die-2d

December 12, 2024

1 David Khachatryan: 2D Simulation of a Jack in a shaking box

1.0.1 Definition of coordinate frames

[632]:
```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt
from sympy import sin, cos, Matrix, simplify, lambdify, Eq
```

[633]:
```
# Define helper for generating SE(3) matrix
t = sym.symbols('t')

# rotational
def Rotz(theta):
    """Generates a 4x4 SE(3) transformation matrix for rotation about the␣
 ↪z-axis."""
    return sym.Matrix([
        # 4x4 rotation matrix
        [sym.cos(theta), -sym.sin(theta), 0, 0],
        [sym.sin(theta), sym.cos(theta), 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

# translational se(3)
def T(x, y):
    """Generates a 4x4 SE(3) transformation matrix for translation."""
    return sym.Matrix([
        # 4x4 translation matrix
        [1, 0, 0, x],
        [0, 1, 0, y],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

# Define the helper for hatting a 3x1 vector
def hat(v: sym.Matrix) -> sym.Matrix:
    """
```

```python
    Converts a 3x1 vector into a 3x3 skew-symmetric matrix.

    Parameters
    ----------
    v : sympy.Matrix
        A 3x1 vector.

    Returns
    -------
    sympy.Matrix
        A 3x3 skew-symmetric matrix.
    """
    return sym.Matrix([[0, -v[2], v[1]],
                       [v[2], 0, -v[0]],
                       [-v[1], v[0], 0]])

# Define the helper for unhating a 3x3 skew-symmetric matrix
def unhat(v: sym.Matrix) -> sym.Matrix:
    """
    Extracts the 3x1 vector from a 3x3 skew-symmetric matrix.

    The unhat operation is the inverse of the hat operation.

    Parameters
    ----------
    v : sympy.Matrix
        A 3x3 skew-symmetric matrix.

    Returns
    -------
    sympy.Matrix
        A 3x1 vector extracted from the skew-symmetric matrix.
    """
    V = Matrix([0, 0, 0, 0, 0, 0])
    V[0, 0] = v[0, 3]
    V[1, 0] = v[1, 3]
    V[2, 0] = v[2, 3]
    V[3, 0] = v[2, 1]
    V[4, 0] = v[0, 2]
    V[5, 0] = v[1, 0]
    return V


# Define the helper for getting the inverse of a g matrix
def inverse_g(g: sym.Matrix) -> sym.Matrix:
    """
    Computes the inverse of an SE(3) transformation matrix.
```

```
    The inverse of an SE(3) matrix is given as:
        g_inv = [R.T  -R.T * p]
                [  0        1  ],

    where:
        - R.T is the transpose of the rotation matrix.
        - p is the position vector.
    """
    R = g[:3, :3]
    p = g[:3, 3]
    return sym.Matrix([[R.T, -R.T * p], [0, 0, 0, 1]])

def g_dot_SE3(transform, angular_velocity):
    """
    Computes the time derivative of an SE(3) transformation matrix.

    The SE(3) matrix `transform` is composed of a rotation matrix `R` and a␣
 ↪position vector `p`.
    The derivative is given as:
        g_dot = [R_dot   p_dot]
                [  0       0 1],

    where:
        - R_dot = R * hat(w),
          `w` is the angular velocity vector.
        - p_dot = derivative of the position vector `p`.

    Parameters
    ----------
    transform : sympy.Matrix
        The SE(3) homogeneous transformation matrix (3x3 in planar case).
    angular_velocity : sympy.Symbol
        Angular velocity (derivative of the rotation angle, theta).

    Returns
    -------
    sympy.Matrix
        Time derivative of the SE(3) homogeneous transformation matrix (g_dot).

    Notes
    -----
    This assumes a 2D planar system with rotation about the z-axis (right-hand␣
 ↪rule). The system uses:
        - A 3x3 matrix for rotation and translation.
        - Angular velocity vector w = [0, 0, d(theta)/dt], where z is the axis␣
 ↪of rotation.
```

```
    """
    # Compute the angular velocity vector (in matrix form for SE(3))
    w = sym.Matrix([0, 0, angular_velocity.diff(t)])

    # Extract Components
    R = transform[:3, :3]  # Rotation matrix
    p = transform[:3, 3]   # Translation vector

    # Derivative of rotation: R_dot = R * hat(w)
    R_dot = R * hat(w)

    # Derivative of translation: p_dot
    p_dot = p.diff(t)

    # Construct g_dot
    # use T and Rotz to construct the g_dot matrix
    g_dot_matrix = sym.Matrix([[R_dot, p_dot], [0, 0, 0, 1]])
    return g_dot_matrix

def body_velocity(transform, angular_velocity):
    """
    Computes the body velocity from the SE(3) transformation matrix.

    The body velocity is given as a 6D spatial velocity vector (in planar␣
 ↪motion, reduced to 3D):
        V_body = [v  ] -> Linear velocity from R_inv * (p_dot)
                 [w  ] -> Angular velocity

    Parameters
    ----------
    transform : sympy.Matrix
        The SE(3) homogeneous transformation matrix for a frame (3x3 in planar␣
 ↪case).
    angular_velocity : sympy.Symbol
        Angular velocity (derivative of the rotation angle, theta).

    Returns
    -------
    sympy.Matrix
        The spatial body velocity vector for the specified frame.

    Raises
    ------
    ValueError
        If an invalid output type is specified.

    Notes
```

```python
    -----
    - Body velocity is computed as:
          V_body = g^(-1) * g_dot
      where g^(-1) is the inverse of the SE(3) matrix.
    - This function is planar, so angular velocity is scalar (about z-axis).
    """
    # Compute g^-1 (the inverse of SE(3))
    transform_inverse = inverse_g(transform)

    # Compute g_dot
    g_dot_matrix = g_dot_SE3(transform, angular_velocity)

    # Multiply to get body velocity: V_body = g^(-1) * g_dot
    V_body_matrix = transform_inverse * g_dot_matrix

    # Extract body velocity: w = angular component, v = linear component
    w_hat = V_body_matrix[:3, :3]
    v = V_body_matrix[:3, 3]
    w = unhat(w_hat)
    return v.col_join(w)

def integrate(f,xt,dt,time):
    """
    This function takes in an initial condition x(t) and a timestep dt,
    as well as a dynamical system f(x) that outputs a vector of the
    same dimension as x(t). It outputs a vector x(t+dt) at the future
    time step.

    Parameters
    ============
    dyn: Python function
        derivate of the system at a given step x(t),
        it can considered as \dot{x}(t) = func(x(t))
    xt: NumPy array
        current step x(t)
    dt:
        step size for integration

    Return
    ============
    new_xt:
        value of x(t+dt) integrated from x(t)
    """
    k1 = dt * f(xt, 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
```

```
<>:187: SyntaxWarning:

invalid escape sequence '\d'

<>:187: SyntaxWarning:

invalid escape sequence '\d'

/tmp/ipykernel_984525/3401856478.py:187: SyntaxWarning:

invalid escape sequence '\d'
```

[634]:

```
# We have 6 configuration variables
x_b = sym.Function('x_b')(t)
y_b = sym.Function('y_b')(t)
theta_b = sym.Function('theta_b')(t)
x_j = sym.Function('x_j')(t)
y_j = sym.Function('y_j')(t)
theta_j = sym.Function('theta_j')(t)
lamb = sym.symbols(r'lambda')
x_b_dot_Plus, y_b_dot_Plus, theta_b_dot_Plus, x_j_dot_Plus, y_j_dot_Plus,␣
 ↪theta_j_dot_Plus = sym.symbols(r'x_b_dot_+, y_b_dot_+, theta_b_dot_+,␣
 ↪x_j_dot_+, y_j_dot_+, theta_j_dot_+')
xbl, ybl, tbl, xjl, yjl, tjl, xbldot, ybldot, tbldot, xjldot, yjldot, tjldot =␣
 ↪sym.symbols('x_box_l, y_box_l, theta_box_l, x_jack_l, y_jack_l,␣
 ↪theta_jack_l, x_box_ldot, y_box_ldot, theta_box_ldot, x_jack_ldot,␣
 ↪y_jack_ldot, theta_jack_ldot')

q = Matrix([
    x_b,
    y_b,
    theta_b,
    x_j, y_j,
    theta_j]
)
qdot = q.diff(t)
qddot = qdot.diff(t)
```

[635]:

```
# Parameters for the box and jack
box_length, box_mass = 4, 50  # Box length and mass
box_moi = (4) * box_mass * box_length ** 2  # Moment of inertia for the box

jack_length, m_jack = 1, 1  # Jack length and mass
```

```
jack_moi = (4) * m_jack * (jack_length) ** 2  # Moment of inertia for the jack
jack_mass = 1  # Mass of the jack
g = 9.81

# Homogeneous transformation matrices
g_wa = sym.Matrix([
    [cos(theta_b), -sin(theta_b), 0, x_b],
    [sin(theta_b), cos(theta_b), 0, y_b],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_wb = sym.Matrix([
    [cos(theta_j), -sin(theta_j), 0, x_j],
    [sin(theta_j), cos(theta_j), 0, y_j],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_a_a1 = sym.Matrix([
    [1, 0, 0, box_length],
    [0, 1, 0, box_length],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_a_a2 = sym.Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, -box_length],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_a_a3 = sym.Matrix([
    [1, 0, 0, -box_length],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_a_a4 = sym.Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, box_length],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])
```

```python
g_b_b1 = sym.Matrix([
    [1, 0, 0, jack_length],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_b_b2 = sym.Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, -jack_length],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_b_b3 = sym.Matrix([
    [1, 0, 0, -jack_length],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_b_b4 = sym.Matrix([
    [1, 0, 0, 0],
    [0, 1, 0, jack_length],
    [0, 0, 1, 0],
    [0, 0, 0, 1]
])

g_w_a1 = g_wa @ g_a_a1
g_w_a2 = g_wa @ g_a_a2
g_w_a3 = g_wa @ g_a_a3
g_w_a4 = g_wa @ g_a_a4

g_w_j1 = g_wb @ g_b_b1
g_w_j2 = g_wb @ g_b_b2
g_w_j3 = g_wb @ g_b_b3
g_w_j4 = g_wb @ g_b_b4

g_a1_j1 = inverse_g(g_w_a1) @ g_w_j1
g_a1_j2 = inverse_g(g_w_a1) @ g_w_j2
g_a1_j3 = inverse_g(g_w_a1) @ g_w_j3
g_a1_j4 = inverse_g(g_w_a1) @ g_w_j4

g_a2_j1 = inverse_g(g_w_a2) @ g_w_j1
g_a2_j2 = inverse_g(g_w_a2) @ g_w_j2
g_a2_j3 = inverse_g(g_w_a2) @ g_w_j3
g_a2_j4 = inverse_g(g_w_a2) @ g_w_j4
```

```
g_a3_j1 = inverse_g(g_w_a3) @ g_w_j1
g_a3_j2 = inverse_g(g_w_a3) @ g_w_j2
g_a3_j3 = inverse_g(g_w_a3) @ g_w_j3
g_a3_j4 = inverse_g(g_w_a3) @ g_w_j4

g_a4_j1 = inverse_g(g_w_a4) @ g_w_j1
g_a4_j2 = inverse_g(g_w_a4) @ g_w_j2
g_a4_j3 = inverse_g(g_w_a4) @ g_w_j3
g_a4_j4 = inverse_g(g_w_a4) @ g_w_j4
```

[636]:
```
origin = sym.Matrix([0, 0, 0, 1])
r_wa = g_wa @ origin
r_wb = g_wb @ origin
```

[637]:
```
# Now calculate velocities of the box and jack
#v_a = sym.simplify(body_velocity(g_wa, theta_b))
#v_b = sym.simplify(body_velocity(g_wb, theta_j))
#
v_a = unhat(inverse_g(g_wa) @ g_wa.diff(t))
v_b = unhat(inverse_g(g_wb) @ g_wb.diff(t))
```

[638]:
```
# Now calculate inertia
I_a = sym.Matrix([
    [4*box_mass, 0, 0, 0, 0, 0],
    [0, 4*box_mass, 0, 0, 0, 0],
    [0, 0, 4*box_mass, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, box_moi]
])

I_b = sym.Matrix([
    [4*jack_mass, 0, 0, 0, 0, 0],
    [0, 4*jack_mass, 0, 0, 0, 0],
    [0, 0, 4*jack_mass, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, jack_moi]
])
```

1.1 Euler-Lagrange equations

[639]:
```
# Now calculate the kinetic energy
KE = sym.simplify((1/2)*v_a.T*I_a*v_a + (1/2)*v_b.T*I_b*v_b)[0]
V = 4 * box_mass * g * y_b + 4 * jack_mass * g * y_j
L = KE - V
```

```python
## Now calculate the Euler-Lagrange equations
#L_qdot = L.diff(qdot)
#L_qdot_dot = L_qdot.diff(t)
#L_q = L.diff(q)
#eqs = sym.simplify(L_q - L_qdot_dot)
#
#lhs = eqs
#rhs = F_ext
#EL_eqs = Eq(rhs, lhs)
#display(EL_eqs)
#
## Now solve the equations
#sol = sym.solve(EL_eqs, qddot, dict=True)
#display(sol)
# External forces (box shaking parameters)
k = 10000
theta_d_b = sin(np.pi * t / 2.5)
F_theta_b = k * theta_d_b
F_y_b = 4 * box_mass * g
F_ext = Matrix([0, F_y_b, F_theta_b, 0, 0, 0])

# Solve lagrange equations
dL_dq = simplify(Matrix([L]).jacobian(q).T)
dL_dqdot = simplify(Matrix([L]).jacobian(qdot).T)
ddL_dqdot_dt = simplify(dL_dqdot.diff(t))

lhs = simplify(ddL_dqdot_dt - dL_dq)
rhs = simplify(F_ext)
EL_Eqs = simplify(Eq(lhs, rhs))
# Solve the Euler-Lagrange Equations:
sol = sym.solve(EL_Eqs, qddot, dict=True)

# Compute the Hamiltonian:
H = simplify((dL_dqdot.T * qdot)[0] - L)
```

[640]:
```python
# Create a dictionary to store the lambdified functions
ddot_funcs = {
    'x_box': lambdify([*q, *qdot, t], sol[0][qddot[0]]),
    'y_box': lambdify([*q, *qdot, t], sol[0][qddot[1]]),
    'theta_box': lambdify([*q, *qdot, t], sol[0][qddot[2]]),
    'x_jack': lambdify([*q, *qdot, t], sol[0][qddot[3]]),
    'y_jack': lambdify([*q, *qdot, t], sol[0][qddot[4]]),
    'theta_jack': lambdify([*q, *qdot, t], sol[0][qddot[5]])
}

def dynamics(s, t):
    # Compute accelerations using the lambdified functions
```

```
    q_ddots = [func(*s, t) for func in ddot_funcs.values()]

    # Combine velocity and acceleration into the state derivative
    sdot = np.array([
        *s[6:],  # Velocities
        *q_ddots # Accelerations
    ])

    return sdot
```

[641]:
```
#$ Tau plus handling boilerplate:
elements = []
for i in range(6):
    elements.extend([qdot[i], sol[0][qddot[i]]])

qddot_Matrix = Matrix(elements)
state_variable_mapping = {
    q[i]: vars()[f'{var}_l'] for i, var in enumerate(['x_b', 'y_b', 'theta_b',␣
 ↪'x_j', 'y_j', 'theta_j'])
}
state_variable_mapping.update({
    qdot[i]: vars()[f'{var}_ldot'] for i, var in enumerate(['x_b', 'y_b',␣
 ↪'theta_b', 'x_j', 'y_j', 'theta_j'])
})

# Substitute the variables in the qddot matrix
qddot_d = qddot_Matrix.subs(state_variable_mapping)

# Define the lambdified function
qddot_lambdify = lambdify(
    [
        xbl, xbldot, ybl, ybldot, tbl, tbldot,
        xjl, xjldot, yjl, yjldot, tjl, tjldot, t
    ],
    qddot_d
)
```

1.2 Defining impact constraints

[ ]:
```
# Wall impact handling boilerplate:
wall_b1_j1 = (g_a1_j1[3]).subs(state_variable_mapping)
wall_b1_j2 = (g_a1_j2[3]).subs(state_variable_mapping)
wall_b1_j3 = (g_a1_j3[3]).subs(state_variable_mapping)
wall_b1_j4 = (g_a1_j4[3]).subs(state_variable_mapping)
wall_b2_j1 = (g_a2_j1[7]).subs(state_variable_mapping)
wall_b2_j2 = (g_a2_j2[7]).subs(state_variable_mapping)
wall_b2_j3 = (g_a2_j3[7]).subs(state_variable_mapping)
```

```
wall_b2_j4 = (g_a2_j4[7]).subs(state_variable_mapping)
wall_b3_j1 = (g_a3_j1[3]).subs(state_variable_mapping)
wall_b3_j2 = (g_a3_j2[3]).subs(state_variable_mapping)
wall_b3_j3 = (g_a3_j3[3]).subs(state_variable_mapping)
wall_b3_j4 = (g_a3_j4[3]).subs(state_variable_mapping)
wall_b4_j1 = (g_a4_j1[7]).subs(state_variable_mapping)
wall_b4_j2 = (g_a4_j2[7]).subs(state_variable_mapping)
wall_b4_j3 = (g_a4_j3[7]).subs(state_variable_mapping)
wall_b4_j4 = (g_a4_j4[7]).subs(state_variable_mapping)

# Constraint
constraint = simplify(
    Matrix([
        [wall_b1_j1], [wall_b1_j2], [wall_b1_j3], [wall_b1_j4],
        [wall_b2_j1], [wall_b2_j2], [wall_b2_j3], [wall_b2_j4],
        [wall_b3_j1], [wall_b3_j2], [wall_b3_j3], [wall_b3_j4],
        [wall_b4_j1], [wall_b4_j2], [wall_b4_j3], [wall_b4_j4]
    ])
)

Hamiltonian_ = H.subs(state_variable_mapping)
dL_dqdot_dum = dL_dqdot.subs(state_variable_mapping)
dPhidq_dum = constraint.jacobian([xbl, ybl, tbl, xjl, yjl, tjl])
impact_dict = {
    xbldot:x_b_dot_Plus,
    ybldot:y_b_dot_Plus,
    tbldot:theta_b_dot_Plus,
    xjldot:x_j_dot_Plus,
    yjldot:y_j_dot_Plus,
    tjldot:theta_j_dot_Plus
}

# tau+ evaluations:
dL_dqdot_dumPlus = simplify(dL_dqdot_dum.subs(impact_dict))
dPhidq_dumPlus = simplify(dPhidq_dum.subs(impact_dict))
Hamiltonian_Plus = simplify(Hamiltonian_.subs(impact_dict))
impact_eqns_list = []

# Define equations
lhs = Matrix([dL_dqdot_dumPlus[0] - dL_dqdot_dum[0],
              dL_dqdot_dumPlus[1] - dL_dqdot_dum[1],
              dL_dqdot_dumPlus[2] - dL_dqdot_dum[2],
              dL_dqdot_dumPlus[3] - dL_dqdot_dum[3],
              dL_dqdot_dumPlus[4] - dL_dqdot_dum[4],
              dL_dqdot_dumPlus[5] - dL_dqdot_dum[5],
              Hamiltonian_Plus - Hamiltonian_])
```

```
for i in range(constraint.shape[0]):
    rhs = Matrix([lamb*dPhidq_dum[i,0],
                  lamb*dPhidq_dum[i,1],
                  lamb*dPhidq_dum[i,2],
                  lamb*dPhidq_dum[i,3],
                  lamb*dPhidq_dum[i,4],
                  lamb*dPhidq_dum[i,5],
                  0])
    impact_eqns_list.append(simplify(Eq(lhs, rhs)))
```

1.3 Impact Update

[ ]:
```
dum_list = [
    x_b_dot_Plus, y_b_dot_Plus, theta_b_dot_Plus,
    x_j_dot_Plus, y_j_dot_Plus, theta_j_dot_Plus
]

def impact_update(s, impact_eqns, dum_list):
    """
    This function takes in the current state of the system and the impact␣
 ↪equations
    and returns the updated state of the system after the impact.

    Parameters
    ============
    s: NumPy array
        current state of the system

    impact_eqns: list
        list of impact equations

    dum_list: list
        list of dummy variables

    Return
    ============
    s: NumPy array
        updated state of the system
    """
    subs = {
        xbl:s[0], ybl:s[1], tbl:s[2],
        xjl:s[3], yjl:s[4], tjl:s[5],
        xbldot:s[6], ybldot:s[7], tbldot:s[8],
        xjldot:s[9], yjldot:s[10], tjldot:s[11]
    }
    new_impact_eqns = impact_eqns.subs(subs)
    impact_solns = sym.solve(
```

```
        new_impact_eqns,
        [
            x_b_dot_Plus, y_b_dot_Plus, theta_b_dot_Plus,
            x_j_dot_Plus, y_j_dot_Plus, theta_j_dot_Plus,
            lamb
        ],
        dict=True
    )

    if len(impact_solns) != 1:
        for sol in impact_solns:
            lamb_sol = sol[lamb]
            if abs(lamb_sol) > 1e-06:
                return np.array([
                    *s[:6],
                    float(sym.N(sol[dum_list[0]])),
                    float(sym.N(sol[dum_list[1]])),
                    float(sym.N(sol[dum_list[2]])),
                    float(sym.N(sol[dum_list[3]])),
                    float(sym.N(sol[dum_list[4]])),
                    float(sym.N(sol[dum_list[5]])),
                ])
```

1.4 Impact Condition

[ ]:
```
phi_func = lambdify(
    [
        xbl, ybl, tbl,
        xjl, yjl, tjl,
        xbldot, ybldot, tbldot,
        xjldot, yjldot, tjldot
    ],
    constraint
)

def impact_condition(s, phi_func, threshold = 1e-1):
    """
    This function checks if the system is in impact condition.
    It returns True if the system is in impact condition, False otherwise.

    Parameters
    ----------
    s : np.array
        The state of the system.

    phi_func : function
        The function that calculates the impact constraints.
```

```
    threshold : float
        The threshold for the impact condition.

    Returns
    -------
    int: The index of the impact constraint that is less than the threshold.
         -1 if no impact condition is met.
    """

    # Get the impact constraints
    phi_val = phi_func(*s)

    # Check if any of the constraints are less than the threshold
    for i in range(phi_val.shape[0]):
        if abs(phi_val[i]) < threshold:
            return i

    return -1
```

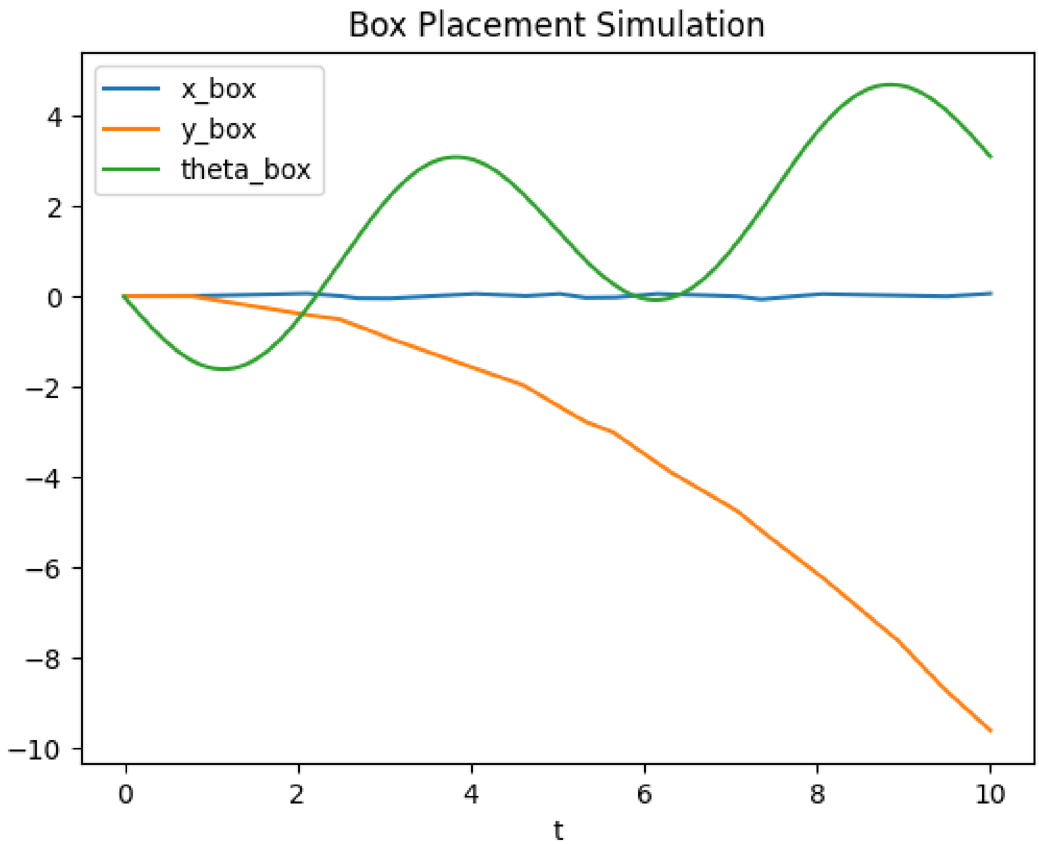
[645]:
```
def simulate_impact(f, x0, tspan, dt, integrate):
    """
    This function simulates the trajectory of a dynamical system
    from a given initial condition x0, over a time span tspan,
    with a time step dt. It uses the numerical integration method
    specified in the input argument 'integrate'.

    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
    """
```

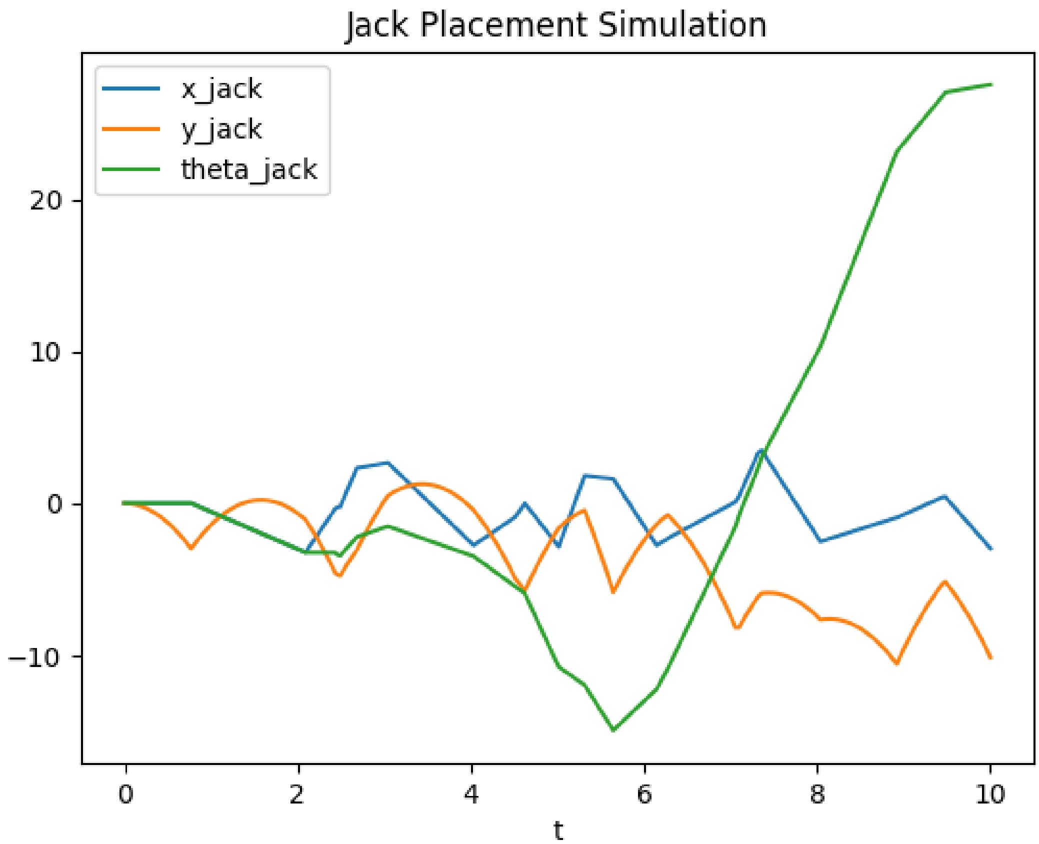
```
    # Initialize the count of the number of time steps
    num = int((max(tspan) - min(tspan)) / dt)

    # Copy the initial condition
    x = np.copy(x0)

    # Initialize the trajectory array
    xtraj = np.zeros((len(x0), num))
    time = 0
    for i in range(num):
        # Update the time
        time += dt

        # Check for impact condition
        impact = impact_condition(x, phi_func, 1e-1)
        if impact != -1:
            # Update the system after impact
            x = impact_update(x, impact_eqns_list[impact], dum_list)

        # Integrate the system
        xtraj[:, i]=integrate(f, x, dt, time)

        # Update the state for the next iteration
        x = np.copy(xtraj[:,i])
    return xtraj
```

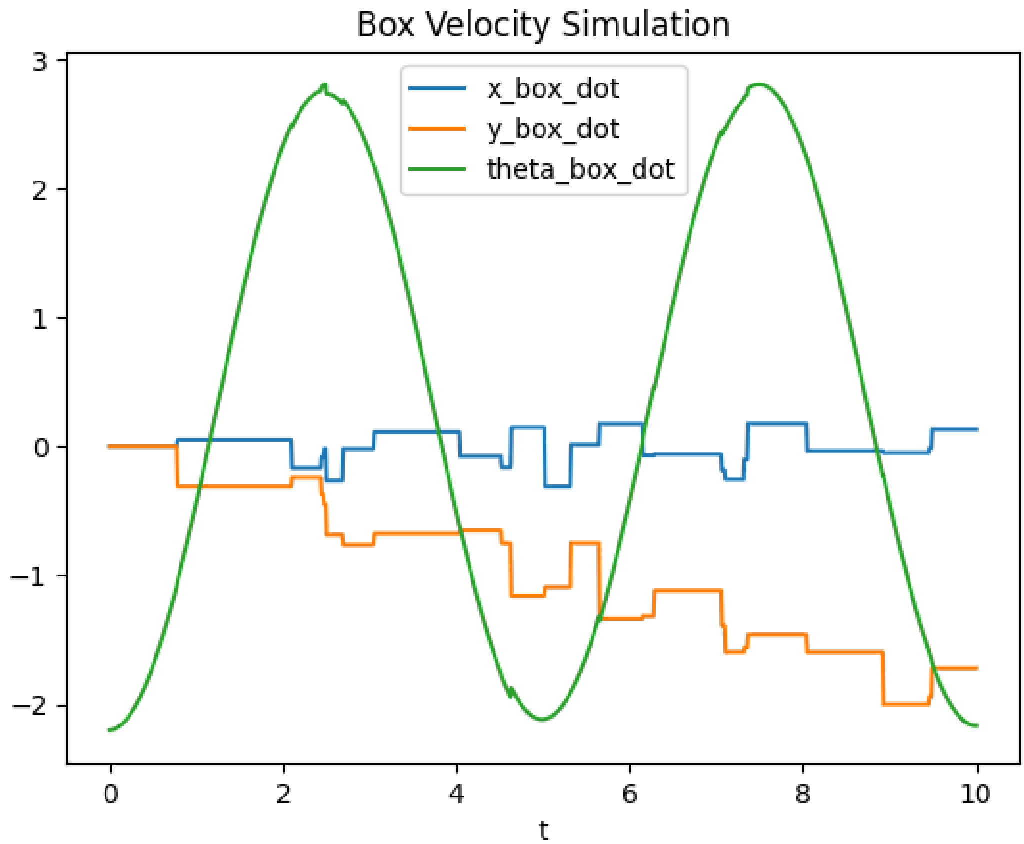
```
<>:2: SyntaxWarning:

invalid escape sequence '\d'

<>:2: SyntaxWarning:

invalid escape sequence '\d'

/tmp/ipykernel_984525/520826766.py:2: SyntaxWarning:

invalid escape sequence '\d'
```

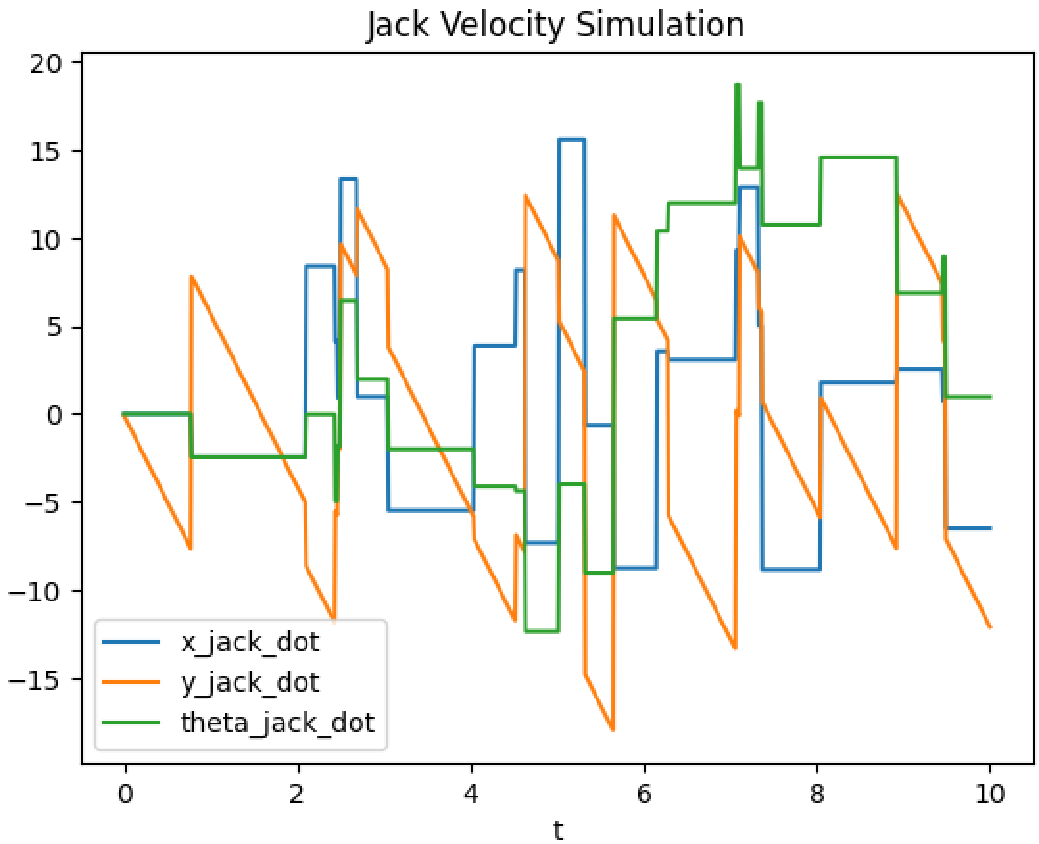
[646]:
```
# Simulate the motion:
tspan = [0, 10]
dt = 0.01
s0 = np.array([0, 0, 0, 0, 0, 0, 0, 0, -2.2, 0, 0, 0])

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

```python
plt.figure()
plt.plot(tvec, traj[0], label='x_box')
plt.plot(tvec, traj[1], label='y_box')
plt.plot(tvec, traj[2], label='theta_box')
plt.title('Box Placement Simulation')
plt.xlabel('t')

# Add the plot labels as legends on the plot
plt.legend()

plt.show()

plt.figure()
plt.plot(tvec, traj[3], label='x_jack')
plt.plot(tvec, traj[4], label='y_jack')
plt.plot(tvec, traj[5], label='theta_jack')
plt.title('Jack Placement Simulation')
plt.xlabel('t')
plt.legend()
plt.show()

plt.figure()
plt.plot(tvec, traj[6], label='x_box_dot')
plt.plot(tvec, traj[7], label='y_box_dot')
plt.plot(tvec, traj[8], label='theta_box_dot')
plt.title('Box Velocity Simulation')
plt.xlabel('t')
plt.legend()
plt.show()

plt.figure()
plt.plot(tvec, traj[9], label='x_jack_dot')
plt.plot(tvec, traj[10], label='y_jack_dot')
plt.plot(tvec, traj[11], label='theta_jack_dot')
plt.title('Jack Velocity Simulation')
plt.legend()
plt.xlabel('t')

plt.show()
```


Box Placement Simulation
x_box
y_box
theta_box
4
2
0
−2
−4
−6
−8
−10
0
2
4
6
8
10
t


Jack Placement Simulation
x_jack
y_jack
theta_jack
20
10
0
−10
0
2
4
6
8
10
t


Box Velocity Simulation
x_box_dot
y_box_dot
theta_box_dot
3
2
1
0
−1
−2
0
2
4
6
8
10
t



[647]:
```python
def animate(config_array,l=1,w=0.2,T=10):
    """
    Function to generate web-based animation of the system

    Parameters:
    ================================================
    config_array:
        trajectory of theta1 and theta2, should be a NumPy array with
        shape of (2,N)
    L1:
        length of the first pendulum
    L2:
        length of the second pendulum
    T:
        length/seconds of animation duration

    Returns: None
    """
    ################################
```

```python
# Imports required for animation. (leave this part)
from plotly.offline import init_notebook_mode, iplot
from IPython.display import display, HTML
import plotly.graph_objects as go

#######################
# Browser configuration. (leave this part)
def configure_plotly_browser_state():
    import IPython
    display(IPython.core.display.HTML('''
        <script src="/static/components/requirejs/require.js"></script>
        <script>
          requirejs.config({
            paths: {
              base: '/static/base',
              plotly: 'https://cdn.plot.ly/plotly-1.5.1.min.js?noext',
            },
          });
        </script>
        '''))
configure_plotly_browser_state()
init_notebook_mode(connected=False)


###############################################
# Getting data from pendulum angle trajectories.
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]

b1_x_array = np.zeros(N, dtype=np.float32)
b1_y_array = np.zeros(N, dtype=np.float32)
b2_x_array = np.zeros(N, dtype=np.float32)
b2_y_array = np.zeros(N, dtype=np.float32)
b3_x_array = np.zeros(N, dtype=np.float32)
b3_y_array = np.zeros(N, dtype=np.float32)
b4_x_array = np.zeros(N, dtype=np.float32)
b4_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)
j1_x_array = np.zeros(N, dtype=np.float32)
```

```
    j1_y_array = np.zeros(N, dtype=np.float32)
    j2_x_array = np.zeros(N, dtype=np.float32)
    j2_y_array = np.zeros(N, dtype=np.float32)
    j3_x_array = np.zeros(N, dtype=np.float32)
    j3_y_array = np.zeros(N, dtype=np.float32)
    j4_x_array = np.zeros(N, dtype=np.float32)
    j4_y_array = np.zeros(N, dtype=np.float32)

    for t in range(N):
        g_w_b = np.array([[np.cos(theta_box_array[t]), -np.
↪sin(theta_box_array[t]), 0, x_box_array[t]],
                            [np.sin(theta_box_array[t]), np.
↪cos(theta_box_array[t]), 0, y_box_array[t]],
                            [0, 0, 1, 0],
                            [0, 0, 0, 1]])

        g_w_j = np.array([[np.cos(theta_jack_array[t]), -np.
↪sin(theta_jack_array[t]), 0, x_jack_array[t]],
                            [np.sin(theta_jack_array[t]), np.
↪cos(theta_jack_array[t]), 0, y_jack_array[t]],
                            [0, 0, 1, 0],
                            [0, 0, 0, 1]])


        b1 = g_w_b.dot(np.array([box_length, box_length, 0, 1]))
        b1_x_array[t] = b1[0]
        b1_y_array[t] = b1[1]
        b2 = g_w_b.dot(np.array([box_length, -box_length, 0, 1]))
        b2_x_array[t] = b2[0]
        b2_y_array[t] = b2[1]
        b3 = g_w_b.dot(np.array([-box_length, -box_length, 0, 1]))
        b3_x_array[t] = b3[0]
        b3_y_array[t] = b3[1]
        b4 = g_w_b.dot(np.array([-box_length, box_length, 0, 1]))
        b4_x_array[t] = b4[0]
        b4_y_array[t] = b4[1]

        j = g_w_j.dot(np.array([0, 0, 0, 1]))
        j_x_array[t] = j[0]
        j_y_array[t] = j[1]
        j1 = g_w_j.dot(np.array([jack_length, 0, 0, 1]))
        j1_x_array[t] = j1[0]
        j1_y_array[t] = j1[1]
        j2 = g_w_j.dot(np.array([0, -jack_length, 0, 1]))
        j2_x_array[t] = j2[0]
        j2_y_array[t] = j2[1]
        j3 = g_w_j.dot(np.array([-jack_length, 0, 0, 1]))
```

```
        j3_x_array[t] = j3[0]
        j3_y_array[t] = j3[1]
        j4 = g_w_j.dot(np.array([0, jack_length, 0, 1]))
        j4_x_array[t] = j4[0]
        j4_y_array[t] = j4[1]

    ####################################
    # Axis limits.
    xm = -11
    xM = 11
    ym = -15
    yM = 11

    ###########################
    # Defining data dictionary.
    data=[dict(name = 'Box'),
          dict(name = 'Jack'),
          dict(name = 'Mass1_Jack'),
    ]

    ################################
    # Preparing simulation layout.
    layout=dict(autosize=False, width=1000, height=1000,
                xaxis=dict(range=[xm, xM], autorange=False,␣
↪zeroline=True,dtick=1),
                yaxis=dict(range=[ym, yM], autorange=False,␣
↪zeroline=False,scaleanchor = "x",dtick=1),
                title='2D: Jack in a Box Simulation',
                hovermode='closest',
                updatemenus= [{'type': 'buttons',
                               'buttons': [{'label': 'Play','method': 'animate',
                                            'args': [None, {'frame':␣
↪{'duration': T, 'redraw': False}}]},
                                           {'args': [[None], {'frame':␣
↪{'duration': T, 'redraw': False}, 'mode': 'immediate',
                                            'transition': {'duration':␣
↪0}}],'label': 'Pause','method': 'animate'}
                                          ]
                              }]
               )

    ########################################
    # Defining the frames of the simulation.
    frames=[dict(data=[
           ␣
↪dict(x=[b1_x_array[k],b2_x_array[k],b3_x_array[k],b4_x_array[k],b1_x_array[k]],
```

```
↪y=[b1_y_array[k],b2_y_array[k],b3_y_array[k],b4_y_array[k],b1_y_array[k]],
                 mode='lines',
                 line=dict(color='purple', width=3)
                 ),
            ␣
↪dict(x=[j1_x_array[k],j3_x_array[k],j_x_array[k],j2_x_array[k],j4_x_array[k]],
                 ␣
↪y=[j1_y_array[k],j3_y_array[k],j_y_array[k],j2_y_array[k],j4_y_array[k]],
                 mode='lines',
                 line=dict(color='black', width=3)
                 ),
            go.Scatter(
                x=[j1_x_array[k],j2_x_array[k],j3_x_array[k],j4_x_array[k]],
                y=[j1_y_array[k],j2_y_array[k],j3_y_array[k],j4_y_array[k]],
                mode="markers",
                marker=dict(color='red', size=6)),
                 ]) for k in range(N)]

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

##############
# The animation:
animate(traj)
```

<IPython.core.display.HTML object>

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```