)
    BA_y_array = np.zeros(N, dtype=np.float32)
    BB_x_array = np.zeros(N, dtype=np.float32)
    BB_y_array = np.zeros(N, dtype=np.float32)
    BC_x_array = np.zeros(N, dtype=np.float32)
    BC_y_array = np.zeros(N, dtype=np.float32)
    BD_x_array = np.zeros(N, dtype=np.float32)
    BD_y_array = np.zeros(N, dtype=np.float32)

    J_x_array = np.zeros(N, dtype=np.float32)
    J_y_array = np.zeros(N, dtype=np.float32)
    JA_x_array = np.zeros(N, dtype=np.float32)
    JA_y_array = np.zeros(N, dtype=np.float32)
    JB_x_array = np.zeros(N, dtype=np.float32)
    JB_y_array = np.zeros(N, dtype=np.float32)
    JC_x_array = np.zeros(N, dtype=np.float32)
    JC_y_array = np.zeros(N, dtype=np.float32)
    JD_x_array = np.zeros(N, dtype=np.float32)
    JD_y_array = np.zeros(N, dtype=np.float32)

    # For each time step, calculate the positions of the box's walls and jack's arms based on their traject
    for t in range(N):
        g_W_B = SE3(theta_box_array[t], sym.Matrix([x_box_array[t], y_box_array[t], 0]))
        g_W_J = SE3(theta_jack_array[t], sym.Matrix([x_jack_array[t], y_jack_array[t], 0]))
        l_box = 16
        # Compute the positions of the center of mass of the box's walls
        BA = g_W_B.dot(np.array([l_box/2, l_box/2, 0, 1]))
        BA_x_array[t] = BA[0]
        BA_y_array[t] = BA[1]
        BB = g_W_B.dot(np.array([l_box/2, -l_box/2, 0, 1]))
        BB_x_array[t] = BB[0]
        BB_y_array[t] = BB[1]
        BC = g_W_B.dot(np.array([-l_box/2, -l_box/2, 0, 1]))
        BC_x_array[t] = BC[0]
        BC_y_array[t] = BC[1]
        BD = g_W_B.dot(np.array([-l_box/2, l_box/2, 0, 1]))
        BD_x_array[t] = BD[0]
        BD_y_array[t] = BD[1]

        # Compute the positions of the jack's center and arms
        J = g_W_J.dot(np.array([0, 0, 0, 1]))
        J_x_array[t] = J[0]
        J_y_array[t] = J[1]
        JA = g_W_J.dot(np.array([1, 0, 0, 1]))
        JA_x_array[t] = JA[0]
        JA_y_array[t] = JA[1]
        JB = g_W_J.dot(np.array([0, -1, 0, 1]))
        JB_x_array[t] = JB[0]
        JB_y_array[t] = JB[1]
        JC = g_W_J.dot(np.array([-1, 0, 0, 1]))
        JC_x_array[t] = JC[0]
        JC_y_array[t] = JC[1]
        JD = g_W_J.dot(np.array([0, 1, 0, 1]))
        JD_x_array[t] = JD[0]
        JD_y_array[t] = JD[1]

    # Define the axis limits
    xm = -13
    xM = 13
    ym = -13
    yM = 13

    # Prepare data for the plot
    data=[dict(name = 'Box'),
          dict(name = 'Jack'),
          dict(name = 'Mass1_Jack'),
        ]

    # Prepare the layout of the simulation
    layout=dict(xaxis=dict(range=[xm, xM], autorange=False, zeroline=False,dtick=2),
                yaxis=dict(range=[ym, yM], autorange=False, zeroline=False,scaleanchor = "x",dtick=2),
                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}, 'r
                                            'transition': {'duration': 0}}],'label': 'Pause','method': 'ani
                                          ]
                              }]
               )

    # Define the frames for the simulation
    frames=[dict(data=[
            dict(x=[BA_x_array[k],BB_x_array[k],BC_x_array[k],BD_x_array[k],BA_x_array[k]],
                 y=[BA_y_array[k],BB_y_array[k],BC_y_array[k],BD_y_array[k],BA_y_array[k]],
                 mode='lines',
                 line=dict(color='black', width=3)
                 ),
            dict(x=[JA_x_array[k],JC_x_array[k],J_x_array[k],JB_x_array[k],JD_x_array[k]],
                 y=[JA_y_array[k],JC_y_array[k],J_y_array[k],JB_y_array[k],JD_y_array[k]],
                 mode='lines',
                 line=dict(color='red', width=3)
                 ),
            go.Scatter(
                x=[JA_x_array[k],JB_x_array[k],JC_x_array[k],JD_x_array[k]],
                y=[JA_y_array[k],JB_y_array[k],JC_y_array[k],JD_y_array[k]],
                mode="markers",
                marker=dict(color='red', size=6)),
                 ]) for k in range(N)]


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

```

```python


animate(traj[0])
```