ME449 - Homework 3 - Zhengyang Kris Weng Submission

In [1]:
```
import numpy as np
import modern_robotics as mr
from tqdm import tqdm
import ur5_parameters
ur5 = ur5_parameters.UR5()
```

Part0: Setup

This homework is done using a jupyter notebook `Weng_Zhengyang_asst3.ipynb` . To generate all the .csv trajectories necessary for simulation, run all the code blocks in this .ipynb notebook. Make sure files `ur5_parameters.py` is under the same directory as this file.

Please find the answers to the corresponding part in the MarkDown cells following code blcoks.

Part 1: Simulating a falling robot.

In the first part, the robot will fall in gravity without damping or the external spring (joint damping and spring stiffness are set to zero). Since there is no damping or friction, the total energy of the robot (kinetic plus potential) should remain constant during motion. Gravity is g = 9.81 m/s2 in the –ˆzs-direction, i.e., gravity acts downward. Simulate the robot falling from rest at the home configuration for five seconds. The output data should be saved as a .csv file, where each of the N rows has six numbers separated by commas. This .csv file is suitable for animation with the CoppeliaSim UR5 csv animation scene. Adjust the animation scene playback speed (“Time Multiplier”) so it takes roughly five seconds of wall clock time to play your csv file. You can evaluate if your simulation is preserving total energy by visually checking if the robot appears to swing to the same height (same potential energy) each swing. Choose values of dt (a) where the energy appears nearly constant (without choosing dt unnecessarily small) and (b) where the energy does not appear constant (because your timestep is too coarse). Capture a video for each case and note the dt chosen for each case. Explain how you would calculate the total energy of the robot at each timestep if you wanted to plot the total energy to confirm that your simulation approximately preserves it.

In [5]:
```
# Puppet function:
def puppet_q1(thetalist, dthetalist, g, Mlist, Slist, Glist, t, dt, damping, stiffness, restLen
    """
    Simulate a robot under damping and spring reaction. Q1: free falling in gravity

    Args:
        thetalist (np.array): n-vector of initial joint angles (rad)
        dthetalist (np.array): n-vector of initial joint velocities (rad/s)
        g (np.array): 3-vector of gravity in s frame (m/s^2)
        Mlist (np.array): 8 frames of link configuration at home pose
        Slist (np.array): 6-vector of screw axes at home configuration
        Glist (np.array): Spatial inertia matrices of the links
        t (float): total simulation time (s)
        dt (float): simulation time step (s)
        damping (float): viscous damping coefficient (Nmn/rad)
        stiffness (float): spring stiffness coefficient (N/m)
        restLength (float): length of the spring at rest (m)
    Returns:
        thetamat (np.array): N x n matrix of joint angles (rad). Each row is a set of joint ang
        dthetamat (np.array): N x n matrix of joint velocities (rad/s). Each row is a set of jo
    """
    # Initialize
    N = int(t/dt)
    n = len(thetalist)
    thetamat = np.zeros((N + 1, n))
    dthetamat = np.zeros((N + 1, n))
    thetamat[0] = thetalist
    dthetamat[0] = dthetalist

    for i in tqdm(range(N)):
        i_acc = mr.ForwardDynamics(thetalist, dthetalist, np.zeros(n), g, np.zeros(n), Mlist, G
        i_pos, i_vel = mr.EulerStep(thetalist, dthetalist, i_acc, dt)
        thetamat[i + 1] = i_pos
        dthetamat[i + 1] = i_vel

        # Update
        thetalist = i_pos
        dthetalist = i_vel

    return thetamat, dthetamat
```

In [6]:
```
q1_thetalist0 = np.array([0, 0, 0, 0, 0, 0])
q1_dthetalist0 = np.array([0, 0, 0, 0, 0, 0])
g = np.array([0, 0, -9.81])

q1_thetamat, _ = puppet_q1(q1_thetalist0, q1_dthetalist0, g, ur5.Mlist, ur5.Slist, ur5.Glist, 5

# Save to csv file
np.savetxt('q1_thetamat.csv', q1_thetamat, delimiter=',')
```

```
100%|██████████| 5000/5000 [00:32<00:00, 156.21it/s]
```

In [7]:
```
q1_thetamat_coarse, _ = puppet_q1(q1_thetalist0, q1_dthetalist0, g, ur5.Mlist, ur5.Slist, ur5.G

# Save to csv file
np.savetxt('q1_thetamat_coarse.csv', q1_thetamat_coarse, delimiter=',')
```

```
100%|██████████| 500/500 [00:03<00:00, 156.37it/s]
```

Part 1 response:

Loooking at video part1a and part1b.mp4, we can see that energy is not conserved due to error in numerical integration when timestep is too coarse. A good measure of total system energy would be the Hamiltonian of the system, as a sum of potential energy and kinetic energy of each link. By plotting the trend of Hamiltonian of each configuration in the trajectory, this will visualize the preservation of energy in the system. [TODO: IMPLEMENT COMPUTE H WHEN HAVE TIME]

Part 2: Adding damping.

Now experiment with different damping coefficients as the robot falls from the home configuration. Damping causes a torque at each joint equal to the negative of the joint rate times the damping. Create two videos showing that (a) when you choose damping to be positive, the robot loses energy as it swings, and (b) when you choose damping to be negative, the robot gains energy as it swings. Use t = 5 s and dt = 0.01 s, and for the case of positive damping, the damping coefficient should almost (but not quite) bring the robot to rest by the end of the video. 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? How would this behavior change if you chose shorter simulation timesteps?

In [8]:
```
def puppet_q2(thetalist, dthetalist, g, Mlist, Slist, Glist, t, dt, damping, stiffness, restLen
    """
    Simulate a robot under damping and spring reaction. Q2: Adding damping to robot. Damping ca

    Args:
        thetalist (np.array): n-vector of initial joint angles (rad)
        dthetalist (np.array): n-vector of initial joint velocities (rad/s)
        g (np.array): 3-vector of gravity in s frame (m/s^2)
        Mlist (np.array): 8 frames of link configuration at home pose
        Slist (np.array): 6-vector of screw axes at home configuration
        Glist (np.array): Spatial inertia matrices of the links
        t (float): total simulation time (s)
        dt (float): simulation time step (s)
        damping (float): viscous damping coefficient (Nmn/rad)
        stiffness (float): spring stiffness coefficient (N/m)
        restLength (float): length of the spring at rest (m)
    Returns:
        thetamat (np.array): N x n matrix of joint angles (rad). Each row is a set of joint ang
        dthetamat (np.array): N x n matrix of joint velocities (rad/s). Each row is a set of jo
    """
    # Initialize
    N = int(t/dt)
    n = len(thetalist)
    thetamat = np.zeros((N + 1, n))
    dthetamat = np.zeros((N + 1, n))
    thetamat[0] = thetalist
    dthetamat[0] = dthetalist

    for i in tqdm(range(N)):
        tau_damping = - damping * dthetalist
        i_acc = mr.ForwardDynamics(thetalist, dthetalist, tau_damping, g, np.zeros(n), Mlist, G
        i_pos, i_vel = mr.EulerStep(thetalist, dthetalist, i_acc, dt)
        thetamat[i + 1] = i_pos
        dthetamat[i + 1] = i_vel

        # Update
        thetalist = i_pos
        dthetalist = i_vel

    return thetamat, dthetamat
```

In [9]:
```
q2_thetalist0 = np.array([0, 0, 0, 0, 0, 0])
q2_dthetalist0 = np.array([0, 0, 0, 0, 0, 0])
g = np.array([0, 0, -9.81])

q2_thetamat, _ = puppet_q2(q2_thetalist0, q2_dthetalist0, g, ur5.Mlist, ur5.Slist, ur5.Glist, 5
# Save to csv file
np.savetxt('q2_thetamat.csv', q2_thetamat, delimiter=',')
```

```
100%|██████████| 500/500 [00:03<00:00, 156.30it/s]
```

In [10]:
```
q2_thetamat_neg, _ = puppet_q2(q2_thetalist0, q2_dthetalist0, g, ur5.Mlist, ur5.Slist, ur5.Glis
# Save to csv file
print(q2_thetamat_neg)
np.savetxt('q2_thetamat_neg.csv', q2_thetamat_neg, delimiter=',')
```

```
100%|██████████| 500/500 [00:03<00:00, 156.56it/s]
```

```
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  2.57237340e-03 -2.87368129e-03  3.01307887e-04
   3.18411925e-22 -1.13423177e-18]
 ...
 [-3.07812861e-01 -9.92195181e+00  1.31700123e+01 -6.81968171e+00
  -8.36251555e-01  5.89264993e+01]
 [-3.02987560e-01 -1.00110250e+01  1.32499198e+01 -6.84260777e+00
  -8.76917115e-01  5.96387109e+01]
 [-2.97193787e-01 -1.00998345e+01  1.33216146e+01 -6.85772600e+00
  -9.17977817e-01  6.03581957e+01]]
```

In [11]:

```
q2_thetamat_large, _ = puppet_q2(q2_thetalist0, q2_dthetalist0, g, ur5.Mlist, ur5.Slist, ur5.Gl
# Save to csv file
print(q2_thetamat_large)
np.savetxt('q2_thetamat_large.csv', q2_thetamat_large, delimiter=',')
```

```
  0%|          | 0/500 [00:00<?, ?it/s]/usr/lib/python3/dist-packages/modern_robotics/core.py:9
27: RuntimeWarning: overflow encountered in multiply
  + np.dot(ad(Vi[:, i + 1]), Ai[:, i]) * dthetalist[i]
  3%|▎         | 17/500 [00:00<00:02, 165.73it/s]/usr/lib/python3/dist-packages/modern_robotic
s/core.py:143: RuntimeWarning: invalid value encountered in sin
  return np.eye(3) + np.sin(theta) * omgmat \
/usr/lib/python3/dist-packages/modern_robotics/core.py:144: RuntimeWarning: invalid value encou
ntered in cos
  + (1 - np.cos(theta)) * np.dot(omgmat, omgmat)
/usr/lib/python3/dist-packages/modern_robotics/core.py:366: RuntimeWarning: invalid value encou
ntered in multiply
  np.dot(np.eye(3) * theta \
/usr/lib/python3/dist-packages/modern_robotics/core.py:367: RuntimeWarning: invalid value encou
ntered in cos
  + (1 - np.cos(theta)) * omgmat \
/usr/lib/python3/dist-packages/modern_robotics/core.py:368: RuntimeWarning: invalid value encou
ntered in sin
  + (theta - np.sin(theta)) \
100%|██████████| 500/500 [00:03<00:00, 155.04it/s]
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  2.57237340e-03 -2.87368129e-03  3.01307887e-04
   3.18411925e-22 -1.13423177e-18]
 ...
 [            nan             nan             nan             nan
              nan             nan]
 [            nan             nan             nan             nan
              nan             nan]
 [            nan             nan             nan             nan
              nan             nan]]
```

Part 2 Response:

As I increase the magnitude of damping coefficient in the simulation, the function start to run into numerical stability issues and starts to produce nan values in output. This happend because of the numerical instability in the euler integration and the fact that coarse timestep landing next iteration on gradients amplifying such effect; as a result simulation output grew at an extremely fast rate and overflowed. Increasing granularity in timestep helps addressing this issue.

Part 3: Adding a spring.

Make gravity and damping zero and design referencePos to return a constant springPos at (0, 1, 1) in the {s} frame. The spring's restLength is zero. Experiment with different stiffness values, and simulate the robot for t = 10 s and dt = 0.01 s starting from the home configuration. (a) Capture a video for a choice of stiffness that makes the robot oscillate a couple of times and record the stiffness value. Considering the system's total energy, does the motion of the robot make sense? What do you expect to happen to the total energy over time? Describe the strange behavior you see if you choose the spring constant to be large; if you don't see any strange behavior, explain why. (b) Now add a positive damping to the simulation that makes the arm nearly come to rest by the end of the video. For both videos, record the stiffness and damping you used.

In [5]:
```
def puppet_q3(thetalist, dthetalist, g, Mlist, Slist, Glist, t, dt, damping, stiffness, restLen
    """
    Simulate a robot under damping and spring reaction. Q3: Adding a static spring.

    Args:
        thetalist (np.array): n-vector of initial joint angles (rad)
        dthetalist (np.array): n-vector of initial joint velocities (rad/s)
        g (np.array): 3-vector of gravity in s frame (m/s^2)
        Mlist (np.array): 8 frames of link configuration at home pose
        Slist (np.array): 6-vector of screw axes at home configuration
        Glist (np.array): Spatial inertia matrices of the links
        t (float): total simulation time (s)
        dt (float): simulation time step (s)
        damping (float): viscous damping coefficient (Nmn/rad)
        stiffness (float): spring stiffness coefficient (N/m)
        restLength (float): length of the spring at rest (m)
    Returns:
        thetamat (np.array): N x n matrix of joint angles (rad). Each row is a set of joint ang
        dthetamat (np.array): N x n matrix of joint velocities (rad/s). Each row is a set of jo
    """
    # Initialize
    N = int(t/dt)
    n = len(thetalist)
    thetamat = np.zeros((N + 1, n))
    dthetamat = np.zeros((N + 1, n))
    thetamat[0] = thetalist
    dthetamat[0] = dthetalist

    for i in tqdm(range(N)):
        # Calculate damping
        # print(f"Iteration {i}")
        tau_damping = - damping * dthetalist
        # Calculate spring force
        spring_force_vec = calculate_spring_wrench(thetalist, Slist, stiffness, restLength, ref
        # print(spring_force_vec)
        # Forward dynamics
        i_acc = mr.ForwardDynamics(thetalist, dthetalist, tau_damping, g, spring_force_vec, Mli
        i_pos, i_vel = mr.EulerStep(thetalist, dthetalist, i_acc, dt)
        thetamat[i + 1] = i_pos
        dthetamat[i + 1] = i_vel

        # Update
        thetalist = i_pos
        dthetalist = i_vel

    return thetamat, dthetamat

def referencePos_q3(t):
    """
    Generate a reference position for springPos

    Args:
        t (float): current time (s)
    Returns:
        np.array: 3-vector of reference position
    """
    return np.array([0, 1, 1])

def calculate_spring_wrench(thetalist, Slist, stiffness, restLength, springPos):
    """
    Calculate the 6-vector spring wrench acting on the end-effector.

    Args:
        thetalist (np.array): n-vector of joint angles (rad)
        Mlist (np.array): 8 frames of link configuration at home pose
        stiffness (float): spring stiffness coefficient (N/m)
        restLength (float): length of the spring at rest (m)
        springPOs (np.array): 3-vector of spring position in {s} frame
    Returns:
        np.array: 6-vector of spring forces and torque acting on the robot. Expressed in end-ef
    """
    # Get end effector transformation matrix for current configuration

    eePos = mr.FKinSpace(ur5.M_EE, Slist, thetalist)
    # print(f"eePos = {eePos}")
    # Extract position vector (first 3 elements of last column)
    p = np.array(eePos[:3,3])
```

```
    # Calculate spring length
    spring_length = np.linalg.norm(p - springPos) - restLength
    # print(f"spring_length = {spring_length}")
    # print(f"expected spring force = {stiffness * spring_length}")

    # Calculate spring force vector in {s} frame
    spring_force = - stiffness * spring_length * (springPos - p) / np.linalg.norm(p - springPos
    # print(f"spring_force = {spring_force}")
    # print(f"norm = {np.linalg.norm(spring_force)}")

    # Convert to end effector frame: T_{ee}^{s} * F_{s}
    spring_force_ee = mr.TransInv(eePos) @ np.array([*spring_force, 1]).T
    # print(f"spring_force_ee = {spring_force_ee}")
    # print(f"norm = {np.linalg.norm(spring_force_ee[:3])}")

    spring_wrench_ee = np.array([0, 0, 0, *spring_force_ee[:3]])
    return spring_wrench_ee
```

In [6]:
```
q3_thetalist0 = np.array([0, 0, 0, 0, 0, 0])
q3_dthetalist0 = np.array([0, 0, 0, 0, 0, 0])
g_q3 = np.array([0, 0, 0])

q3_thetamat, _ = puppet_q3(q3_thetalist0, q3_dthetalist0, g_q3, ur5.Mlist, ur5.Slist, ur5.Glist
# Save to csv file
# print(q3_thetamat)
np.savetxt('q3_thetamat.csv', q3_thetamat, delimiter=',')
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]100%|██████████| 1000/1000 [00:07<00:00, 131.99it/s]
```

In [7]:
```
q3_thetamat_damp, _ = puppet_q3(q3_thetalist0, q3_dthetalist0, g_q3, ur5.Mlist, ur5.Slist, ur5.(
# Save to csv file
# print(q3_thetamat_damp)
np.savetxt('q3_thetamat_damp.csv', q3_thetamat_damp, delimiter=',')
```

```
100%|██████████| 1000/1000 [00:07<00:00, 133.29it/s]
```

Part 3 Response:

From the simulation in part3a, since the total energy of the system is mostly conserved, we expect the robot to swing (under the effect of the spring) to about the same level. The motion in the simulation makes sense in this way. The total energy, when simulated at a reasonable timestep, should be roughly conserved over time. Now, if the spring constant becomes too large, the simulation becomes again very unstable and the robot spins out of control.

For the first video, part3a, there's no damping, and stiffness = 10 N/m. For the second video with damping, damping = 2 Nm/rad, and stiffness = 10 N/m

Part 4: A moving spring.

Use the joint damping and spring stiffness from Part 3(b), a spring restLength of zero, and zero gravity. Now set referencePos to return a sinusoidal motion of springPos. springPos should sinusoidally oscillate along a line, starting from one endpoint at (1, 1, 1) to another endpoint at (1, −1, 1), completing two full back-and-forth cycles in 10 s. Simulate with the robot starting at the home configuration for t = 10 s with dt = 0.01 s and create a movie of the simulation

In [8]:
```
def referencePos_q4(t):
    """
    Generate a reference position for springPos that oscillates sinusoidally along a line
    Args:
        t (float): current time (s)
    Returns:
        np.array: 3-vector of reference position
    """
    # Start point: (1, 1, 1)
    # End point: (1, -1, 1)
    # 2 full cycles in 10s means angular frequency = 4π/10 rad/s

    # Only y-coordinate varies, x and z stay constant at 1
    omega = 4 * np.pi / 10  # angular frequency for 2 cycles in 10s
    y = np.cos(omega * t)  # oscillates between 1 and -1
    ref_point = np.array([1, y, 1])
    # print(f"SpringPos = {ref_point}")
    return ref_point

def puppet_q4(thetalist, dthetalist, g, Mlist, Slist, Glist, t, dt, damping, stiffness, restLen
    """
    Simulate a robot under damping and spring reaction. Q3: Adding a static spring.

    Args:
        thetalist (np.array): n-vector of initial joint angles (rad)
        dthetalist (np.array): n-vector of initial joint velocities (rad/s)
        g (np.array): 3-vector of gravity in s frame (m/s^2)
        Mlist (np.array): 8 frames of link configuration at home pose
        Slist (np.array): 6-vector of screw axes at home configuration
        Glist (np.array): Spatial inertia matrices of the links
        t (float): total simulation time (s)
        dt (float): simulation time step (s)
        damping (float): viscous damping coefficient (Nmn/rad)
        stiffness (float): spring stiffness coefficient (N/m)
        restLength (float): length of the spring at rest (m)
    Returns:
        thetamat (np.array): N x n matrix of joint angles (rad). Each row is a set of joint ang
        dthetamat (np.array): N x n matrix of joint velocities (rad/s). Each row is a set of jo
    """
    # Initialize
    N = int(t/dt)
    n = len(thetalist)
    thetamat = np.zeros((N + 1, n))
    dthetamat = np.zeros((N + 1, n))
    thetamat[0] = thetalist
    dthetamat[0] = dthetalist

    for i in tqdm(range(N)):
        # Calculate damping
        # print(f"Iteration {i}")
        tau_damping = - damping * dthetalist
        # Calculate spring force
        spring_force_vec = calculate_spring_wrench(thetalist, Slist, stiffness, restLength, ref
        # print(spring_force_vec)
        # Forward dynamics
        i_acc = mr.ForwardDynamics(thetalist, dthetalist, tau_damping, g, spring_force_vec, Mli
        i_pos, i_vel = mr.EulerStep(thetalist, dthetalist, i_acc, dt)
        thetamat[i + 1] = i_pos
        dthetamat[i + 1] = i_vel

        # Update
        thetalist = i_pos
        dthetalist = i_vel

    return thetamat, dthetamat
```

In [9]:
```
q4_thetalist0 = np.array([0, 0, 0, 0, 0, 0])
q4_dthetalist0 = np.array([0, 0, 0, 0, 0, 0])
g_q4 = np.array([0, 0, 0])

q4_thetamat, _ = puppet_q4(q4_thetalist0, q4_dthetalist0, g_q4, ur5.Mlist, ur5.Slist, ur5.Glist
# Save to csv file
# print(q4_thetamat)
np.savetxt('q4_thetamat.csv', q4_thetamat, delimiter=',')
```

```
100%|██████████| 1000/1000 [00:08<00:00, 120.20it/s]
```

Part 4 Response

Run the codes above to generate csv trajectory for part 4.