HW6

November 14, 2023

# 1 Homework 6

Qingyuan Chen

[1]:
```
#IMPORT ALL NECESSARY PACKAGES AT THE TOP OF THE CODE
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

# Custom display helpers
from IPython.display import Markdown

def md_print(md_str: str):
    display(Markdown(md_str))

def lax_eq(equation):
    return sym.latex(equation , mode='inline')



import sympy as sym
def get_eu_la(L: sym.Function, q: sym.Matrix, t: sym.symbols):
    """Generate euler lagrangian using sympy jacobian

    Args:
        L (sym.Function): Lagrangian equation
        q (sym.Matrix): matrix of system-var q
        t (sym.symbols): time symbol (needed for q.diff(t))
    """

    q_dot = q.diff(t)
    dL_dq = sym.simplify(sym.Matrix([L]).jacobian(q).T)
    dL_dq_dot = sym.simplify(sym.Matrix([L]).jacobian(q_dot).T)

    return sym.simplify(dL_dq_dot.diff(t) - dL_dq)


def solve_and_print(variables: sym.Matrix,
```

```python
                    eu_la_eq: sym.Eq , quiet = False) -> list[dict[any]]:
    """Solve the given eu_la equation

    Args:
        variables (sym.Matrix): var to solve for
        eu_la_eq (sym.Eq): eu_la equation
        quiet (bool): turn off any printing if True

    Returns:
        list[dict[sym.Function]]: list of solution dicts (keyed with variables)
    """
    solution_dicts = sym.solve(eu_la_eq, variables, dict=True)
    i = 0
    print(f"Total of {len(solution_dicts)} solutions")
    for solution_dict in solution_dicts:
        i += 1
        if not quiet: md_print(f"solution : {i} / {len(solution_dicts)}")
        for var in variables:
            sol = solution_dict[var]
            if not quiet: md_print(f"{lax_eq(var)} = {lax_eq(sol.expand())}")
    return solution_dicts

def lambdify_sys(var_list: list, function_dict: dict[any, sym.Function], keys␣
 ↪=None):
    lambda_dict ={}
    if keys is None:
        keys = function_dict.keys()
    for var in keys:
        acceleration_function = (function_dict[var])
        lambda_func = sym.lambdify(var_list, acceleration_function )
        lambda_dict[var] = lambda_func
    return lambda_dict

def make_system_equation(lambda_dict,lam_keys):
    '''
    '''
    def system_equation(state , lambda_dict ,lam_keys):
        state = state.tolist()
        accel_list = []

        for key in lam_keys:
            accel_list.append(lambda_dict[key](*state) )

        velocity_list = state[int (len(state)/2): :]
        out_list = velocity_list + accel_list
        return np.array(out_list)
    return lambda state: system_equation(state , lambda_dict , lam_keys)
```

```python
def make_system_equation_6(lambda_dict,lam_keys):
    def system_equation(state , lambda_dict ,lam_keys):
        '''
        argumetn:
        state -> array of 4 item, x,theta,xdot ,thetadot
        return -> array of 4 item, xdot, thetadot, xddot, thetaddot
        '''
        q_1 = state[0]
        q_2 =state[1]
        q_3 =state[2]
        v_1 = state[3]
        v_2 = state[4]
        v_3 = state[5]
        a_1 =  lambda_dict[lam_keys[0]](q_1,q_2,q_3,v_1,v_2 ,v_3)
        a_2 = lambda_dict[lam_keys[1]](q_1,q_2,q_3,v_1,v_2,v_3)
        a_3 = lambda_dict[lam_keys[2]](q_1,q_2,q_3,v_1,v_2,v_3)

        return np.array([v_1, v_2, v_3, a_1, a_2, a_3])

    return lambda state: system_equation(state , lambda_dict , lam_keys)
```

[2]:
```python
# Integrate and simulate

def integrate(f, xt, dt):
    """
    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)
```

```
    k2 = dt * f(xt+k1/2.)
    k3 = dt * f(xt+k2/2.)
    k4 = dt * f(xt+k3)

    new_xt = xt + (1/6.) * (k1+2.0*k2+2.0*k3+k4)
    return new_xt

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))
    for i in range(N):
        xtraj[:,i]=integrate(f,x,dt)
        x = np.copy(xtraj[:,i])
    return tvec , xtraj
```

## 1.1 Problem 1

**Proof:** Property of a matrix $R \in SO(n)$ is $R  R^T = I $ and $det(R) = 1$

For Rotation matrix $A, B$ which $A \in SO(n)$ and $B \in SO(n)$

$A \cdot A^T = I, B \cdot B^T = I, det(A) = 1, det(B) = 1$

$[A \cdot B] \cdot [A \cdot B]^T$

$ =A  B  B^T  A^T $

$=A  I  A^T $

$= I$

$det(A \cdot B)$

$= det(A) \cdot det(b)$

$ = 1$

## 1.2 Problem 2

spliting $$g = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$$

into. $$\begin{cases} g_R = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \\ g_p = \begin{bmatrix} I_{2\times1} & P \\ 0 & 1 \end{bmatrix} \end{cases}$$

when applying the transformation on vector v

origional: $$g \cdot V = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \\ 1 \end{bmatrix} = \begin{bmatrix} R \cdot V + P \\ 1 \end{bmatrix}$$

translation then rotation

split: $$g_p \cdot g_R \cdot V = \begin{bmatrix} I_{2\times1} & P \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \\ 1 \end{bmatrix}_{3\times1}$$

$$= \begin{bmatrix} I_{2\times1} & P \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \cdot R \\ 1 \end{bmatrix}_{3\times1} = \begin{bmatrix} V \cdot R + P \\ 1 \end{bmatrix}$$

the same as $g \cdot v$,

### 1.3 Problem 3

spliting $g = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$

into. $\begin{cases} g_R = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \\ g_p = \begin{bmatrix} I_{2\times 1} & P \\ 0 & 1 \end{bmatrix} \end{cases}$

when applying the transformation on vector $v$

origional : $g \cdot V = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \\ 1 \end{bmatrix} = \begin{bmatrix} R \cdot V + P \\ 1 \end{bmatrix}$

translation then rotation

split: $g_p \cdot g_R \cdot V = \begin{bmatrix} I_{2\times 1} & P \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \\ 1 \end{bmatrix}_{3\times 1}$

$= \begin{bmatrix} I_{2\times 1} & P \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \cdot R \\ 1 \end{bmatrix}_{3\times 1} = \begin{bmatrix} V \cdot R + P \\ 1 \end{bmatrix}$

the same as $g \cdot v$,

rotation then translation

$$g_R \cdot g_p \cdot V = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} I_{2x1} & P \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V \\ 1 \end{bmatrix}_{3x1}$$

$$= \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V + P \\ 1 \end{bmatrix}_{3x1} = \begin{bmatrix} R \cdot V + R \cdot P \\ 1 \end{bmatrix}_{3x1}$$

whis is not the same as $g \cdot V$.

Also: $g_R \cdot g_p = \begin{bmatrix} R & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} I_{2x1} & P \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R & R \cdot P \\ 0 & 1 \end{bmatrix}$

Is not the same as $g = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$

**The homogeneous transformation is the same as translation then rotation**

**If rotation is done first, then the translation is happening in a frame that has been rotated, thus the effect of translation will become rotation*translation**

### 1.4 Problem 4

Frame definition is same in the given file

[3]:
```
from IPython.core.display import HTML
display(HTML("<table><tr><td><img src='https://github.com/MuchenSun/ME314pngs/
↪raw/master/doubpend_frames.jpg' width=500' height='350'></table>"))
```

```
<IPython.core.display.HTML object>
```

[4]:
```
def rot(theta):
    return sym.Matrix([[sym.cos(theta) , -sym.sin(theta) ,0],
                        [sym.sin(theta) , sym.cos(theta) ,0],
                        [ 0             , 0              ,1]])

def trans(x,y):
    return sym.Matrix([
        [1,0,x],
        [0,1,y],
        [0,0,1],
    ])
```

```python
def get_x_y(T):
    return T[0,2] , T[1,2]

m1=m2=1
R1=R2=1
g=-9.8

t= sym.symbols('t')

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


theta1_dot = theta1.diff(t)
theta2_dot = theta2.diff(t)
theta1_ddot = theta1_dot.diff(t)
theta2_ddot = theta2_dot.diff(t)

q = sym.Matrix([theta1,theta2])
q_ddot = sym.Matrix([theta1_ddot , theta2_ddot])
T_WA = rot(theta1)
T_AB = trans(0,-R1)
T_BC = rot(theta2)
T_CD = trans(0,-R2)

T_WB = T_WA @ T_AB
T_WD = T_WB @ T_BC @ T_CD

x1,y1 = get_x_y(T_WB)
x2,y2 = get_x_y(T_WD)

# x1 = x1.simplify()
# x2 = x2.simplify()
# y1 = y1.simplify()
# y2 = y2.simplify()
m1_kinematic = 0.5* m1 * (x1.diff(t) **2 + y2.diff(t) **2)
m2_kinematic = 0.5* m2 * (x2.diff(t) **2 + y2.diff(t) **2)
m1_potential = -m1*g*y1
m2_potential = -m2*g*y2
total_kinematic = (m1_kinematic + m2_kinematic).trigsimp().simplify()
total_potential = (m1_potential + m2_potential).trigsimp().simplify()

# display(total_kinematic)
eu_la = get_eu_la((total_kinematic - total_potential).simplify() , q,t)
eu_la = eu_la.trigsimp()
# display(eu_la)
```

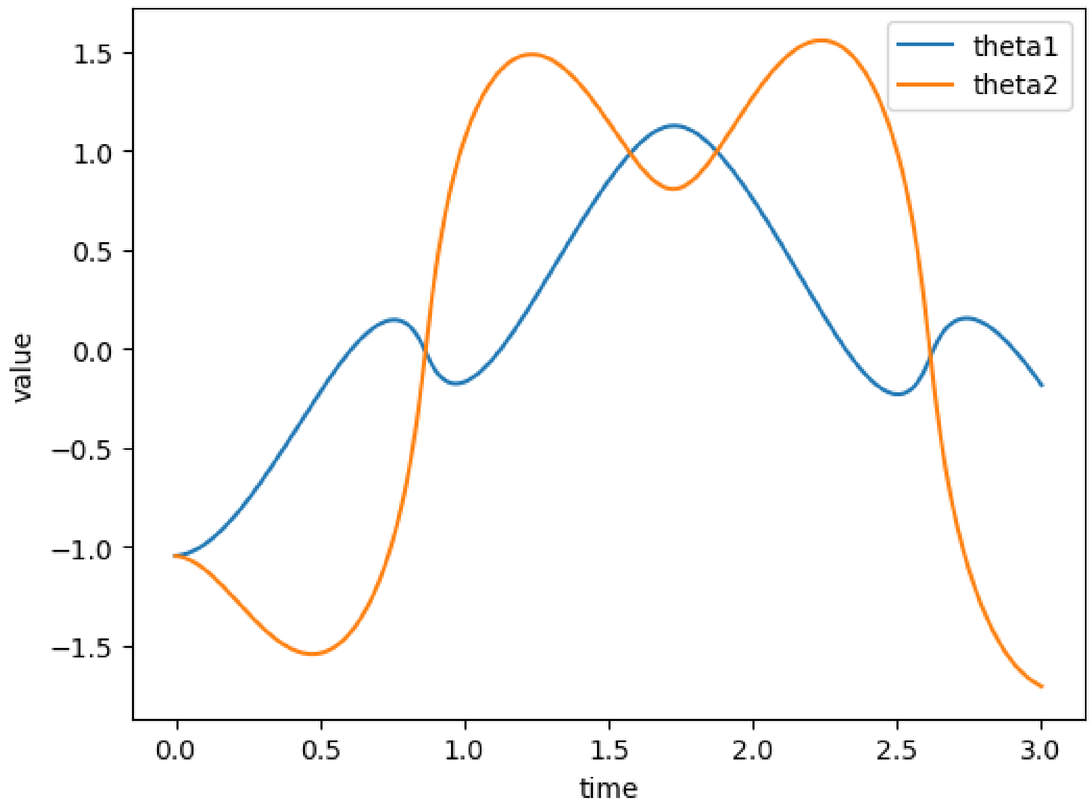
[5]:
```
eu_la_sol = solve_and_print(q_ddot , eu_la , quiet=True)[0]
```

```
Total of 1 solutions
```

[6]:
```
lambda_dict = lambdify_sys([theta1,theta2,theta1_dot,theta2_dot] , eu_la_sol␣
↪,q_ddot)
system_eq = make_system_equation(lambda_dict,q_ddot)


t_range = [0,3]
dt = 0.01
q0 = np.array([ -np.pi/3 , -np.pi/3 , 0,0  ])


tvec , traj = simulate( system_eq , q0,t_range,dt,integrate)


plt.figure(1)
plt.plot(tvec , traj[0] , label = "theta1")
plt.plot(tvec , traj[1] , label = "theta2")
plt.xlabel("time")
plt.ylabel("value")
plt.legend()
plt.plot()
```

```
/usr/lib/python3/dist-packages/scipy/__init__.py:146: UserWarning: A NumPy
version >=1.17.3 and <1.25.0 is required for this version of SciPy (detected
version 1.26.1
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

[6]:
```
[]
```




[7]:
```
def animate_double_pend(theta_array,L1=1,L2=1,T=10):
    """
    Function to generate web-based animation of double-pendulum system

    Parameters:
    ================================================
    theta_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
    """

    ################################
    # Imports required for animation.
```

```python
from plotly.offline import init_notebook_mode, iplot
from IPython.display import display, HTML
import plotly.graph_objects as go

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

###############################################
# Getting data from pendulum angle trajectories.
xx1=L1*np.sin(theta_array[0])
yy1=-L1*np.cos(theta_array[0])
xx2=xx1+L2*np.sin(theta_array[0]+theta_array[1])
yy2=yy1-L2*np.cos(theta_array[0]+theta_array[1])
N = len(theta_array[0]) # Need this for specifying length of simulation

###############################################
# Define arrays containing data for frame axes
# In each frame, the x and y axis are always fixed
def rot(theta):
    return np.array([[np.cos(theta) , -np.sin(theta) ,0],
                     [np.sin(theta) , np.cos(theta) ,0],
                     [ 0             , 0              ,1]])

def trans(x,y):
    return np.array([
        [1,0,x],
        [0,1,y],
        [0,0,1],
    ])

def get_x_y(T):
    return T[0,2] , T[1,2]
```

```
    x_axis = np.array([0.3, 0.0])
    y_axis = np.array([0.0, 0.3])
    # Use homogeneous tranformation to transfer these two axes/points
    # back to the fixed frame
    frame_a_x_axis = np.zeros((2,N))
    frame_a_y_axis = np.zeros((2,N))

    f_bc_origin = np.zeros((2,N))
    f_d_origin = np.zeros((2,N))
    f_b_x_tip = np.zeros((2,N))
    f_b_y_tip = np.zeros((2,N))
    f_c_x_tip = np.zeros((2,N))
    f_c_y_tip = np.zeros((2,N))
    f_d_x_tip = np.zeros((2,N))
    f_d_y_tip = np.zeros((2,N))

    for i in range(N): # iteration through each time step
        # evaluate homogeneous transformation
        t_wa = np.array([[np.cos(theta_array[0][i]), -np.
↪sin(theta_array[0][i]), 0],
                         [np.sin(theta_array[0][i]),  np.
↪cos(theta_array[0][i]), 0],
                         [                        0,                          ␣
↪0, 1]])

        theta1 = theta_array[0][i]
        theta2 = theta_array[1][i]
        T_WA = rot(theta1)
        T_AB = trans(0,-1)
        T_BC = rot(theta2)
        T_CD = trans(0,-1)

        T_WB = T_WA @ T_AB
        T_WC = T_WB @ T_BC
        T_WD = T_WC @ T_CD

        # transfer the x and y axes in body frame back to fixed frame at
        # the current time step
        frame_a_x_axis[:,i] = t_wa.dot([x_axis[0], x_axis[1], 1])[0:2]
        frame_a_y_axis[:,i] = t_wa.dot([y_axis[0], y_axis[1], 1])[0:2]

        f_bc_origin[:,i] = get_x_y( T_WB )
        f_d_origin[:,i] = get_x_y( T_WD )
        f_b_x_tip[:,i] = get_x_y( T_WB @ trans(0.3,0) )
        f_b_y_tip[:,i] = get_x_y( T_WB @ trans(0,0.3) )
        f_c_x_tip[:,i] = get_x_y( T_WC @ trans(0.3,0) )
```

```
        f_c_y_tip[:,i] = get_x_y( T_WC @ trans(0,0.3) )
        f_d_x_tip[:,i] = get_x_y( T_WD @ trans(0.3,0) )
        f_d_y_tip[:,i] = get_x_y( T_WD @ trans(0,0.3) )

    #####################################
    # Using these to specify axis limits.
    xm = -3 #np.min(xx1)-0.5
    xM = 3 #np.max(xx1)+0.5
    ym = -3 #np.min(yy1)-2.5
    yM = 3 #np.max(yy1)+1.5

    ###########################
    # Defining data dictionary.
    # Trajectories are here.
    data=[
        # note that except for the trajectory (which you don't need this time),
        # you don't need to define entries other than "name". The items defined
        # in this list will be related to the items defined in the "frames" list
        # later in the same order. Therefore, these entries can be considered as
        # labels for the components in each animation frame
        # dict(name='Arm',x=xx1, y=yy1,mode='lines', line=dict(width=2,␣
↪color='blue')),
          #    There is a bug in local version of this. I have to give it some␣
↪fake data for the link to show up later
        dict(name='Arm',x=xx1, y=yy1,mode='lines'),
        dict(name='Mass 1'),
        dict(name='Mass 2'),
        dict(name='World Frame X',x=xx1, y=yy1,mode='lines'),
        dict(name='World Frame Y',x=xx1, y=yy1,mode='lines'),
        dict(name='A Frame X Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='A Frame Y Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='B Frame X Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='B Frame Y Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='C Frame X Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='C Frame Y Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='D Frame X Axis',x=xx1, y=yy1,mode='lines'),
        dict(name='D Frame Y Axis',x=xx1, y=yy1,mode='lines'),

        # You don't need to show trajectory this time,
        # but if you want to show the whole trajectory in the animation (like␣
↪what
        # you did in previous homeworks), you will need to define entries other␣
↪than
        # "name", such as "x", "y". and "mode".

        # dict(x=xx1, y=yy1,
        #      mode='markers', name='Pendulum 1 Traj',
```

```
    #      marker=dict(color="fuchsia", size=2)
    #     ),
    # dict(x=xx2, y=yy2,
    #      mode='markers', name='Pendulum 2 Traj',
    #      marker=dict(color="purple", size=2)
    #     ),
    ]

################################
# Preparing simulation layout.
# Title and axis ranges are here.
layout=dict(autosize=False, width=1000, height=1000,
            xaxis=dict(range=[xm, xM], autorange=False,␣
↪zeroline=False,dtick=1),
            yaxis=dict(range=[ym, yM], autorange=False,␣
↪zeroline=False,scaleanchor = "x",dtick=1),
            title='Double Pendulum 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.
# This is what draws the lines from
# joint to joint of the pendulum.
frames=[dict(data=[# first three objects correspond to the arms and two␣
↪masses,
                   # same order as in the "data" variable defined above␣
↪(thus
                   # they will be labeled in the same order)
                   dict(x=[0,xx1[k],xx2[k]],
                        y=[0,yy1[k],yy2[k]],
                        mode='lines',
                        line=dict(color='orange', width=3),
                        ),
                   go.Scatter(
                        x=[xx1[k]],
```

```
                        y=[yy1[k]],
                        mode="markers",
                        marker=dict(color="blue", size=12)),
                    go.Scatter(
                        x=[xx2[k]],
                        y=[yy2[k]],
                        mode="markers",
                        marker=dict(color="blue", size=12)),
                    # display x and y axes of the fixed frame in each␣
↪animation frame
                    dict(x=[0,x_axis[0]],
                        y=[0,x_axis[1]],
                        mode='lines',
                        line=dict(color='green', width=3),
                        ),
                    dict(x=[0,y_axis[0]],
                        y=[0,y_axis[1]],
                        mode='lines',
                        line=dict(color='red', width=3),
                        ),
                    # display x and y axes of the {A} frame in each␣
↪animation frame
                    dict(x=[0, frame_a_x_axis[0][k]],
                        y=[0, frame_a_x_axis[1][k]],
                        mode='lines',
                        line=dict(color='green', width=3),
                        ),
                    dict(x=[0, frame_a_y_axis[0][k]],
                        y=[0, frame_a_y_axis[1][k]],
                        mode='lines',
                        line=dict(color='red', width=3),
                        ),
                    dict(x=[f_bc_origin[0][k], f_b_x_tip[0][k]],
                        y=[f_bc_origin[1][k], f_b_x_tip[1][k]],
                        mode='lines',line=dict(color='green', width=3),),
                    dict(x=[f_bc_origin[0][k], f_b_y_tip[0][k]],
                        y=[f_bc_origin[1][k], f_b_y_tip[1][k]],
                        mode='lines',line=dict(color='red', width=3),),
                    dict(x=[f_bc_origin[0][k], f_c_x_tip[0][k]],
                        y=[f_bc_origin[1][k], f_c_x_tip[1][k]],
                        mode='lines',line=dict(color='green', width=3),),
                    dict(x=[f_bc_origin[0][k], f_c_y_tip[0][k]],
                        y=[f_bc_origin[1][k], f_c_y_tip[1][k]],
                        mode='lines',line=dict(color='red', width=3),),
                    dict(x=[f_d_origin[0][k], f_d_x_tip[0][k]],
                        y=[f_d_origin[1][k], f_d_x_tip[1][k]],
                        mode='lines',line=dict(color='green', width=3),),
```

```
                        dict(x=[f_d_origin[0][k], f_d_y_tip[0][k]],
                             y=[f_d_origin[1][k], f_d_y_tip[1][k]],
                             mode='lines',line=dict(color='red', width=3),),


                       ]) for k in range(N)]

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

animate_double_pend(traj , T = 3)
```

```
<IPython.core.display.HTML object>
```

### 1.5 Collaboration list

- Srikanth Schelbert
- Graham Clifford
- Ananya Agarwal

The notebook is generated locally. Thus no google collab is available.