Homework 3

Sean Morton, ME449

Introduction

My submission is broken down into helper functions, part 1, part 2, and part 3. Each part of the submission has a cell block where a simulation runs, a block with outputs that describe the simulation and the resulting trajectory, and a written description.

Simulations in my code took a long time to run - each simulation took about 2 minutes in the real world. This might be improved by running the code as a .py file and not in .ipynb (as someone suggested to me), but I'm not sure. If you have any suggestions for code optimization, please let me know.

Helper functions

In [1]:
```
#imports
import numpy as np
import core as mr
import matplotlib.pyplot as plt
import time

from IPython.display import Markdown, display
```

In [2]:
```
#parameters of the UR5 robot

M01 = [[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0.089159],
       [0, 0, 0, 1]]

M12 = [[0, 0, 1, 0.28],
       [0, 1, 0, 0.13585],
       [-1, 0, 0, 0],
       [0, 0, 0, 1]]

M23 = [[1, 0, 0, 0],
       [0, 1, 0, -0.1197],
       [0, 0, 1, 0.395],
       [0, 0, 0, 1]]

M34 = [[0, 0, 1, 0],
       [0, 1, 0, 0],
       [-1, 0, 0, 0.14225],
       [0, 0, 0, 1]]

M45 = [[1, 0, 0, 0],
       [0, 1, 0, 0.093],
       [0, 0, 1, 0],
       [0, 0, 0, 1]]
```

```
M56 = [[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0.09465],
       [0, 0, 0, 1]]

M67 = [[1, 0, 0, 0],
       [0, 0, 1, 0.0823],
       [0, -1, 0, 0],
       [0, 0, 0, 1]]

G1 = np.diag([0.010267495893, 0.010267495893,  0.00666, 3.7, 3.7, 3.7])
G2 = np.diag([0.22689067591, 0.22689067591, 0.0151074, 8.393, 8.393, 8.393])
G3 = np.diag([0.049443313556, 0.049443313556, 0.004095, 2.275, 2.275, 2.275])
G4 = np.diag([0.111172755531, 0.111172755531, 0.21942, 1.219, 1.219, 1.219])
G5 = np.diag([0.111172755531, 0.111172755531, 0.21942, 1.219, 1.219, 1.219])
G6 = np.diag([0.0171364731454, 0.0171364731454, 0.033822, 0.1879, 0.1879, 0.1879])
```

In [3]:
```
Glist = [G1, G2, G3, G4, G5, G6]
Mlist = [M01, M12, M23, M34, M45, M56, M67]

Slist = [[0,         0,         0,         0,        0,        0],
         [0,         1,         1,         1,        0,        1],
         [1,         0,         0,         0,       -1,        0],
         [0, -0.089159, -0.089159, -0.089159, -0.10915, 0.005491],
         [0,         0,         0,         0,  0.81725,        0],
         [0,         0,     0.425,   0.81725,        0,  0.81725]]
```

In [4]:
```
#helper functions
def write_csv_line(csv_filename, data):
    with open(csv_filename, 'a') as f:
        data_str = ','.join([str(i) for i in data]) + '\n'
        f.write(data_str)
###

def write_csv_mat(csv_filename, mat):
    f = open(csv_filename, 'w') #clear out old data
    f.close()
    for row in mat:
            write_csv_line(csv_filename, row)
###

def ModEulerStep(thetalist, dthetalist, ddthetalist, dt):
    """EulerStep from the MR library, but with an additional
    second-order term from acceleration contributing to changes in position.
    """
    return thetalist + (dt * np.array(dthetalist) + 0.5 * dt**2 * np.array(dthetali
           dthetalist + dt * np.array(ddthetalist)
###

def filter_nan(array):
    """Unstable arrays may have NAN terms in them, which cannot be interpreted
    in CoppeliaSim. This function truncates an array to include only
    the portion of the array before the first NAN is detected.
    """
    firstnan = None
    for i in range(len(array)):
        row = array[i,:]
        nanmask = np.isnan(row)
        if np.any(nanmask): #if NAN is in row; "True" in nanmask
            firstnan = i
```

```
            break

    if firstnan:
        array = array[:firstnan]
    return array
```

In [5]:

```
#component functions for spring force, damping force, and simulation

def SpringForce(Slist, thetalist, Mlist, stiffness, springPos, restLength):
    '''
    - Calculates the "spring force" acting on the end effector
        of the robot, using position of end of spring, position
        of end effector, and spring parameters.
    - takes in:
        -Slist: list of screw axes in space frame
        -thetalist: list of current joint angles
        -Mlist: home configurations of each joint rel. to.
            each other
        -stiffness: scalar with spring constant
        -springPos: 3-vector position
        -restLength: scalar with resting length of spring
    - calls:
        - T = FKinBody(M_endeff, Blist, thetalist)
        - [R,p] = TranstoRp(FKinBody)
        - x_diff = np.linalg.norm(deltap) - restLength
    - returns:
        Ftip, a 6x1 end-effector wrench caused by the spring force
    '''

    # - Find home configuration of end effector
    M_matrices = [np.matrix(M) for M in Mlist]
    M_ee = M_matrices[0] * \
            M_matrices[1] * \
            M_matrices[2] * \
            M_matrices[3] * \
            M_matrices[4] * \
            M_matrices[5] * \
            M_matrices[6]

    #     - apply product of exponentials in space frame
    Tsb = mr.FKinSpace(M_ee, Slist, thetalist)
    springPos_4_s = np.append(np.array(springPos), [1]) #add an extra 1 so we can m
    springPos_4_s = np.matrix(springPos_4_s).T

    pb = mr.TransInv(Tsb) * springPos_4_s
    pb = np.array(pb[0:3].tolist())

    #springposn minus posn of end effector, in b,  will just be springposn - [0 0 0
    dp = pb[:]

    #direction of force given by unit vector of springposn
    unit_dir = -dp / np.linalg.norm(dp)

    #magnitude of force given by (stiffness) * (norm(springPosn_b) - restlength)
    mag_force = stiffness * (np.linalg.norm(pb) - restLength)
    sf = (mag_force * unit_dir).T[0].tolist()
    Fb = np.array([0, 0, 0, sf[0], sf[1], sf[2]])

    return Fb.tolist()

def DampingForce(B, thetad_list):
```

```
    '''
    - Make a torque at each joint, equal to -B * w
    - takes in:
        - damping constant B
        - thetad_list: an nd-array, where n = # of joints of robot
    - returns:
        - an nd-array of joint torques, where n = # of joints
    '''
    tau_list = -B * np.array(thetad_list)
    return tau_list.tolist()
```

In [6]:

```
#testing: plug in joint angles that give known positions; check that
#position difference is close to expected values; force is a constant times that
M_matrices = [np.matrix(M) for M in Mlist]
M_ee = M_matrices[0] * \
        M_matrices[1] * \
        M_matrices[2] * \
        M_matrices[3] * \
        M_matrices[4] * \
        M_matrices[5] * \
        M_matrices[6]

robot_posn = [0.1, 0.3, 0.6]
R = np.identity(3)
T = mr.RpToTrans(R, robot_posn)
thetalist0 = [0.504, -0.168, -1.713, 4.131, 0.000, 0.000]
thetalist, success = mr.IKinSpace(Slist, M_ee, T, thetalist0, 0.001, 0.0001)
thetalist = thetalist.round(4).tolist()

# print(M_ee) #looks as expected in CoppeliaSim
# print(thetalist) #[0.8966, -0.0886, -1.5636, 3.2229, -1.5708, 0.6742]
# print(success)

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

spring_posn = [0.2, 0.4, 0.7]
stiffness = 1

#check that spring force matches expected value - zero restLength
restLength = 0
expected_wrench = [0, 0, 0, -0.1, -0.1, -0.1]
wrench = SpringForce(Slist, thetalist, Mlist, stiffness, spring_posn, restLength)
assert np.allclose(expected_wrench, wrench, rtol = 1E-4, atol = 1E-4)

#with nonzero restLength
restLength = np.sqrt(3)/10
wrench = SpringForce(Slist, thetalist, Mlist, stiffness, spring_posn, restLength)
expected_wrench = np.array([0, 0, 0, 0, 0, 0])
assert np.allclose(expected_wrench, wrench, rtol = 1E-4, atol = 1E-4) #---> this us

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



#test damping function
thetalist = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
damping = 0.2
assert np.allclose(DampingForce(damping, thetalist), [-0.02, -0.04, -0.06, -0.08, -

print("all assertions passed")
```

```
all assertions passed
```

In [7]:
```
def Puppet(damping, stiffness, restLength, springPos,\
           tf, dt, \
           g, Mlist, Glist, Slist, \
           thetalist0, thetadlist0):
    '''
    - takes in
        - time parameters tf, dt
        - damping, stiffness, restLength
        - g, Mlist, Glist, Slist (g may be [0,0,0])
        - thetalist0, dthetalist0
    - calculates
        - end effector wrench Ftip
        - applied joint torques taulist
    - calls
        - mr.ForwardDynamics(thetalist,dthetalist,taulist,
            g,Ftip,Mlist,Glist,Slist)
        - ModEulerStep(thetalist,dthetalist,ddthetalist,dt)
        - SpringForce()
        - DampingForce()
    - returns: a N x n matrix of joint values, N = # of timesteps,
        n = number of joints
    '''

    thetalist = np.array(thetalist0)
    thetadlist = np.array(thetadlist0)
    t_array = np.arange(0, tf, dt)
    traj_array = np.zeros([len(t_array), len(thetalist)]) #N timesteps, n angles

    for i, t in enumerate(t_array):

        #calculate forces; call ForwardDynamics() with starting values of t, td, ta
        taulist = DampingForce(damping, thetadlist)
        Fb_tip = SpringForce(Slist, thetalist, Mlist, stiffness, springPos, restLer

        thetaddlist = mr.ForwardDynamics(thetalist, thetadlist, taulist, \
                                         g, Fb_tip, Mlist, Glist, Slist)

        # - use numerical integration to find theta, thetad at next timestep
        thetalist_new, thetadlist_new = ModEulerStep(thetalist, thetadlist, thetadd

        # - store value of theta at next timestep in an array; reset theta and thet
        traj_array[i,:] = thetalist_new
        thetalist = thetalist_new
        thetadlist = thetadlist_new

    #may want to change names of inputs to exactly match the asst later
    return traj_array

#####
```

Part 1

Code:

In [29]:
```
#testing
damping_p1 = 0
stiffness_p1 = 0
```

```
restLength_p1 = 0
tf_p1 = 5
dt_p1_fine = 0.005

g_p1 = 9.81 * np.array([0, 0, -1])
thetalist0 = [0,0,0,0,0,0]
dthetalist0 =[0,0,0,0,0,0]
springPosn = [0, 0, 2]

realtime0 = time.time()
traj_p1_fine = Puppet(\
      damping_p1, stiffness_p1, restLength_p1, springPosn, \
      tf_p1, dt_p1_fine, g_p1, \
      Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [9]:
```
#process and save
display(Markdown("**Part 1a: Gravity, no spring/damping, small dt**"))
print(f"Shape of Traj array: {traj_p1_fine.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
print(f"dt: {dt_p1_fine}")
print(f"tf: {tf_p1}")
write_csv_mat("../csv/HW3_p1_grav_fine.csv",traj_p1_fine)
```

Part 1a: Gravity, no spring/damping, small dt

```
Shape of Traj array: (1000, 6)
Elapsed: 113.4 real-world seconds

dt: 0.005
tf: 5
```

In [10]:
```
#show a timestep that makes system unstable
dt_p1_coarse = 0.015
realtime0 = time.time()
traj_p1_coarse = Puppet( \
    damping_p1, stiffness_p1, restLength_p1, springPosn, \
    tf_p1, dt_p1_coarse, g_p1, \
    Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [11]:
```
#filter out NAN results
traj_p1_coarse = filter_nan(traj_p1_coarse)
```

In [12]:
```
#process and save
display(Markdown("**Part 1b: Gravity, no spring/damping, large dt**"))
print(f"Shape of Traj array: {traj_p1_coarse.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
print(f"dt: {dt_p1_coarse}")
print(f"tf: {tf_p1}")
write_csv_mat("../csv/HW3_p1_grav_coarse.csv",traj_p1_coarse)
```

Part 1b: Gravity, no spring/damping, large dt

```
Shape of Traj array: (334, 6)
Elapsed: 37.9 real-world seconds

dt: 0.015
tf: 5
```

Written:

The videos show the robot falls in gravity with a slow increase in energy of the system over time. For large timesteps dt, this growth in energy is substantial.

Part 2

Code:

In [13]:
```
#uses otherwise the same inputs as problem 1
damping_p2_pos = 3

realtime0 = time.time()
traj_p2_damped_pos = Puppet( \
    damping_p2_pos, stiffness_p1, restLength_p1, springPosn, \
    tf_p1, dt_p1_fine, g_p1, \
    Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [14]:
```
display(Markdown("**Part 2a:  Gravity and damping; positive damping**"))
print(f"Shape of Traj array: {traj_p2_damped_pos.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
write_csv_mat("../csv/HW3_p2_damped_pos.csv",traj_p2_damped_pos)

print(f"Damping coefficient used: {damping_p2_pos}")
print(f"Stiffness coefficient used: {stiffness_p1}")
print(f"dt: {dt_p1_fine}")
print(f"tf: {tf_p1}")
```

Part 2a: Gravity and damping; positive damping

```
Shape of Traj array: (1000, 6)
Elapsed: 111.2 real-world seconds

Damping coefficient used: 3
Stiffness coefficient used: 0
dt: 0.005
tf: 5
```

In [15]:
```
damping_p2_neg = -0.1

realtime0 = time.time()
traj_p2_damped_neg = Puppet( \
    damping_p2_neg, stiffness_p1, restLength_p1, springPosn, \
    tf_p1, dt_p1_fine, g_p1, \
    Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

```
C:\Users\seanp\Downloads\22F_MECH_ENG_449\hw3\code\core.py:927: RuntimeWarning: ov
erflow encountered in multiply
  + np.dot(ad(Vi[:, i + 1]), Ai[:, i]) * dthetalist[i]
C:\Users\seanp\Downloads\22F_MECH_ENG_449\hw3\code\core.py:143: RuntimeWarning: in
valid value encountered in sin
  return np.eye(3) + np.sin(theta) * omgmat \
C:\Users\seanp\Downloads\22F_MECH_ENG_449\hw3\code\core.py:144: RuntimeWarning: in
valid value encountered in cos
  + (1 - np.cos(theta)) * np.dot(omgmat, omgmat)
C:\Users\seanp\Downloads\22F_MECH_ENG_449\hw3\code\core.py:366: RuntimeWarning: in
valid value encountered in multiply
  np.dot(np.eye(3) * theta \
C:\Users\seanp\Downloads\22F_MECH_ENG_449\hw3\code\core.py:367: RuntimeWarning: in
valid value encountered in cos
  + (1 - np.cos(theta)) * omgmat \
C:\Users\seanp\Downloads\22F_MECH_ENG_449\hw3\code\core.py:368: RuntimeWarning: in
valid value encountered in sin
  + (theta - np.sin(theta)) \
```

In [16]:
```
#filter out NAN results
traj_p2_damped_neg = filter_nan(traj_p2_damped_neg)
```

In [17]:
```
display(Markdown("**Part 2b:  Gravity and damping; negative damping**"))
print(f"Shape of Traj array: {traj_p2_damped_neg.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
write_csv_mat("../csv/HW3_p2_damped_neg.csv",traj_p2_damped_neg)

print(f"Damping coefficient used: {damping_p2_neg}")
print(f"Stiffness coefficient used: {stiffness_p1}")
print(f"dt: {dt_p1_fine}")
print(f"tf: {tf_p1}")
```

Part 2b: Gravity and damping; negative damping

```
Shape of Traj array: (347, 6)
Elapsed: 121.3 real-world seconds

Damping coefficient used: -0.1
Stiffness coefficient used: 0
dt: 0.005
tf: 5
```

In [39]:
```
damping_p2_large = 8

realtime0 = time.time()
traj_p2_damped_large = Puppet( \
    damping_p2_large, stiffness_p1, restLength_p1, springPosn, \
    tf_p1, dt_p1_fine, g_p1, \
    Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [40]:
```
#filter out NAN results
traj_p2_damped_large = filter_nan(traj_p2_damped_large)
```

In [41]:
```
display(Markdown("**Part 2c:  Gravity and damping; large positive damping**"))
print(f"Shape of Traj array: {traj_p2_damped_large.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
write_csv_mat("../csv/HW3_p2_damped_large.csv",traj_p2_damped_large)

print(f"Damping coefficient used: {damping_p2_large}")
print(f"Stiffness coefficient used: {stiffness_p1}")
```

```
print(f"dt: {dt_p1_fine}")
print(f"tf: {tf_p1}")
```

Part 2c: Gravity and damping; large positive damping

```
Shape of Traj array: (50, 6)
Elapsed: 86.3 real-world seconds

Damping coefficient used: 8
Stiffness coefficient used: 0
dt: 0.005
tf: 5
```

Written:

Do you see any strange behavior in the simulation if you choose the damping constant to be a large positive value? Can you explain it?

When the damping constant is chosen to be a large positive value, the motion of the robot blows up, growing exponentially to incredibly fast speeds. The most likely cause for this behavior is that the joint torques calculated by the damping function are inversely proportional to the current joint velocities of the robot. When damping coefficient B is large, the acceleration caused by damping can cause the magnitude of velocity to increase from one timestep to the next, but the direction of velocity to flip. This repeated amplification of velocity can cause the motion of the robot to blow up to infinite velocity.

How would this behavior be affected if you chose shorter simulation timesteps?

If our simulation timestep dt were smaller but all other factors stayed the same, the motion of the robot might not blow up to infinite speeds as quickly, or at all. For any given values of dthetalist, the damping force -B * dthetalist is the same, independent of t. However, when dt is smaller, the update to dthetalist at the next timestep is small.

For example, for a system with linear velocity only, if thetad = 2, damping_accel = -8 and dt = 1, then thetad at next timestep = thetad + dt * damping_accel = -6; repeated iterations of this cause the velocity to blow up.

If only dt is changed, and thetad = 2, damping_accel = -8 and dt = 0.1, thetad at next timestep is thetad + dt * damping_accel = 1.2. The velocity has shrunk, so the velocity of the system should not blow up to infinity.

I'm leaving out consideration of inertia and other aspects that make rotational systems different, but I would think the same would be true of rotational systems with inertia: smaller dt = less likely to diverge.

Part 3

Code:

In [20]:
```
#params for problem:
g_p3 = [0,0,0]
springPosn = [0,0,2]
```

```
stiffness_p3_nd = 8
restLength_p3_nd = 0
damping_p3_nd = 0

tf_p3 = 10
dt_p3 = 0.01

realtime0 = time.time()
traj_p3_spring = Puppet( \
     damping_p3_nd, stiffness_p3_nd, restLength_p3_nd, springPosn, \
     tf_p3, dt_p3, g_p3,
     Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [21]:
```
display(Markdown("**Part 3a: Spring Force, No Damping**"))
print(f"Shape of Traj array: {traj_p3_spring.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
write_csv_mat("../csv/HW3_p3_spring.csv",traj_p3_spring)

print(f"\nDamping coefficient used: {damping_p3_nd}")
print(f"Stiffness coefficient used: {stiffness_p3_nd}")
print(f"dt: {dt_p3}")
print(f"tf: {tf_p3}")
```

Part 3a: Spring Force, No Damping

```
Shape of Traj array: (1000, 6)
Elapsed: 113.5 real-world seconds


Damping coefficient used: 0
Stiffness coefficient used: 8
dt: 0.01
tf: 10
```

In [22]:
```
#3b: modify damping and stiffness only
stiffness_p3_posd = 8
restLength_p3_posd = 0
damping_p3_posd = 3

tf_p3 = 10
dt_p3 = 0.01

realtime0 = time.time()
traj_p3_spring_damped = Puppet( \
    damping_p3_posd, stiffness_p3_posd, restLength_p3_posd, springPosn, \
    tf_p3, dt_p3, g_p3, \
    Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [23]:
```
display(Markdown("**Part 3b: Spring Force with Damping**"))
print(f"Shape of Traj array: {traj_p3_spring_damped.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
write_csv_mat("../csv/HW3_p3_spring_damped.csv",traj_p3_spring_damped)

print(f"Damping coefficient used: {damping_p3_posd}")
print(f"Stiffness coefficient used: {stiffness_p3_posd}")
print(f"dt: {dt_p3}")
print(f"tf: {tf_p3}")
```

Part 3b: Spring Force with Damping

```
Shape of Traj array: (1000, 6)
Elapsed: 101.2 real-world seconds

Damping coefficient used: 3
Stiffness coefficient used: 8
dt: 0.01
tf: 10
```

In [24]:
```
#part 3c
springPosn = [0,0,2]
stiffness_p3_bigd = 50
restLength_p3_bigd = 0
damping_p3_bigd = 0

tf_p3 = 10
dt_p3 = 0.01

realtime0 = time.time()
traj_p3_spring_large = Puppet( \
     damping_p3_bigd, stiffness_p3_bigd, restLength_p3_bigd, springPosn, \
     tf_p3, dt_p3, g_p3,
     Mlist, Glist, Slist, thetalist0, dthetalist0)
realtimef = time.time()
```

In [25]:
```
display(Markdown("**Part 3c: Spring Force, Large Stiffness**"))
print(f"Shape of Traj array: {traj_p3_spring_large.shape}")
print(f"Elapsed: {round(realtimef - realtime0,1)} real-world seconds\n")
write_csv_mat("../csv/HW3_p3_spring_large.csv",traj_p3_spring_large)

print(f"Damping coefficient used: {damping_p3_bigd}")
print(f"Stiffness coefficient used: {stiffness_p3_bigd}")
print(f"dt: {dt_p3}")
print(f"tf: {tf_p3}")
```

Part 3c: Spring Force, Large Stiffness

```
Shape of Traj array: (1000, 6)
Elapsed: 103.2 real-world seconds

Damping coefficient used: 0
Stiffness coefficient used: 50
dt: 0.01
tf: 10
```

Written:

Does the motion make sense to you?

The motion of the arm is consistent with a force being applied from the end effector of the robot towards a spring position of [0,0,2] in the space frame. The robot's consistent overshoot of the target position can be attributed to the inertia of each link in the open chain. The only part of the motion that surprised me was the consistent rotation of the last joint (to which the end effector is attached), but this may be due to the rotational inertia of the last few joints in the chain.

Should total energy be conserved?

Total energy should be conserved because there are no sources of energy into the system, and no damping forces that dissipate energy. At any given time the sum of kinetic energy and spring potential energy should be constant.

Does the total energy appear to be conserved?

Total energy appears to be increasing slightly. The last couple of motions of the robot arm show the arm swinging farther down after overshooting the target position. This may imply the Euler integration scheme is adding some energy at each timestep.

Do you see any strange behavior if you choose the spring constant to be large?

When I chose a spring constant of 50, I noticed that the motions of the robot arm did not always oscillate around the spring position as time went on. The robot arm started off with clear motion in the direction of the spring position, but then the system became more energetic, and the end effector accelerated in many different directions in rapid succession.

This behavior may be due to the system gaining energy. Given that the spring forces are higher, the accelerations of the joints are higher as well, so the error from EulerStep() may be greater for this test with greater spring stiffness.

In [ ]: