```python
import numpy as np
import sympy as sym
import pandas as pd
import matplotlib.pyplot as plt

import dill
from tqdm import tqdm

from geometry import *
from helpers import *

#from IPython.core.display import display

def rk4(dxdt, x, t, dt):
    '''
    Applies the Runge-Kutta method, 4th order, to a sample function,
    for a given state q0, for a given step size. Currently only
    configured for a 2-variable dependent system (x,y).
    ==========
    dxdt: a Sympy function that specifies the derivative of the system of interest
    t: the current timestep of the simulation
    x: current value of the state vector
    dt: the amount to increment by for Runge-Kutta
    ======
    returns:
    x_new: value of the state vector at the next timestep
    '''
    k1 = dt * dxdt(t, x)
    k2 = dt * dxdt(t + dt/2.0, x + k1/2.0)
    k3 = dt * dxdt(t + dt/2.0, x + k2/2.0)
    k4 = dt * dxdt(t + dt, x + k3)
    x_new = x + (k1 + 2.0*k2 + 2.0*k3 + k4)/6.0

    return x_new

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))

    print("\nSimulating:")
    for i in tqdm(range(N)):
        t = tvec[i]
        xtraj[:,i]=integrate(f,x,t,dt)
        x = np.copy(xtraj[:,i])
    return xtraj

def compute_lagrangian():

    #define kinetic and potential energy
    KE_B1 = 0.5 * (VbB1.T @ inertia_B1 @ VbB1)[0]
    KE_B2 = 0.5 * (VbB2.T @ inertia_B2 @ VbB2)[0]
    U = m*g*(ym1 + ym2)

    lagrangian = KE_B1 + KE_B2 - U
    return lagrangian

def compute_solve_EL(F_mat):
    '''
    Encapsulate the solving process in a function for the sake of
    not having to run this again every time. For the sake of reducing
    stack calls, could put this in its own file instead.
    '''

    #calculate forced Euler-Lagrange equations. no forces of constraint
    qd = q.diff(t)
    qdd = qd.diff(t)

    lagrangian = compute_lagrangian()
```

```
    lhs = compute_EL_lhs(lagrangian, q, t)
    RHS = sym.zeros(len(lhs), 1)
    RHS = RHS + F_mat

    lhs = lhs.subs(subs_dict) #defined in geometry.py
    total_eq = sym.Eq(lhs, RHS)

    #do symbolic substitutions before solving to speed up computation
    print("\nPress any key to simplify the E-L equations. ", end='')
    input()
    print("Simplifying:")

    #waited on all simplify() calls until here
    t0 = time.time()
    total_eq_simpl = total_eq.simplify()
    tf = time.time()

    print(f"\ntotal_eq.simplify(): \nElapsed: {round(tf - t0, 2)} seconds" )

    #attempt to round near-zero values to zero. source: https://tinyurl.com/f7t8wbmw
    total_eq_rounded = total_eq_simpl
    for a in sym.preorder_traversal(total_eq_simpl):
        if isinstance(a, sym.Float):
            total_eq_rounded = total_eq_rounded.subs(a, round(a, 8))

    print("Euler-Lagrange equations - simplified:")
    #display(total_eq_simpl)
    print("Euler-Lagrange equations - rounded:")
    #display(total_eq_rounded)

    print("\nPress any key to solve the E-L equations. ", end='')
    input()
    print("Solving:")

    t0 = time.time()
    #soln = sym.solve(total_eq, qdd, dict = True, simplify = False, manual = True)
    #soln = sym.solve(total_eq, qdd, dict = True, manual = True)
    soln = sym.solve(total_eq_rounded, qdd, dict = True, simplify = False)

    tf = time.time()
    print(f"\nsym.solve(): \nElapsed: {round(tf - t0, 2)} seconds" )

    eqns_solved = format_solns(soln)

    #simplify equations one by one
    eqns_new = []
    print("\nSimplifying EL equations:")
```

```python
    for eq in tqdm(eqns_solved):
        eq_new = sym.simplify(eq)
        eqns_new.append(eq_new)

    return eqns_new

def construct_dxdt(f_eqs_array):
    '''Generates our dynamics function dxdt() using the
    second-derivative equations derived from the Euler-Lagrange
    equations.

    Arguments:
    - f_eqs_array: an array of Numpy functions, lambda functions, or
        other univatiate functions of time

    Returns: dxdt, a function f(t,s)
    '''
    F_mat = sym.Matrix([
        sym.symbols(r'F_x'),
        sym.symbols(r'F_y'),
        sym.symbols(r'F_\theta1'),
        sym.symbols(r'F_\theta2'),
        sym.symbols(r'F_\phi1'),
        sym.symbols(r'F_\phi2'),
    ])

    eqns_new = dill_load('../dill/EL_simplified.dill')
    q_ext = sym.Matrix([q, q.diff(t), F_mat])

    #lambdify the second derivative equations and construct dynamics function
    xdd_sy      = eqns_new[0].rhs
    ydd_sy      = eqns_new[1].rhs
    theta1dd_sy = eqns_new[2].rhs
    theta2dd_sy = eqns_new[3].rhs
    phi1dd_sy   = eqns_new[4].rhs
    phi2dd_sy   = eqns_new[5].rhs

    xdd_np      = sym.lambdify(q_ext,      xdd_sy)
    ydd_np      = sym.lambdify(q_ext,      ydd_sy)
    theta1dd_np = sym.lambdify(q_ext, theta1dd_sy)
    theta2dd_np = sym.lambdify(q_ext, theta2dd_sy)
    phi1dd_np   = sym.lambdify(q_ext,   phi1dd_sy)
    phi2dd_np   = sym.lambdify(q_ext,   phi2dd_sy)


    def dxdt(t,s):

        F_array = [f(s,t) for f in f_eqs_array]
        s_ext = np.append(s, F_array)
```

```
        #format of s_ext:
        #0-5: state values
        #6-11: values of derivative of state
        #12-17: values of force at given time

        return np.array([
            *s[6:12],
            xdd_np(*s_ext),
            ydd_np(*s_ext),
            theta1dd_np(*s_ext),
            theta2dd_np(*s_ext),
            phi1dd_np(*s_ext),
            phi2dd_np(*s_ext),
        ])

    #return type is a function
    return dxdt

```

```
import numpy as np
import sympy as sym
import dill
import time
from tqdm import tqdm

from helpers import *

#define frames and symbols. let L1 = L2, m1 = m2 for computational efficiency
#as this condition is unlikely to change
L, w, m, g = sym.symbols(r'L, w, m, g')
t = sym.symbols(r't')

x = sym.Function(r'x')(t)
y = sym.Function(r'y')(t)
theta1 = sym.Function(r'\theta_1')(t)
theta2 = sym.Function(r'\theta_2')(t)
phi1 = sym.Function(r'\Phi_1')(t)
phi2 = sym.Function(r'\Phi_2')(t)

q = sym.Matrix([x, y, theta1, theta2, phi1, phi2])
q_ext = sym.Matrix([q, q.diff(t)])

subs_dict = {
    L : 1,
    w : 1/6.0,
    m : 1,
    g : 9.81,
}

#make sure geometry for plotting matches
#the Sympy substitution dict!
L_num = 1
w_num = 1/6.0

#--------symbolic transformation matrices, right side---------------#

Rab = sym.Matrix([
    [sym.cos(theta2), -sym.sin(theta2), 0],
    [sym.sin(theta2),  sym.cos(theta2), 0],
    [              0,                0, 1],
])

RdB2 = sym.Matrix([
    [sym.cos(phi2), -sym.sin(phi2), 0],
    [sym.sin(phi2),  sym.cos(phi2), 0],
    [            0,              0, 1],
])

```

```python
p_bd = sym.Matrix([0, -L, 0])

Gab = SOnAndRnToSEn(Rab, [0, 0, 0])
Gbd = SOnAndRnToSEn(sym.eye(3), p_bd)
GdB2 = SOnAndRnToSEn(RdB2, [0, 0, 0])

Gsa = SOnAndRnToSEn(sym.eye(3), [x, y, 0])
Gsb = Gsa @ Gab
Gsd = Gsb @ Gbd
GsB2 = Gsd @ GdB2 #formerly Gsf


#--------symbolic transformation matrices, left side---------------#

Rac = sym.Matrix([
    [sym.cos(theta1), -sym.sin(theta1), 0],
    [sym.sin(theta1),  sym.cos(theta1), 0],
    [              0,                0, 1],
])

ReB1 = sym.Matrix([
    [sym.cos(phi1), -sym.sin(phi1), 0],
    [sym.sin(phi1),  sym.cos(phi1), 0],
    [            0,              0, 1],
])

p_ce = sym.Matrix([0, -L, 0])

Gac = SOnAndRnToSEn(Rac, [0, 0, 0])
Gce = SOnAndRnToSEn(sym.eye(3), p_ce)
GeB1 = SOnAndRnToSEn(ReB1, [0, 0, 0])

Gsa = SOnAndRnToSEn(sym.eye(3), [x, y, 0])
Gsc = Gsa @ Gac
Gse = Gsc @ Gce
GsB1 = Gse @ GeB1 #formerly Gsg


#------------line + box geometry; plotting geometry------------------#

#Lnum and wnum defined under subs_dict

win_height = 600
win_width = 800
pixels_to_unit = 200
coordsys_len = 50

vertices_mat = np.array([
    [ w_num/2.0,  w_num/2.0, 0, 1],
```

```
    [-w_num/2.0,  w_num/2.0, 0, 1],
    [-w_num/2.0, -w_num/2.0, 0, 1],
    [ w_num/2.0, -w_num/2.0, 0, 1],
    [ w_num/2.0,  w_num/2.0, 0, 1], #add first vertex onto end of matrix again so line wraps around
]).T #in "bar" form  so they can be multiplied by trans. matrix

line_coords_mat = np.array([
    [0,      0, 0, 1],
    [0, -L_num, 0, 1],
]).T

#have these variables in global namespace
width  = win_width  // pixels_to_unit
height = win_height // pixels_to_unit

#let frame GUI be the coordinates as seen on the GUI,
#frame r be the frame at GUI coords (0,0) with axes in same direction
#as frame s. This is not in SE(3) so InvSEn() cannot be used with this.
GrGUI = np.array([
    [width/win_width,    0, 0, 0],
    [0, -height/win_height, 0, 0],
    [0,                  0, 1, 0],
    [0,                  0, 0, 1]
])

Grs = SOnAndRnToSEn(np.identity(3), [width/2, -height/2, 0])
GsGUI = np.dot(InvSEn(Grs), GrGUI)

#define important positions
ym1 = ( GsB1    @ sym.Matrix([0, 0, 0, 1]) )[1]
ym2 = ( GsB2    @ sym.Matrix([0, 0, 0, 1]) )[1]
posn_top = Gsa @ sym.Matrix([0, 0, 0, 1])


#----------------------------------------------#

#inertial properties of system, in symbolic form.
#reused in both Lagrangian and Hamiltonian calculation,
#so define it once

VbB1 = CalculateVb6(GsB1,t)
VbB2 = CalculateVb6(GsB2,t)

scriptI_B1 = m * sym.Matrix([
    [w**2,    0,      0],
    [   0, w**2,      0],
    [   0,    0, 2*w**2]
])
```

```python

scriptI_B2 = m * sym.Matrix([
    [w**2,    0,      0],
    [   0, w**2,      0],
    [   0,    0, 2*w**2]
])

inertia_B1 = InertiaMatrix6(m, scriptI_B1)
inertia_B2 = InertiaMatrix6(m, scriptI_B2)


```

```python
import tkinter as tk
import numpy as np

from geometry import *
from helpers import *
from impacts import *
from el_equations import *
from plotting_helpers import *

###########

#contains the symbolic impact equations - will be solved using nsolve() during sim
impact_eqns_0_32 = dill_load('../dill/impact_eqns_0_32.dill')

class GUI:

    def __init__(self, win_height, win_width):

        #future improvement: should inherit from the Tk class
        self.root = tk.Tk()
        self.root.title("Clacker Balls Simulation")
        self.canvas = tk.Canvas(self.root, bg="white", height=win_height, width=win_width)
        self.win_height = win_height
        self.win_width = win_width

        #member data we expect to change on each loop
        self.timer_handle = None #set this from the outside once canvas is packed
        self.last_frametime = 0
        self.q_ind = 0
        self.impact_photoID = None
        self.mouse_posn_gui = [win_width//2, win_height//2]
        self.mouse_posn_s = [0,0]

    ###

    #these values are determined externally, but loading them all in __init__
    #would be too long, so do it here

    def load_arrays(self, line_coords_mat, vertices_mat):
        self.line_coords_mat = line_coords_mat
        self.vertices_mat    = vertices_mat

    def load_gui_params(self, L, w, coordsys_len, GsGUI, framerate, photo_filepath):
        self.L = L
        self.w = w
```

```
        self.coordsys_len = coordsys_len
        self.GsGUI = GsGUI
        self.framerate_ms = framerate
        self.impact_photoID = draw_image(self.canvas, self.root, \
            (self.win_width//2, self.win_height//2), photo_filepath, size=0,
                  tags='sparks', state='hidden')

    def load_simulation(self, dxdt, t_span, dt, ICs, atol):

        #describe variables we're solving for - qd1_tau+, qd2_tau+, ... lambda
        sym_t = sym.symbols(r't')
        qd_q = sym.Matrix([sym.Matrix(q.diff(sym_t)), q])
        lamb = sym.symbols(r'\lambda')
        _, _, _, _, _, qd_taup_list = gen_sym_subs(q, qd_q)

        self.sol_vars = qd_taup_list
        self.sol_vars.append(lamb)

        #timescales and tolerance for checking if phi(q) near zero
        self.dt = dt
        self.t_array = np.arange(t_span[0], t_span[1], dt)
        self.traj_array = np.zeros([len(ICs), len(self.t_array)])
        self.traj_array[:,0] = ICs
        self.dxdt = dxdt
        self.atol = atol

    #------------------------------------#

    def get_GUI_coords(self, q):
        '''
        Takes the present value of the state array and returns the
        coordinates of the key items on the GUI: the coords of
        the lines for the two strings, and the coords of the boxes
        for the two masses.

        Arguments:
        - q: current value of extended state array [q; qdot]
        - line_coords_mat: a 4xn array, n = 2 points per line,
            with the coordinates of lines in their reference frames
        - vertices_mat: a 4x5 array (4 vertices per box, plus the initial
            coordinate repeated) with coordinates of vertices of the boxes
            in their reference frames
        - GsGUI: transformation of points from space frame to GUI frame
            (note: not SE(3) - scaling + mirroring operations)
        - L: length of string
        - w: width of box

        Returns:
        - box1_vert_gui:    cods of object in GUI frame
```

```
        - box2_vert_gui:    coords of object in GUI frame
        - line1_coords_gui: coords of object in GUI frame
        - line2_coords_gui: coords of object in GUI frame
        '''

        #extract coords
        x, y, theta1, theta2, phi1, phi2 = q[0:6]

        #define frames

        #---------------right side---------------#

        Rab = np.array([
            [np.cos(theta2), -np.sin(theta2), 0],
            [np.sin(theta2),  np.cos(theta2), 0],
            [              0,                0, 1],
        ])

        RdB2 = np.array([
            [np.cos(phi2), -np.sin(phi2), 0],
            [np.sin(phi2),  np.cos(phi2), 0],
            [            0,              0, 1],
        ])

        p_bd = np.array([0, -self.L, 0])

        Gab = SOnAndRnToSEn(Rab, [0, 0, 0])
        Gbd = SOnAndRnToSEn(np.eye(3), p_bd)
        GdB2 = SOnAndRnToSEn(RdB2, [0, 0, 0])

        Gsa = SOnAndRnToSEn(np.eye(3), [x, y, 0])
        Gsb = Gsa @ Gab
        Gsd = Gsb @ Gbd
        GsB2 = Gsd @ GdB2 #formerly Gsf


        #---------------left side---------------#

        Rac = np.array([
            [np.cos(theta1), -np.sin(theta1), 0],
            [np.sin(theta1),  np.cos(theta1), 0],
            [             0,                0, 1],
        ])

        ReB1 = np.array([
            [np.cos(phi1), -np.sin(phi1), 0],
            [np.sin(phi1),  np.cos(phi1), 0],
            [           0,              0, 1],
        ])
```

```python

        p_ce = np.array([0, -self.L, 0])

        Gac = SOnAndRnToSEn(Rac, [0, 0, 0])
        Gce = SOnAndRnToSEn(np.eye(3), p_ce)
        GeB1 = SOnAndRnToSEn(ReB1, [0, 0, 0])

        Gsa = SOnAndRnToSEn(np.eye(3), [x, y, 0])
        Gsc = Gsa @ Gac
        Gse = Gsc @ Gce
        GsB1 = Gse @ GeB1 #formerly Gsg

        #make objects in the frames of interest - home frame --> s frame
        line1_coords_s  = np.dot(Gsc,  self.line_coords_mat)
        line2_coords_s  = np.dot(Gsb,  self.line_coords_mat)
        box1_vertices_s = np.dot(GsB1, self.vertices_mat)
        box2_vertices_s = np.dot(GsB2, self.vertices_mat)

        #-----------#

        #convert object positions into the frame of the canvas
        box1_vert_gui    = np.dot(np.linalg.inv(self.GsGUI), box1_vertices_s)[0:2, :]
        box2_vert_gui    = np.dot(np.linalg.inv(self.GsGUI), box2_vertices_s)[0:2, :]
        line1_coords_gui = np.dot(np.linalg.inv(self.GsGUI),  line1_coords_s)[0:2, :]
        line2_coords_gui = np.dot(np.linalg.inv(self.GsGUI),  line2_coords_s)[0:2, :]

        #turn line/box coords into lists of [x1, y1, x2, y2, ...]
        box1_vert_gui    = (   box1_vert_gui.T.flatten() ).astype( int)
        box2_vert_gui    = (   box2_vert_gui.T.flatten() ).astype( int)
        line1_coords_gui = (line1_coords_gui.T.flatten() ).astype( int)
        line2_coords_gui = (line2_coords_gui.T.flatten() ).astype( int)

        return box1_vert_gui, \
               box2_vert_gui, \
               line1_coords_gui, \
               line2_coords_gui

    #----------------------------------#

    #event handlers

    def on_mouse_over(self, event):
        self.canvas.coords('user_posx',
                     event.x, event.y,
                     event.x + self.coordsys_len, event.y)
```

```
        self.canvas.coords('user_posy',
                     event.x, event.y,
                     event.x, event.y - self.coordsys_len)
        self.mouse_posn_gui = [event.x, event.y]

        #calculate position of user in s frame
        mouse_posn_guibar = np.array([event.x, event.y, 0, 1])
        self.mouse_posn_s = np.dot(GsGUI, mouse_posn_guibar)[0:2]

    def close(self):
        try:
            self.root.quit()
            self.root.destroy()
        except:
            pass


    def on_frame(self):
        ''' Animation update event, passed to the Tkinter canvas. Uses real-time
        data being collected and processed using the dxdt() function and the impact
        handling functions.
        '''

        #compare current real time to previous
        elapsed = time.perf_counter() - self.last_frametime
        elapsed_ms = int(elapsed*1000)
        prev_impact = False

        #elapsed time is a fraction of the total framerate in ms
        frame_delay = self.framerate_ms - elapsed_ms

        #---------------------#

        #things to be updated on each frame
        if self.q_ind < (max(self.traj_array.shape) - 1):

            #get current value of s
            t = self.t_array[self.q_ind]
            s = self.traj_array[:,self.q_ind]

            #calculate s for next timestep, and check for impact
            s_next = rk4(self.dxdt, s, t, self.dt)
            impact_dt, impact_indices  = impact_condition(s_next[0:6])

            #GUI plotting variables
            box1_vert_gui, box2_vert_gui, line1_coords_gui, line2_coords_gui = \
```

```
                    self.get_GUI_coords(s)

                if (impact_dt):
                    '''This is designed to alter the velocity of the particle
                    just before impact. If we applied the impact update after impact
                    (same position, changed velocity), there's a chance the objects
                    would stay stuck inside each other.
                    '''

                    #find phi(q) we can apply to the system. choose one to apply
                    any_nearzero, phi_indices, phi_arr_np = phi_nearzero(s_next[0:6], self.atol)
                    valid_phiq_indices, argmin = filter_phiq(impact_indices, phi_indices, phi_arr_np)

                    #this is a case I eventually want to figure out
                    if len(valid_phiq_indices) == 0:
                        print("Invalid phi(q)/impact condition combination") #throw an error in the future

                    else: #valid case

                        #for fun: plot a little "spark" every time objects collide
                        body_num = (argmin//16)+1
                        vertex_ind = (argmin%16)//4
                        if body_num == 1:
                            sparks_coords = box1_vert_gui[2*vertex_ind : 2*vertex_ind + 2]
                        elif body_num == 2:
                            sparks_coords = box2_vert_gui[2*vertex_ind : 2*vertex_ind + 2]

                        self.canvas.coords('sparks', *sparks_coords)
                        make_visible(self.canvas, self.impact_photoID)
                        self.root.update_idletasks()
                        impact_eqs = impact_eqns_0_32[argmin]

                        #solve for next state, using numerical nsolve() on symbolic expressions
                        s_alt = impact_update(s, impact_eqs, self.sol_vars)
                        s_next = rk4(self.dxdt, s_alt, t, self.dt)

                #apply update to trajectory vector
                self.traj_array[:, self.q_ind+1] = s_next
                prev_impact = impact_dt
```

```
        #------------------GUI UPDATES-----------------------#

        if prev_impact:
            make_invisible(self.canvas, self.impact_photoID)

        #apply updates to object posns
        if self.q_ind == 0:
            #create objects on the canvas
            linewidth = 2
            self.canvas.create_line(*box1_vert_gui,    tag='box1', fill='black', width=linewidth)
            self.canvas.create_line(*box2_vert_gui,    tag='box2', fill='black', width=linewidth)
            self.canvas.create_line(*line1_coords_gui, tag='line1', fill='blue', width=linewidth)
            self.canvas.create_line(*line2_coords_gui, tag='line2', fill='red', width=linewidth)

        else:
            #update positions of the objects by tags
            self.canvas.coords('box1', *box1_vert_gui)
            self.canvas.coords('box2', *box2_vert_gui)
            self.canvas.coords('line1', *line1_coords_gui)
            self.canvas.coords('line2', *line2_coords_gui)

        #see plotting_helpers.py
        label_vertices(self.canvas, box1_vert_gui, box2_vert_gui)
        self.q_ind += 1

        #---------------------#

        #update the frame delay of the timer object
        self.timer_handle = self.root.after(frame_delay, self.on_frame)

        #update last_frametime for next frame
        self.last_frametime = time.perf_counter()

```

```python
import numpy as np
import sympy as sym
import dill
import time
from tqdm import tqdm

from geometry import *

#-----------GEOMETRIC FUNCTIONS-----------#

def SOnAndRnToSEn(R, p):

    #do type checking for the matrix types
    if type(R) == list:
        R = np.matrix(R)

    n = R.shape[0]
    if ((R.shape[0] != R.shape[1]) or                                  #R is NP array or Sym matrix
        ((type(p) is np.ndarray and max(p.shape) != R.shape[0]) or   #p is NP array and shape mismatch or..
         ((isinstance(p, list) or isinstance(p, sym.Matrix)) and
          ( len(p) != R.shape[0] ))   )  ):                           #p is Sym matrix or "list" and shape mismatch
        raise Exception(f"Shape of R {R.shape} and p ({len(p)}) mismatch; exiting.")
        return None

    #construct a matrix based on returning a Sympy Matrix
    if isinstance(R, sym.Matrix) or isinstance(p, sym.Matrix):
        #realistically one of these needs to be symbolic to do this

        if isinstance(R, np.ndarray) or isinstance(p, np.ndarray):
            raise Exception("R and p cannot mix/match Sympy and Numpy types")
            return None

        G = sym.zeros(n+1)
        G[:n, n] = sym.Matrix(p)

    #construct a matrix based on returning a Numpy matrix
    elif isinstance(R, np.ndarray) or isinstance(R, list):
        G = np.zeros([n+1, n+1])
        # print(f"\nSOnAndRnToSEn Debug: \n\nR:\n{R}    \n\np:\n{p}   ")
        G[:n, n] = np.array(p).T

    else:
        raise Exception("Error: type not recognized")
        return None

```

```
    G[:n,:n] = R
    G[-1,-1] = 1
    return G

def SEnToSOnAndRn(SEnmat):
    '''Decomposes a SE(n) vector into its rotation matrix and displacement components.
    '''
    if isinstance(SEnmat, list):
        SEnmat = np.matrix(SEnmat)
    n = SEnmat.shape[0]
    return SEnmat[:(n-1), :(n-1)], SEnmat[:(n-1), n-1]

def HatVector3(w):
    '''Turns a vector in R3 to a skew-symmetric matrix in so(3).
    Works with both Sympy and Numpy matrices.
    '''
    #create different datatype representations based on type of w
    if isinstance(w, list) or isinstance(w, np.ndarray) \
        or isinstance(w, np.matrix):
        f = np.array
    elif isinstance(w, sym.Matrix): #NP and Sym
        f = sym.Matrix

    return f([
        [    0, -w[2],  w[1]],
        [ w[2],     0, -w[0]],
        [-w[1],  w[0],     0]
    ])

def UnhatMatrix3(w_hat):
    '''Turns a skew-symmetric matrix in so(3) into a vector in R3.
    '''
    if isinstance(w_hat, list) or isinstance(w_hat, np.ndarray) \
        or isinstance(w_hat, np.matrix):
        f = np.array
        w_hat = np.array(w_hat)
    elif isinstance(w_hat, sym.Matrix) or isinstance(w_hat, sym.ImmutableMatrix):
        f = sym.Matrix
    else:
        raise Exception(f"UnhatMatrix3: Unexpected type of w_hat: {type(w_hat)}")

    #matrix checking, for use in potential debug. generalized to both Sympy and Numpy
    same = np.array([w_hat + w_hat.T == f([
        [0, 0, 0],
        [0, 0, 0],
```

```
        [0, 0, 0]
    ]
    ) ] )

#     if (not same.all()):
#         raise Exception("UnhatMatrix3: w_hat not skew_symmetric")

    #NP and Sym
    return f([
        -w_hat[1,2],
        w_hat[0,2],
        -w_hat[0,1],
    ])

def InvSEn(SEnmat):
    '''Takes the inverse of a SE(n) matrix.
    Compatible with Numpy, Sympy, and list formats.
    '''
    if isinstance(SEnmat, list):
        SEnmat = np.matrix(SEnmat)
    ###
    n = SEnmat.shape[0]
    R = SEnmat[:(n-1), :(n-1)]
    p = SEnmat[:(n-1),   n-1 ]

    return SOnAndRnToSEn(R.T, -R.T @ p)

def InertiaMatrix6(m, scriptI):
    '''Takes the mass and inertia matrix properties of an object in space,
    and constructs a 6x6 matrix corresponding to [[mI 0]; [0 scriptI]].
    Currently only written for Sympy matrix representations.
    '''
    if (m.is_Matrix or not scriptI.is_square):
        raise Exception("Type error: m or scriptI in InertiaMatrix6")

    mat = sym.zeros(6)
    mI = m * sym.eye(3)
    mat[:3, :3] = mI
    mat[3:6, 3:6] = scriptI
    return mat

def HatVector6(vec):
    '''Convert a 6-dimensional body velocity into a 4x4 "hatted" matrix,
    [[w_hat v]; [0 0]], where w_hat is skew-symmetric.
    w =
    '''
    if isinstance(vec, np.matrix) or isinstance(vec, np.ndarray):
        vec = np.array(vec).flatten()

```

```
    v = vec[:3]
    w = vec[3:6]

    #this ensures if there are symbolic variables, they stay in Sympy form
    if isinstance(vec, sym.Matrix):
        v = sym.Matrix(v)
        w = sym.Matrix(w)

    w_hat = HatVector3(w)

    #note that the result isn't actually in SE(3) but
    #that the function below creates a 4x4 matrix from a 3x3 and
    #1x3 matrix - with type checking - so we'll use it
    mat = SOnAndRnToSEn(w_hat, v)
    return mat

def UnhatMatrix4(mat):
    '''Convert a 4x4 "hatted" matrix,[[w_hat v]; [0 0]], into a 6-dimensional
    body velocity [v, w].
    '''
    #same as above - matrices aren't SE(3) and SO(3) but the function
    #can take in a 4x4 mat and return a 3x3 and 3x1 mat
    [w_hat, v] = SEnToSOnAndRn(mat)
    w = UnhatMatrix3(w_hat)

    if (isinstance(w, np.matrix) or isinstance(w, np.ndarray)):
        return np.array([v, w]).flatten()
    elif isinstance(w, sym.Matrix):
        return sym.Matrix([v, w])
    else:
        raise Exception("Unexpected datatype in UnhatMatrix4")

def CalculateVb6(G,t):
    '''Calculate the body velocity, a 6D vector [v, w], given a trans-
    formation matrix G from one frame to another.
    '''
    G_inv = InvSEn(G)
    Gdot = G.diff(t) #for sympy matrices, this also carries out chain rule
    V_hat = G_inv @ Gdot

#     if isinstance(G, sym.Matrix):
#         V_hat = sym.simplify(V_hat)

    return UnhatMatrix4(V_hat)

#-----------EULER-LAGRANGE AND IMPACTS-----------#

def compute_EL_lhs(lagrangian, q, t):
    '''
```

```
    Helper function for computing the Euler-Lagrange equations for a given system,
    so I don't have to keep writing it out over and over again.

    Inputs:
    - lagrangian: our Lagrangian function in symbolic (Sympy) form
    - q: our state vector [x1, x2, ...], in symbolic (Sympy) form

    Outputs:
    - eqn: the Euler-Lagrange equations in Sympy form
    '''

    # wrap system states into one vector (in SymPy would be Matrix)
    #q = sym.Matrix([x1, x2])
    qd = q.diff(t)
    qdd = qd.diff(t)

    # compute derivative wrt a vector, method 1
    # wrap the expression into a SymPy Matrix
    L_mat = sym.Matrix([lagrangian])
    dL_dq = L_mat.jacobian(q)
    dL_dqdot = L_mat.jacobian(qd)

    #set up the Euler-Lagrange equations
    #LHS = dL_dq - dL_dqdot.diff(t)
    LHS = dL_dqdot.diff(t) - dL_dq

    return LHS.T

def format_solns(soln):
    eqns_solved = []
    #eqns_new = []

    for i, sol in enumerate(soln):
        for x in list(sol.keys()):
            eqn_solved = sym.Eq(x, sol[x])
            eqns_solved.append(eqn_solved)

    return eqns_solved

def decompose_factors_dict(factors_dict):
    '''Take the dictionary of factors in the impact equations, and breaks
    them down further. This process can be repeated to get the factors only
    in terms of sines, cosines, numbers, and symbolic variables.

    Returns: new_factors_dict. Contains the same data as factors_dict
        in smaller terms.
    '''
    new_factors_array = np.array([])
```

```python
    new_factors_dict = factors_dict.copy()

    for factor in factors_dict.keys():
        if factor.is_Add:
            #add components to list of factors and remove from old dictionary
            new_factors_array = np.append(new_factors_array, factor.as_ordered_terms())
            del new_factors_dict[factor]

        if factor.is_Pow:
            new_factors_array = np.append(new_factors_array, list(factor.as_powers_dict().keys()))
            del new_factors_dict[factor]

        if factor.is_Mul:
            new_factors_array = np.append(new_factors_array, list(factor.as_coeff_mul()[-1]) )
            del new_factors_dict[factor]

    #fdo data checking and add terms back into the dictionary
    for factor in new_factors_array:
        if factor in new_factors_dict.keys():
            new_factors_dict[factor] += 1
        else:
            new_factors_dict[factor] = 1

    return new_factors_dict

#-----------DATA SAVING FUNCTIONS-----------#


def dill_dump(filename, data):
    dill.settings['recurse'] = True
    with open(filename, 'wb') as f:
        dill.dump(data, f)

def dill_load(filename):
    dill.settings['recurse'] = True
    with open(filename, 'rb') as f:
        data = dill.load(f)
    return data

```

```python
import numpy as np
import sympy as sym
import pandas as pd

import dill
import time
from tqdm import tqdm

from geometry import *
from helpers import *
from el_equations import *


##GLOBAL VARIABLES
# there are some variables that are defined after the functions,
# as the variables depend on the functions
lamb = sym.symbols(r'\lambda')

def calculate_sym_vertices():
    '''
    Calculates 16 symbolic expressions to describe the vertices of the
    2 boxes in the system - 4 vertices * 2 coords(x,y) * 2 boxes.

    This is a moderately time-consuming operation (3min) so the output
    will be saved to a file to prevent losing data.

    Returns: 16 symbolic expressions for vijn_Bk, where i = 1-2 (box#),
    j = 1-4 (vertex#), k = 2-1 (opposite of i #)
    '''

    #define positions of vertices in boxes' home frames
    v1bar = sym.Matrix([ w/2,  w/2, 0, 1])
    v2bar = sym.Matrix([-w/2,  w/2, 0, 1])
    v3bar = sym.Matrix([-w/2, -w/2, 0, 1])
    v4bar = sym.Matrix([ w/2, -w/2, 0, 1])

    #from geometry.py
    GB1B2 = InvSEn(GsB1) @ GsB2
    GB2B1 = InvSEn(GB1B2)

    vbar_list = [v1bar, v2bar, v3bar, v4bar]
    g_list = [GB2B1, GB1B2]

    #do this algorithmically so we can wrap it in a tqdm; track progress
    vertices_coords_list = []
    print("Calculate_sym_vertices(): simplifying vertex coords." )
    for i in tqdm(range(8)):
        #G: 00001111, vbar: 01230123
        vij_Bk = sym.simplify(g_list[i//4] @ vbar_list[i%4])
```

```python
        vertices_coords_list.append(vij_Bk)

    #save results
    print('\nSaving results:')
    filepath = '../dill/vertices_coords_list.dill'
    dill_dump(filepath, vertices_coords_list)
    print(f"Vertices coords saved to {filepath}.")

def convert_coords_to_xy():
    '''Take the symbolic coordinates we found for the two boxes and
    split them into x and y components.
    '''
    vertices_coords_list = dill_load('../dill/vertices_coords_list.dill')
    vertices_xy_list = []
    for coord in vertices_coords_list:
        coordx, coordy, _, _ = coord
        vertices_xy_list.append([coordx, coordy])
        pass

    #flatten
    vertices_list_sym = np.array(vertices_xy_list).flatten().tolist()
    return vertices_list_sym

def calculate_sym_phiq(vlist):
    ''' Calculate the symbolic impact equations Phi(q) for use in
    applying impact updates to the system. There will be 32
    phi(q) equations - 2 per possible vertex + side of impact
    combination.

    Saves symbolic phi(q) to a pickled file for loading
    and use in other files, plus saving variables between sessions
    of Python/Jupyter notebook.

    Arguments:
    vlist - a list of symbolic vertex coordinates broken apart
        into x and y

    Returns: None; load symbolic phi(q) from file for simplicity
    '''

    phiq_list = []
    for vertex in vlist:
        phiq_list.append(vertex + w/2) #impact from left side
        phiq_list.append(vertex - w/2) #impact from right side

    #substitute in values of L and w
    phiq_list = [expr.subs(subs_dict) for expr in phiq_list]
    return phiq_list

```

```
def impact_condition(s):
    '''Contains and evaluates an array of impact conditions for the current
    system, at the current state s.

    Returns: a logical true/false as to whether any impact condition was met;
        list of indices of impact conditions that were met
    '''

    v11x_B2_np, v11y_B2_np, v12x_B2_np, v12y_B2_np, \
    v13x_B2_np, v13y_B2_np, v14x_B2_np, v14y_B2_np, \
    v21x_B1_np, v21y_B1_np, v22x_B1_np, v22y_B1_np, \
    v23x_B1_np, v23y_B1_np, v24x_B1_np, v24y_B1_np  = vertices_list_np

    #define tolerance for impact condition
    #ctol = 1/24.0 #proportional to w/2
    #bound = w_num/2.0 + ctol
    bound = w_num/2.0

    impact_conds = np.array([
        -bound < v11x_B2_np(*s) < bound   and   -bound < v11y_B2_np(*s) < 
          bound,
        -bound < v12x_B2_np(*s) < bound   and   -bound < v12y_B2_np(*s) < 
          bound,
        -bound < v13x_B2_np(*s) < bound   and   -bound < v13y_B2_np(*s) < 
          bound,
        -bound < v14x_B2_np(*s) < bound   and   -bound < v14y_B2_np(*s) < 
          bound,

        -bound < v21x_B1_np(*s) < bound   and   -bound < v21y_B1_np(*s) < 
          bound,
        -bound < v22x_B1_np(*s) < bound   and   -bound < v22y_B1_np(*s) < 
          bound,
        -bound < v23x_B1_np(*s) < bound   and   -bound < v23y_B1_np(*s) < 
          bound,
        -bound < v24x_B1_np(*s) < bound   and   -bound < v24y_B1_np(*s) < 
          bound,
    ])

    #find any impact conditions that have been met
    impact_met = np.any(impact_conds)
    impact_indices = np.nonzero(impact_conds)[0].tolist() #indices where true

    return impact_met, impact_indices

def phi_nearzero(s, atol):
    '''Takes in the current system state s, and returns the indices of
    impact conditions phi(q) that are close to 0 at the given instant in
    time.

```

```
    The results of this function will then be used to determine which of
    the symbolically solved impact updates will be applied to the system.

    Parameters:
    - s: current state of system: x, y, theta1, theta2, phi1, phi2.
    - atol: absolute tolerance for "close to zero", passed to np.isclose().
        Example: atol = 1E-4

    Returns:
    - a list of indices of phi that are near zero at the given time.
    '''

    #apply upper and lower bound condition to all vertices, in x and y directions
    phi_arr_np = np.array([])
    for i in np.arange(0, len(vertices_list_np)):
        phi_arr_np = np.append(phi_arr_np, vertices_list_np[i](*s) + w_num/2.0)
        phi_arr_np = np.append(phi_arr_np, vertices_list_np[i](*s) - w_num/2.0)

    #we're interested in which of the phi conditions were evaluated at close to 0, so return the
    #list of indices close to 0
    closetozero = np.isclose( phi_arr_np, np.zeros(phi_arr_np.shape), atol=atol )
    any_nearzero = np.any(closetozero) #logical T/F
    phi_indices = np.nonzero(closetozero)[0].tolist() #locations where T/F

    return any_nearzero, phi_indices, phi_arr_np

def filter_phiq(impact_indices, phi_indices, phi_arr_np):
    '''Simultaneous impact must be considered for this project, as the
    user interaction means initial conditions cannot be pre-set such that
    no simultaneous impacts occur.

    In the case of simultaneous impact of 2 cubes of the same size, there are
    potential phi(q) for indices impacting walls that approach zero even when
    no impact is occuring at those vertices. Ex: for an exact head-on collision
    of two blocks [ ][ ], the top left vertex of box 1 is "at" the vertical boundary
    of the second box, even though no impact is occurring.

    This function filters the indices of phi(q) that are near zero and returns only
    the indices of phi(q) near zero that correspond to impact conditions (evaluated in
    impact_indices) that have been satisfied.

    Args:
    - phi_indices    (NP array): passed from phi_nearzero()
```

```
    - impact_indices (NP array): passed from impact_condition()

    Returns:
    - valid_phiq     (NP array): a subset of phi_indices that corresponds to an
                                    element of impact_indices
    '''
    phi_indices = np.array(phi_indices)
    impact_indices = np.array(impact_indices)
    inds = np.in1d(phi_indices//4, impact_indices) #evaluates whether elements of phi_indices
                                                     #are related to an impact condition, T/F
    c = np.array(np.nonzero(inds)[0]) #turns locations of these True values to indices
    valid_phiq_indices = phi_indices[c]

    #find location of min valid phi(q)
    valid_phiq = phi_arr_np[valid_phiq_indices]
    argmin = valid_phiq_indices[np.argmin(abs(valid_phiq))]

    #returns the phi(q) equations that both evaluate to ~0 and
    #are related to an impact condition that has been met.
    #returns location of the min |phi(q)| as well, for knowing which one to apply
    return valid_phiq_indices, argmin


def gen_sym_subs(q, qd_q):
    '''
    Makes three sets of symbolic variables for use in the impact equations.
    Inputs:
    - q: our state vector. ex: [theta1 theta2 theta3]
    - qd_q: our state vector, plus velocities. must have velocities first.
        ex: [theta1d theta2d theta3d theta1 theta2 theta3]

    Returns:
    - q_subs: a dictionary of state variables and their "q_1" and "qd_1"
        representations for use in calculation of the impact symbolic equations
    - q_taup_subs: a dictionary that can replace "q_1" and "qd_1" with
        "q_1^{tau+}" and "qd_1^{tau+}" for solving for the impact update
    - q_taum_subs: ^same as above, but for tau-minus
    '''

    #enforce that qd_q, which might get confused for q_ext, has derivatives first
    #and other state variables second
    qd_q = sym.Matrix(qd_q).reshape(1, len(qd_q)).tolist()[0]
    for i in range(len(qd_q)-1):
```

```
        curr = qd_q[i]
        next = qd_q[i+1]
        if not curr.is_Derivative and next.is_Derivative:
            raise Exception("Gen_sym_subs(): qd_q must have derivatives first")

    #create symbolic substitutions for each element in state array
    sym_q_only = [sym.symbols(f"q_{i+1}") for i in range(len(q))]
    sym_qd = [sym.symbols(f"qd_{i+1}") for i in range(len(q))]
    sym_q = sym_qd + sym_q_only

    # - Define substitution dicts for q at tau+ and q at tau-. We may need
    #the list form as well for later substitutions, so return that as well.
    q_taum_list   = [sym.symbols(f"q_{i+1}")    for i in range(len(q))]
    qd_taum_list  = [sym.symbols(f"qd_{i+1}^-") for i in range(len(q))]
    qd_taup_list  = [sym.symbols(f"qd_{i+1}^+") for i in range(len(q))]

    q_state_dict  = {qd_q[i]: sym_q[i]          for i in range(len(qd_q))}
    qd_taum_dict  = {sym_q[i] : qd_taum_list[i] for i in range(len(q))}
    qd_taup_dict  = {sym_q[i] : qd_taup_list[i] for i in range(len(q))}

    return q_state_dict, qd_taum_dict, qd_taup_dict, \
            q_taum_list, qd_taum_list, qd_taup_list

def impact_symbolic_eqs(phi, lagrangian, q, q_subs):
    '''Takes the impact condition phi, Lagrangian L, and state vector
    q, and returns the expressions we use to evaluate for impact.

    Returns, in order: dL_dqdot, dphi_dq, (dL_dqdot * qdot) - L(q,qdot),
    '''
    t = sym.symbols(r't')
    qd = q.diff(t)

    #define dL_dqdot before substitution
    L_mat = sym.Matrix([lagrangian])
    dL_dqd = L_mat.jacobian(qd)

    #define dPhi/dq before substitution
    phi_mat = sym.Matrix([phi])
    dphi_dq = phi_mat.jacobian(q)

    #define third expression
    dL_dqd_dot_qd = dL_dqd.dot(qd)
    expr3 = dL_dqd_dot_qd - lagrangian

    '''
    at this point the equations are in terms of the
    state variables, x,y, theta1, ...

    convert them into simplified versions "q1, q2, q3, ..."
```

```
    for ease of computing the difference between q_tau+ and q_tau-
    '''
    expr_a = dL_dqd.subs(q_subs)
    expr_b = dphi_dq.subs(q_subs)
    expr_c = expr3.subs(q_subs)

    return [expr_a, expr_b, expr_c]

def gen_impact_eqns(phiq_list_sym, lagrangian, q, const_subs):
    '''Methodically calculate all the possible impact updates
    for the two boxes, using the impact equations derived in class.

    Arguments:
    - phiq_list: 32x0 list of symbolic equations for possible impacts
    - lagrangian: symbolic Lagrangian
    - q: state vector, 6x1 Sympy Matrix
    - const_subs: dictionary of substitutions for m, g, L, w
    '''
    lamb = sym.symbols(r'\lambda')
    t = sym.symbols(r't')

    #qd_q is similar to q_ext, only derivatives come first so that
    #substitution works properly
    qd_q = sym.Matrix([sym.Matrix(q.diff(t)), q])

    #substitution dictionaries and lists for use in calculating impact
    #update equations
    q_state_dict, qd_taum_dict, qd_taup_dict, \
        q_taum_list, qd_taum_list, qd_taup_list = gen_sym_subs( q, qd_q)

    #NOTE: with 32 impact equations to solve, this takes a long time.
    #I ran this code in 4 separate Jupyter notebooks at once, each one finding
    #8 symbolic equations. still took 30min+

    impacts_eqns_list = []
    for phi in tqdm(phiq_list_sym):
        dL_dqd, dphi_dq, hamiltonian_term = \
            impact_symbolic_eqs(phi,lagrangian, q, q_state_dict)

        lamb_dphi_dq = lamb * dphi_dq

        #equations at tau+ minus equations at tau-
        dL_dqdot_eqn = \
            dL_dqd.subs(qd_taup_dict) \
            - dL_dqd.subs(qd_taum_dict) \
            - lamb_dphi_dq

        hamiltonian_eqn = \
            hamiltonian_term.subs(qd_taup_dict) \
```

```
            - hamiltonian_term.subs(qd_taum_dict) \

        #sub in m, g, L, w
        dL_dqdot_eqn    = dL_dqdot_eqn.subs(    const_subs)
        hamiltonian_eqn = hamiltonian_eqn.subs( const_subs)

        #these need to be simplified or else they're uninterpretable
        dL_dqdot_eqn = sym.simplify(dL_dqdot_eqn)
        hamiltonian_eqn = sym.simplify(hamiltonian_eqn)
        dL_dqdot_eqn = dL_dqdot_eqn.T

        eqns_matrix = dL_dqdot_eqn.row_insert( len(q), sym.Matrix([hamiltonian_eqn]))
        impacts_eqns_list.append(eqns_matrix)

    return impacts_eqns_list

def impact_update(s, impact_eqs, sol_vars):
    '''Once an impact has been detected, apply the necessary
    impact update based on which equation has just occurred.

    Args:
    - s: full state of system. Contains x, y, theta1, theta2,
        phi1, phi2, and their derivatives.
    - impact_eqs: the symbolic impact equations that need to be solved.
    - sol_vars: list of variables we're solving for. qd1_tau+, qd2_tau+, ..., lambda.
    '''
    curr_state_subs = {**{sym.symbols(f"q_{i+1}") : s[i] for i in range(6)},
                         **{sym.symbols(f"qd_{i+1}^-") : s[i+6] for i in range(6)}}
    impact_eqs_curr = impact_eqs.subs(curr_state_subs)

    attempts = 3
    init_guess = -1 * np.append(s[6:12],[0])
    solns_list = [0]*attempts
    lamb_val_arr = np.zeros(10)
    soln = None

    #credit to Jake for suggesting a multiple-start nsolve like this
    for i in range(attempts):
        try:
            curr_soln = sym.nsolve(impact_eqs_curr, sol_vars, init_guess, dict = True, verify = False)[0]
            solns_list[i] = curr_soln
            lamb_val_arr[i] = curr_soln[lamb]

        except Exception as e:
            print(f"Nsolve threw an error: {e}")
```

```
            init_guess = -1.5*init_guess

    ind = np.argmax(abs(lamb_val_arr))
    soln = solns_list[ind]
    lamb_val = lamb_val_arr[ind]

    #convert dictionary of solutions to a new state: posn of s, velocities from update
    if soln:
        del soln[lamb]
        qd_tauplus = np.array(list(soln.values())).astype('float')
        new_state = np.append(s[0:6], qd_tauplus )
        return new_state
    else:
        print("No solution found by nsolve")
        return s

#GLOBAL VARIABLES
xy_coords_list = convert_coords_to_xy()
vertices_list_np = [sym.lambdify(q, expr.subs(subs_dict)) for expr in xy_coords_list] #from geometry.py
```

```
import sympy as sym
import numpy as np
import pandas as pd

import dill
import time
from tqdm import tqdm
import tkinter as tk

from GUI import GUI
from geometry import *
from helpers import *
from plotting_helpers import *
from el_equations import *
from impacts import *

#-----------------tuning parameters----------------------------#

#time parameters
framerate_ms = 20
#dt = 0.005
dt = 0.01
t_span = [0, 30]
t_array = np.arange(t_span[0], t_span[1], dt)

theta0 = np.pi/4
init_posns = [0, 1, theta0, -theta0, np.pi/2, np.pi/4]
init_velocities = [0, 0, -1, 1, 7, -4]
ICs = init_posns + init_velocities

#spring and damping constants for PD control
#k = 30
k = 50
Bx = 2
By = 5

#tolerance for detecting if phi(q) near zero
atol = 5E-2

#----------------initialize GUI----------------------#

gui = GUI(win_height, win_width) #namespace for variables: geometry.py
gui.load_arrays(line_coords_mat, vertices_mat)    #geometry.py as well
gui.load_gui_params(L_num, w_num, coordsys_len, GsGUI,
                    framerate_ms, '../sprites/impact_sparks.png') #plotting_helpers.py

#-----------------forces and dxdt--------------------#

```

```
#forces use PD control in x and y to track a position
F_eqs_array = np.array([
    lambda s,t: k*(gui.mouse_posn_s[0] - s[0]) - Bx*s[6] , #F_x - damping term applied to vel.
    lambda s,t: k*(gui.mouse_posn_s[1] - s[1]) - By*s[7] + 19.62, #F_y
    lambda s,t: 0, #F_theta1
    lambda s,t: 0, #F_theta2
    lambda s,t: 0, #F_phi1
    lambda s,t: 0, #F_phi2
])

dxdt = construct_dxdt(F_eqs_array)
gui.load_simulation(dxdt, t_span, dt, ICs, atol)

#to make the particle follow a predetermined path instead, uncomment the following code.
#k = 30, dt = 0.005 was working pretty well with the path below.

##define trajectory for particle to follow
#y_tracking = lambda t: -np.sin(3*np.pi*t)+1
#x_tracking = lambda t: 0

##forces use PD control in x and y to track a position
#F_eqs_array = np.array([
#    lambda s,t: k*(x_tracking(t) - s[0]) - Bx*s[6] , #F_x - damping term applied to vel.
#    lambda s,t: k*(y_tracking(t) - s[1]) - By*s[7] + 19.62, #F_y
#    lambda s,t: 0, #F_theta1
#    lambda s,t: 0, #F_theta2
#    lambda s,t: 0, #F_phi1
#    lambda s,t: 0, #F_phi2
#])

#dxdt = construct_dxdt(F_eqs_array)
#gui.load_simulation(dxdt, t_span, dt, ICs, atol)


#----------------populate canvas----------------------#

s_frame       = make_coordsys(gui.canvas, win_width/2, win_height/2, coordsys_len, tag='s_frame')
make_grid(                    gui.canvas, win_width,   win_height, pixels_to_unit)
user_coordsys = make_coordsys(gui.canvas, win_width/2, win_height/2, coordsys_len, tag='user_pos')
s_frame =       make_coordsys(gui.canvas, win_width/2, win_height/2, coordsys_len, tag='s_frame')

#-------------canvas display-------------------#
```

```
gui.canvas.bind("<Motion>",           gui.on_mouse_over)
gui.root.protocol('WM_DELETE_WINDOW', gui.close) #forces closing of all Tk() functions
gui.canvas.pack()

gui.timer_id = gui.root.after(gui.framerate_ms, gui.on_frame)
gui.root.mainloop()
```

```python
#helper functions for GUI operations
import tkinter as tk
import time
import numpy as np
from PIL import Image, ImageTk
import os

from helpers import *
from geometry import *

def make_oval(canvas: tk.Canvas, center: tuple, width: int, height: int, fill: str='hotpink'):
    #from CS110; credit to Sarah Van Wart
    top_left = (center[0] - width, center[1] - height)
    bottom_right = (center[0] + width, center[1] + height)
    return canvas.create_oval([top_left, bottom_right], fill=fill, width=0) #return content ID

def make_circle(canvas: tk.Canvas, center: tuple, radius: int, fill: str='hotpink'):
    return make_oval(canvas, center, radius, radius, fill=fill) #return content ID

def make_grid_label(canvas, x, y, w, h, offset, pixels_to_unit):
    #from CS110; credit to Sarah Van Wart

    #apply offset by finding origin and applying conversion
    #from pixels to units in world
    width_world = w//pixels_to_unit
    height_world = h//pixels_to_unit

    origin_x = width_world//2
    origin_y = height_world//2

    xlabel, ylabel = (x/pixels_to_unit - origin_x, (h-y)/pixels_to_unit - origin_y - 0.5)

    #decide whether label is for x or y
    coord = xlabel if not xlabel == -origin_x else ylabel

    canvas.create_oval(
        x - offset,
        y - offset,
        x + offset,
        y + offset,
        fill='black'
    )
    canvas.create_text(
        x + offset,
```

```python
        y + offset,
        text=str(round(coord,1)),
        anchor="sw",
        font=("Purisa", 12)
    )

def make_grid(canvas, w, h, interval):
    #from CS110; credit to Sarah Van Wart
    #interval = the # of pixels per unit distance in the simulation

    # Delete old grid if it exists:
    canvas.delete('grid_line')
    offset = 2

    # Creates all vertical lines every 0.5 unit
    #for i in range(0, w, interval):
    for i in np.linspace(0, w, 2*w//interval+1).tolist()[:-1]:
        canvas.create_line(i, 0, i, h, tag='grid_line', fill='gray', dash=(2,2))
        make_grid_label(canvas, i, h, w, h, offset, interval)

    # Creates all horizontal lines every 0.5 unit
    #for i in range(0, h, interval):
    for i in np.linspace(0, h, 2*h//interval+1).tolist()[:-1]:
        canvas.create_line(0, i, w, i, tag='grid_line', fill='gray', dash=(2,2))
        make_grid_label(canvas, 0, i, w, h, offset, interval)

def make_coordsys(canvas, x, y, line_length, tag):
    #original work
    canvas.create_line(x, y, x + line_length,               y, arrow=tk.LAST, tag=tag+'x')
    canvas.create_line(x, y,               x, y - line_length, arrow=tk.LAST, tag=tag+'y')

def label_vertices(canvas, box1_vert_gui, box2_vert_gui):
    '''For debug purposes, put labels on each vertex of the boxes so we can see
    which impact conditions are occurring at a given point in time.

    Box1_vert_gui and box2_vert_gui are 10x0 flattened arrays, (x1, y1, x2, y2,...)
    '''
    #uses code from CS110's make_grid() function

    #remove 5th set of box vertices, as it closes the box structure
    box1_vert_gui = np.array(box1_vert_gui)[:-2]
    box2_vert_gui = np.array(box2_vert_gui)[:-2]

    canvas.delete("Vertices")
```

```
    offset = 2

    for i in range(len(box1_vert_gui)//2):
        x, y = box1_vert_gui[2*i : 2*i + 2]
        canvas.create_text(
            x + offset,
            y - offset,
            text=f"V1{i+1}",
            anchor="s",
            font=("Purisa", 8),
            tag="Vertices"
        )

    for i in range(len(box2_vert_gui)//2):
        x, y = box2_vert_gui[2*i : 2*i + 2]
        canvas.create_text(
            x + offset,
            y - offset,
            text=f"V2{i+1}",
            anchor="s",
            font=("Purisa", 8),
            tag="Vertices"
        )

def make_invisible(canvas,id):
    #my own original work from CS110
    canvas.itemconfigure(id, state='hidden')

def make_visible(canvas,id):
    #my own original work from CS110
    canvas.itemconfigure(id, state='normal')

def draw_image(canvas:tk.Canvas, gui, center:tuple, file_path:str, size:int=0,
                  tags:str=None,state='normal'):
    '''
    Makes an image onto the TKinter canvas. Uses a file located at file_path
    relative to the current code.

    Args:
        canvas(TK): the TKinter canvas for displaying images
        gui (TK root/master): necessary for stability when importing 'helpers' into 'main'
        center(Tuple): the location of the center of the image
        file_path(Str): the path of the .PNG you want to paste onto the Tkinter
                canvas. can include folder as well
        size(Int) - optional: the size you want to assign to the image. if size="0", as
                set by default, the image won't be resized from default
        tags(Str) - optional: the identifying word to use in order to group
```

```python
                and move around an image of a certain type
        state(Str) - optional: used to turn a tagged image from visible
                (the 'normal' state) to invisible (state='hidden')

    Returns:
        the ID for the image; position and visibility can be modified
    '''
    #my own work from CS110

    # adds folder directory to path
    directory = os.path.dirname(os.path.realpath(__file__))
    file_path = os.path.join(directory, file_path)
    image = Image.open(file_path)

    # finds default width and height of png

    default_height = image.size[1]
    default_width = image.size[0]

    # changes height of image if needed. an input of '0' will tell the
    # program that the user wants to use the default height.

    if size != 0:
        # "size" input will become the new height
        size_ratio = size/default_height
        height = size
        width = int(round(default_width * size_ratio))

        # antialiasing necessary to keep edges of image smooth when scaling
        image = image.resize((width,height), Image.ANTIALIAS)

    # turns the file name into a tkinter Photo Image using PIL
    photo_image = ImageTk.PhotoImage(image)

    # to keep the image on the screen
    label = tk.Label(gui, image=photo_image)
    label.image = photo_image

    ID = canvas.create_image(center, image = photo_image,
      tags=tags,state=state)
    return ID
```

```python
import numpy as np
import sympy as sym
import dill
import time
from tqdm import tqdm

from geometry import *
from helpers import *

from IPython.display import display

#-----------TESTING FUNCTIONS-----------#

def TestDill(lagrangian):
    #test out pickling a symbolic expression using Dill
    dill.settings['recurse'] = True
    filename ='test_sym_matrix.dill'
    with open(filename, 'wb') as f:
        dill.dump(lagrangian, f)

    lagrangian = 0
    print("Value before dill load:")
    display(lagrangian)

    with open(filename, 'rb') as f:
        lagrangian = dill.load(f)

    print("Value after dill load:")
    display(lagrangian)

def TestHat3():

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

    w1 = [6,5,4]

    w_hat1 = [
        [0, -9, 8],
        [9, 0, -7],
        [-8, 7, 0]
    ]

    w_hat2 = [
        [10, -9, 8],
        [9, 0, -7],
        [-8, 7, 0]
```

```
    ]

    w_hat3 = sym.Matrix([
        [0, -theta1, 8],
        [theta1, 0, -7],
        [-8, 7, 0]
    ])

    T1 = [
        [1,  2,  3,  4],
        [5,  6,  7,  8],
        [9, 10, 11, 12],
        [0,  0,  0,  1]
    ]

    print(f"\nHat: \n {w1} \n{HatVector3(w1)}")
    print(f"\nUnhat: \n{w_hat1} \n{UnhatMatrix3(w_hat1)}")
    #print(f"\nNon-Skew-symm unhat: \n{w_hat2} \n{UnhatMatrix3(w_hat2)}")
    print(f"\nSymbolic unhat: \n{w_hat3} \n{UnhatMatrix3(w_hat3)}")

    print(f"\nTransInv: \n{T1} \n{InvSEn(T1)}")

def TestMatrix4():

    ### testing
    test1, test2, test3 = sym.symbols(r'test_1, test_2, test_3')
    vec1 = np.matrix([1,2,3,4,5,6])
    vec2 = sym.Matrix([test1, test2, test3, test1, test2, test3])
    vec3 = np.array([1, 2, 3, 4, 5, 6])

    #inertia matrix testing
    #---------------------------#

    #print("InertiaMatrix6 tests:")

    ##not currently configured to work
    ## m1 = 4
    ## scriptI1 = 7*np.eye(3)

    m2 = sym.symbols(r'test_m')
    scriptI2 = sym.symbols(r'test_J') * sym.eye(3)
    ## print(InertiaMatrix6(m1, scriptI1))
    #display(InertiaMatrix6(m2, scriptI2))


    #---------------------------#

    mat1 = HatVector6(vec1)
    mat2 = HatVector6(vec2)
```

```
    mat3 = HatVector6(vec3)

    #print("HatVector6 tests:")
    # print(type(mat1))
    # print(mat1, end='\n\n')

    # print(type(mat2))
    # display(mat2)

    # print(type(mat3))
    # print(mat3, end='\n\n')

    #----------------------------#

    vec4 = UnhatMatrix4(mat1)
    vec5 = UnhatMatrix4(mat2)
    vec6 = UnhatMatrix4(mat3)

    # print("UnhatMatrix4 tests:")
    # print(type(vec4))
    # print(vec4, end='\n\n')

    # print(type(vec5))
    # display(vec5)

    # print(type(vec6))
    # print(vec6, end='\n\n')

    pass

def TestVb6():
    #testing
    t = sym.symbols(r't')
    x = sym.Function(r'x')(t)
    y = sym.Function(r'y')(t)
    theta1 = sym.Function(r'\theta_1')(t)
    theta2 = sym.Function(r'\theta_2')(t)

    R = sym.Matrix([
        [sym.cos(-theta2), -sym.sin(-theta2), 0],
        [sym.sin(-theta2),  sym.cos(-theta2), 0],
        [              0,                 0,   1]

    ])

    G = SOnAndRnToSEn(R, [x,y,0])
    V = CalculateVb6(G,t)
    print("\nV:")
    display(V)
```

```python

def TestSEn():

    #test cases

    #SO(2) and R2 - numpy
    mat1 = np.matrix([[1,2],[3,4]])
    p1 = [5,6]
    out = SOnAndRnToSEn(mat1, p1)
    assert np.array_equal(out,  np.matrix([[1,2,5],[3,4,6],[0,0,1]]) ), f"{out}"

    #SO(2) and R2 - sympy
    mat2 = sym.Matrix([[5,6],[7,8]])
    p2 = [9,0]
    out = SOnAndRnToSEn(mat2, p2)
    assert out - sym.Matrix([[5,6,9],[7,8,0],[0,0,1]]) == sym.zeros(3,3), f"{out}"

    #SO(3) and R3 - numpy
    mat3 = np.matrix([[1,2,3],[4,5,6],[7,8,9]])
    p3 = [1.1,2.2,3.3]
    out = SOnAndRnToSEn(mat3, p3)
    assert np.array_equal(out,  np.matrix([[1,2,3,1.1],[4,5,6,2.2],[7,8,9,3.3],[0,0,0,1]]) ), f"{out}"

    #SO(3) and R3 - sympy
    mat4 = sym.Matrix([[1,2,3],[4,5,6],[7,8,9]])
    p4 = [4.4,5.5,6.6]
    out = SOnAndRnToSEn(mat4, p4)
    diff = out - sym.Matrix([[1,2,3,4.4],[4,5,6,5.5],[7,8,9,6.6],[0,0,0,1]])
    assert diff == sym.zeros(4,4), f"{out}\n\n{diff}"

    #dimensional mismatch - check that it throws an error
    #SOnAndRnToSEn(mat2, p4)

    #type mismatch - check that it throws an error
    #SOnAndRnToSEn(mat2, sym.Matrix(p1))
    #SOnAndRnToSEn(mat1, np.matrix(p2))

    #SE(3)
    SE3mat = SOnAndRnToSEn(np.identity(3), [1,2,3])
    [SO3, R3] = SEnToSOnAndRn(SE3mat)
    assert np.array_equal(SO3, np.identity(3)) and np.array_equal(R3, [1,2,3]), f"{SO3}\n{R3}"

    #SE(2)
    SE3mat = SOnAndRnToSEn(np.identity(2), [4,5])
    [SO2, R2] = SEnToSOnAndRn(SE3mat)
```

```
    assert np.array_equal(SO2, np.identity(2)) and np.array_equal(R2, [4,5]),
      f"{SO2}\n{R2}"

    print("All assertions passed")


if __name__ == '__main__':
    #dill_test(lagrangian)
    TestHat3()
    TestMatrix4()
    TestVb6()
    TestSEn()
```