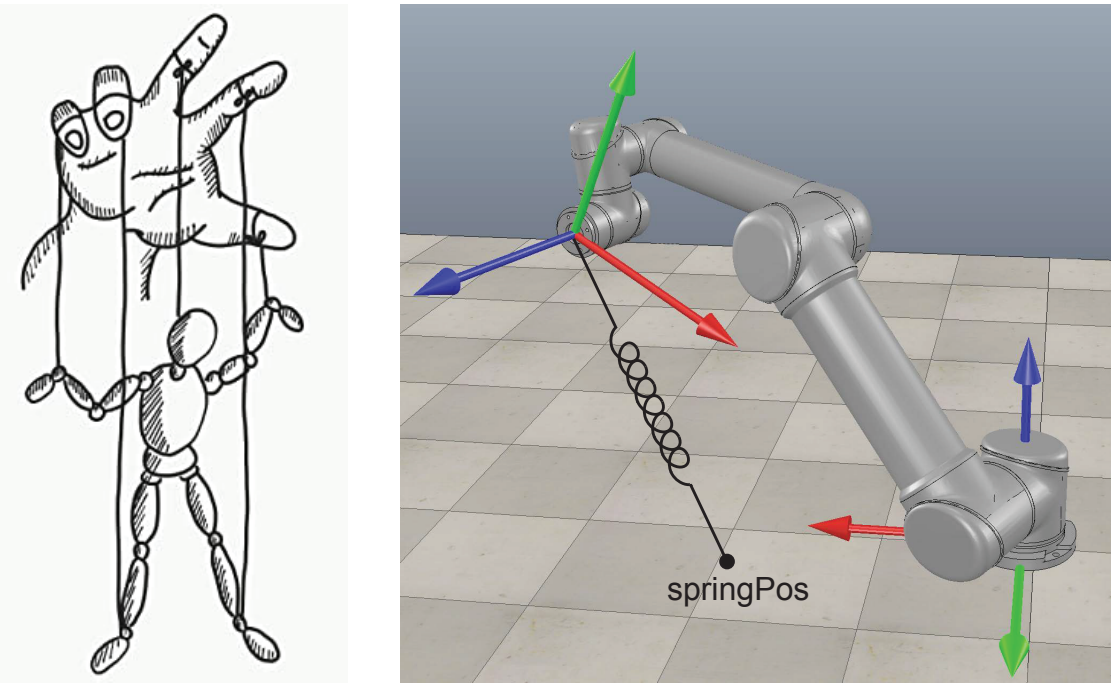
ME 449 Assignment 3
Puppet on a String
Due 1:30 PM, Friday November 15, 2024

You've probably seen animation and physics engine tools that allow you to interactively grab a link of a virtual mechanism and apply forces to it, to make it move like a marionette on a string. In this assignment, you will create the underlying software that allows you to do this for a 6R robot.



Left: A puppet on a string. Right: A robot on a spring.

Create the new function `Puppet`. It takes the following inputs:

- thetalist: an n-vector of initial joint angles (units: rad)
- dthetalist: an n-vector of initial joint rates (units: rad/s)
- g: the gravity 3-vector in the {s} frame (units: m/s^2)
- Mlist: the configurations of the link frames relative to each other at the home configuration. (There are eight frames total: {0} or {s} at the base of the robot, {1} ...{6} at the centers of mass of the links, and {7} or {b} at the end-effector.)
- Slist: the screw axes $\mathcal{S}_i$ in the space frame when the robot is at its home configuration
- Glist: the spatial inertia matrices $\mathcal{G}_i$ of the links (units: kg and kg m^2)
- t: the total simulation time (units: s)
- dt: the simulation timestep (units: s)
- damping: a scalar indicating the viscous damping at each joint (units: Nms/rad)
- stiffness: a scalar indicating the stiffness of the springy string (units: N/m)
- restLength: a scalar indicating the length of the spring when it is at rest (units: m)

As output, `Puppet` produces

- thetamat: an $N \times n$ matrix where row i is the set of joint values after simulation step $i-1$

- dthetmat: an $N \times n$ matrix where row i is the set of joint rates after simulation step $i - 1$

If your total simulation time t is 5 s and dt is 0.01 s, N will equal 500 or 501.

You control the motion of the robot by pulling on a springy string. One end of the spring is at the position springPos, (x, y, z) in the {s} frame, and the other end of the spring is attached to the origin of the {b} frame at the end-effector. If the distance between springPos and {b} is more than restLength, the spring pulls the end-effector toward springPos with a magnitude (stiffness $\times$ (distance $-$ restLength)). If the distance is less than restLength, the spring pushes the end-effector away from springPos.

To control springPos, at each simulation iteration `Puppet` calls a second function you will write, `referencePos`. `referencePos` takes as input the current time and produces as output a three-vector (x, y, z), which is treated as the current springPos in `Puppet`.

At each iteration of the simulation, `Puppet` calls `referencePos` and the MR function `ForwardDynamics` to determine the acceleration of the robot based on springPos and the current state of the robot. Then it calculates the new state of the robot using numerical integration. You can use simple first-order Euler numerical integration, as implemented by `EulerStep`. You are welcome to experiment with other integrators, including adding a $\frac{1}{2}\ddot{\theta}(\mathrm{dt})^2$ term to get a more accurate change in position each timestep, but `EulerStep` is fine for this project.[1]

The robot we will use is the UR5, a popular 6-dof industrial robot arm. The robot has geared motors at each joint, but in this project, we ignore the effects of gearing, such as friction and the increased apparent inertia of the rotor.

The relevant kinematic and inertial parameters of the UR5 are:

$$M_{01} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.089159 \\ 0 & 0 & 0 & 1 \end{bmatrix}, M_{12} = \begin{bmatrix} 0 & 0 & 1 & 0.28 \\ 0 & 1 & 0 & 0.13585 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, M_{23} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -0.1197 \\ 0 & 0 & 1 & 0.395 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$M_{34} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.14225 \\ 0 & 0 & 0 & 1 \end{bmatrix}, M_{45} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0.093 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, M_{56} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.09465 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$M_{67} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.0823 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, G_1 = \mathtt{diag}([0.010267495893, 0.010267495893, 0.00666, 3.7, 3.7, 3.7]),$$

$$G_2 = \mathtt{diag}([0.22689067591, 0.22689067591, 0.0151074, 8.393, 8.393, 8.393]),$$

$$G_3 = \mathtt{diag}([0.049443313556, 0.049443313556, 0.004095, 2.275, 2.275, 2.275]),$$

$$G_4 = \mathtt{diag}([0.111172755531, 0.111172755531, 0.21942, 1.219, 1.219, 1.219]),$$

$$G_5 = \mathtt{diag}([0.111172755531, 0.111172755531, 0.21942, 1.219, 1.219, 1.219]),$$

$$G_6 = \mathtt{diag}([0.0171364731454, 0.0171364731454, 0.033822, 0.1879, 0.1879, 0.1879]),$$

$$\mathtt{Slist} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & -0.089159 & -0.089159 & -0.089159 & -0.10915 & 0.005491 \\ 0 & 0 & 0 & 0 & 0.81725 & 0 \\ 0 & 0 & 0.425 & 0.81725 & 0 & 0.81725 \end{bmatrix}.$$

For your convenience, these parameters are given in Python, Mathematica, and MATLAB at http://hades.mech.northwestern.edu/index.php/Modern_Robotics#Supplemental_Information.

[1] The MR function `ForwardDynamicsTrajectory` also simulates the motion of a robot, so you can look at that if you wish, but it takes different inputs and handles the simulation timesteps differently.

Your code will be tested in stages to help ensure correctness. **Important:** You are welcome to talk to classmates about concepts at the development phase, but you are not allowed to share your code nor look at anyone else's code. AI-based software easily detects shared and altered code with common origins, so do not do it.

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/s^2 in the $-\hat{z}_s$-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.

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?

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.

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.

This simulates a robot control mode called *impedance control*. With impedance control, you can define a moving reference point (or a reference frame) and a virtual spring/damper connecting the reference to the end-effector of the robot. The virtual wrench $\mathcal{F}$ due to the virtual spring/damper is converted to a set of actual joint torques via $\tau = J^{\intercal}\mathcal{F}$ that drive the robot. This control mode can be used to control the motion of a robot in free space or the force a robot applies when in contact with an environment (for example, if the reference is behind a wall, the robot applies a force to the wall that depends on the stiffness of the virtual spring and the distance of the reference behind the wall).

What to turn in: You should turn in a single zip file named FamilyName_GivenName_asst3.zip. It should contain a pdf file FamilyName_GivenName_asst3.pdf; videos named part1a.mp4, part1b.mp4, part2a.mp4, part2b.mp4, part3a.mp4, part3b.mp4, and part4.mp4; corresponding csv files named part1a.csv, etc., that were animated by CoppeliaSim to create your videos; and a directory (folder) called "code." The pdf file should have five sections: Introduction, Part 1, Part 2, Part 3, and Part 4. In the Introduction, provide any explanation needed to understand your overall submission. (This may be very short.) In Parts 1-4 answer all questions posed above and provide any needed information explaining the corresponding videos and csv files, including the dt, t, damping, and stiffness values used for the animations. In the code directory, provide your commented function(s) and sample code that we can execute to generate the csv files you turned in. **Do not submit data or videos for cases where you were simply experimenting to better understand the system behavior; submit data and videos only for parts 1(a), 1(b), 2(a), 2(b), 3(a), 3(b), and 4.**