Homework #6
due Wednesday 11/8/22 11:59pm

This week we are digging a little deeper into the Jacobian inverse and commanding rotations and full 6 DOF robots.

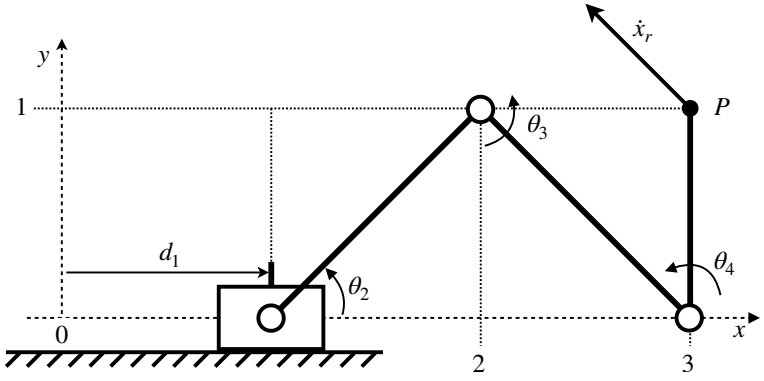
We have again posted a code package `hw6code` on Canvas. The ROS code format should now be pretty familiar: beyond the administrative/initialization code, you will always need the `evaluate()` function to compute (1) a desired trajectory and (2) an inverse Jacobian step. With rotations, transforms, and Jacobians playing such a big role, be sure to check the `TrajectoryUtils.py`, `TransformHelpers.py`, and the NumPy notes attached to HW#5. I've also added launch files, so you avoid starting the three parts separately. Take a look at these, then just run

```
ros2 launch hw6code hw6pX.launch.py
```

As always, please submit the graphs/plots as well as the relevant code for simulations (the `Trajectory` class), together with any other requested information.

Problem 1 (Scaled Joint Velocities - from Last Year's Quiz 2) - 18 points:

Consider the planar 4 DOF robot in the following configuration



The first joint slides horizontally. The second and third links are $\sqrt{2}$m long, the last is 1m long. You are considering only the tip position P in x/y, not its orientation. Your task is to solve the inverse kinematics, that is to

$$\text{find} \quad \dot{\vec{q}} = \begin{bmatrix} \dot{d}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix} \quad \text{that creates an} \quad \dot{\vec{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = J(q)\dot{\vec{q}} \quad \text{to match a known reference} \quad \dot{\vec{x}}_r$$

(a) What is the numerical value of the Jacobian J of this system, in the above configuration?

You will notice that the Jacobian mixes a prismatic and three revolute joints, that is it mixes units. Rather than allowing the choice of units to determine the robot's ultimate behavior, you decide to

consider each joint's top speed. You know the prismatic joint tops out at 1 m/s, the second joint at 2 rad/sec, the third 3 rad/sec, and the fourth 4 rad/sec. So you define the relative joint velocities

$$\nu_i = \frac{\dot{q}_i}{\dot{q}_i{}^{\max}} \qquad \text{or} \qquad \vec{\nu} = W\dot{\vec{q}} \quad \text{with} \quad W = \text{diag}\left(\frac{1}{\dot{q}_i{}^{\max}}\right)$$

In the following, minimize the norm of the relative joint velocities $\|\vec{\nu}\|$, assume the robot is in the shown configuration, and the reference tip velocity is

$$\dot{\vec{x}}_r = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

(b) What is the formula for the best $\dot{\vec{q}}$ given J, W, and $\dot{\vec{x}}_r$? If convenient, define $M = W^{-2}$.

(c) For the given maximum joint velocities, what is the best $\dot{\vec{q}}$?

(d) Further, assume joint 2 locks up, that is $\dot{\theta}_2{}^{\max} = 0$ rad/sec. Under this condition, what is the best $\dot{\vec{q}}$?

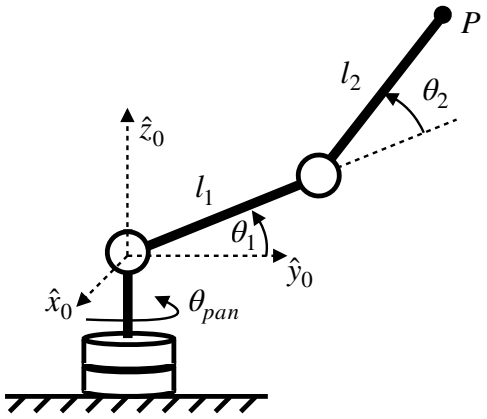
(e) Finally, assume *both* joints 2 and 3 lock up. Under this condition, what is the best $\dot{\vec{q}}$?

Please show your work, including any code you used for matrix computations.

Problem 2 (Jacobian Inversion near Singularities) - 18 points:

Before we get into the larger robots and serious coding, I'd like to examine the Jacobian inversion near singularities. This may also remind you of Homework #2, Problem 5, where we examined the joint velocities near the singularity for a 2 DOF.

Here, consider the classic 3 DOF, for which we have analytic expressions and you have implemented the analytic `fkin()` and `Jac(q)` functions in Homework #5, Problem 1. You may compute the following in Matlab or Python, as you prefer.



As before, with $\ell_1 = \ell_2 = 1$m, and for the nearly singular joint configuration $q = \begin{bmatrix} 0^\circ \\ 44.5^\circ \\ 90^\circ \end{bmatrix}$ we are going to find $\dot{q}$ so the achieved tip velocity $\dot{x} = J(q)\dot{q}$ matches $\dot{x}_r = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$.

If you examine q, you will notice that it places the robot's tip near the vertical axis, making the pan axis nearly useless. So it may be unrealistic to expect a perfect match/solution. Instead we'll explore the various inversion formulas to see how close a match they provide. Also note with the link lengths of 1m, a tip velocity of 1m/s should (ideally, in the sweat spot) produce joints velocities *on the order* of 1rad/s.

(a) Compute the Jacobian matrix $J(q)$ numerically. Also compute the singular value decomposition

$$J = U \ \text{diag}(s_i) \ V^T$$

(b) Use a regular Jacobian inverse J^{-1} to compute the joint velocities

$$\dot{q} = J^{-1}\dot{x}_r \qquad \text{or} \qquad \dot{q} = V \ \text{diag}\left(\frac{1}{s_i}\right) \ U^T \ \dot{x}_r$$

What is the achieved $\dot{x}$? Anything bad happening?

(c) Replace the regular inverse with a weighted inverse

$$J_{\text{Winv}} = (J^T J + \gamma^2 I_N)^{-1} J^T = J^T (J J^T + \gamma^2 I_M)^{-1}$$

and compute

$$\dot{q} = J_{\text{Winv}} \ \dot{x}_r \qquad \text{or} \qquad \dot{q} = V \ \text{diag}\left(\frac{s_i}{s_i^2 + \gamma^2}\right) \ U^T \ \dot{x}_r$$

with a $\gamma = 0.01$ (allowing yourself approximately a distance of 1cm to the singularity). What happens to the joint velocities $\dot{q}$ and the achieved tip velocity $\dot{x}$?

(d) Repeat the weighted inversion with $\gamma = 0.1$. What are $\dot{q}$ and $\dot{x}$? What changed?

(e) As the last option, compute an SVD-based inverse

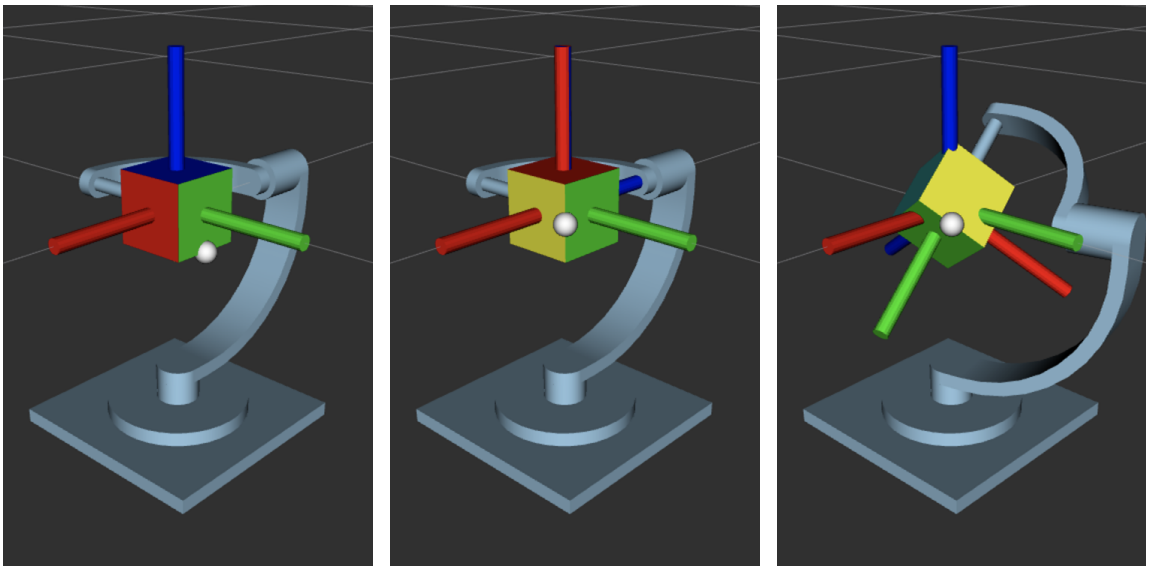
$$\dot{q} = V \ \text{diag}\left(s_i^{\text{inv}}\right) \ U^T \ \dot{x}_r \qquad \text{with} \qquad s_i^{inv} = \begin{cases} \dfrac{1}{s_i} & \text{if} \quad |s_i| \geq \gamma \\ \dfrac{s_i}{\gamma^2} & \text{if} \quad |s_i| < \gamma \end{cases}$$

again with $\gamma = 0.1$. Report $\dot{q}$ and $\dot{x}$. How large are the joint velocities and how well does the tip velocity match the desired? How does this compare to the above?

Problem 3 (Gimbal Rotational Motion) - 20 points:

To start using the general kinematics you code in Homework #5, Problem 5, we are going to rotate a pan-tilt-roll gimbal through various orientations. In particular, please find the `gimbal.urdf` in the `hw6code` package. You may want to use RVIZ and the joint-slider GUI to move and get a sense of the robot.

In particular, notice ROS shows the x axis in red, the y axis in green, and the z axis in blue (RGB). The cube is similarly color-coded. Hopefully this will make it slightly easier to observe the orientations.



The final movement will have two parts. First moving from the starting (zero) orientation R_0 shown on the left, via a rotation about the y axis, to the orientation R_A shown in the middle. This will bring the white ball to the front (technically at coordinates x=0.05, y=0.05, z=0).

Second, rotating the cube in such a way as not to move the white ball. That is, keeping the white ball at the above coordinates while the cube is turning, as shown on the right. The axis of rotation will hence pass from the origin through the white ball. We'll keep this rotation going indefinitely at a constant speed.

Through the various transitions, we want to keep the velocity continuous. To solve all this, we will again build the code up in steps:

(a) Consider only the first phase with a starting and final orientation of

$$R_0 = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad R_A = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \text{Rot}_y(-90^\circ)$$

covering a period of 2s.

Indeed consider this a rotation about a fixed axis by an angle α. Let us build up

$$R_d(t) = R_d(\alpha) \quad \text{and} \quad \omega_d(t) = \omega_d(\alpha, \dot\alpha) \qquad \text{for an appropriate} \qquad \alpha(t) \quad \text{and} \quad \dot\alpha(t)$$

- First, what are the initial angle α_0 and final angle α_f?
- Using a cubic spline, what is the formula for $\alpha(t)$ if the angle starts at the α_0 with zero velocity and, after 2s, ends at the final value α_f with zero velocity. What is the formula for the derivative $\dot{\alpha}(t)$?
- What are the expressions for $R_d(\alpha)$ and $\omega_d(\alpha, \dot{\alpha})$?

Please submit these expressions, then code into `hw6p3.py`.

- Also program the (rotational) inverse kinematics

$$\dot{q}(t) = J(q)^{-1} \left[\dot{x}_d(t) + \lambda e(t - dt)\right]$$

similar to last week, but now using the general `KinematicChain`:

```
from hw5code.KinematicChain    import KinematicChain

(ptip, Rtip, Jv, Jw) = chain.fkin(q)
```

and applying the rotational parts:

$$\dot{q}(t) = J_w(q)^{-1} \left[\omega_d(t) + \lambda e_R(R_d, R)\right]$$

Check `TransformHelpers.py` for the error function. And you can use the `goto()` function to compute $\alpha(t), \dot{\alpha}(t)$. Finally, as before, from $\dot{q}(t)$ integrate to get $q(t)$.

Together this means the cube should smoothly rotate (over 2s) from R_0 to R_A and stop.

(b) For the second part, keep α at the final value α_f. Introduce a new angle β which was zero during the first two seconds and then evolves as

$$\beta(t) = \begin{cases} 0 & \text{if } t \leq 2 \\ t - 3 + e^{2-t} & \text{if } t > 2 \end{cases} \qquad \dot{\beta}(t) = \begin{cases} 0 & \text{if } t \leq 2 \\ 1 - e^{2-t} & \text{if } t > 2 \end{cases}$$

- Before proceeding, what is the axis of rotation during this second phase? Remember the axis is a unit vector and we don't want to move the white ball. Careful whether you are expressing this axis in the tip or world frame!
- Pulling this together, what are the formulas for $R_d(t)$ and $\omega_d(t)$ after $t = 2$?

As before, please code into `hw6p3.py`. Note the formulas for $R_d(t)$ and $\omega_d(t)$ can be distinct from part (a), needing an "if (t>2)" statement in code, or can be combined with the previous into one formula that holds over all time.

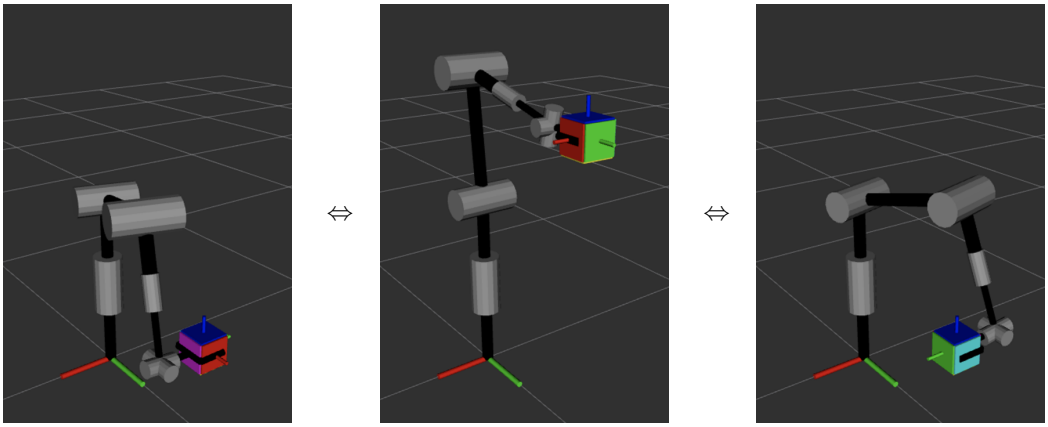
This inverse kinematics should remain the same, as the only change in this second phase is the desired orientation/motion. That means, we should now have a spinning cube!

Please make sure the motion rotates smoothly and indeed keeps the white ball in a fixed location in space after $t = 2$. Submit the final code. And record and submit a graph of the first ∼15s, which should be about 2 revolutions of the cube after the initial turn.

Problem 4 (6 DOF Inverse Kinematics - Up-Down Movements) - 20 points:

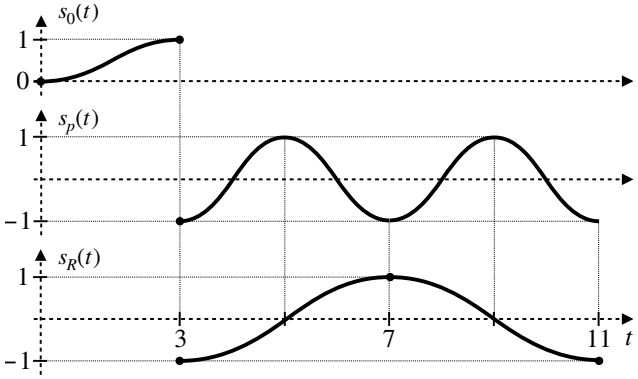
Time to shift to a full 6 DOF robot. This uses the URDF file `sixDOF.urdf` included in the `hw6code` package. You may want to use RVIZ and the joint-slider GUI to move the robot around. Note the "shoulder" is 0.6m off the ground, the "upper arm" and "lower arm" are 0.4m long, and the "wrist" to "tip" distance is 0.15m. The cube is centered around the tip (the center of cube is the tip), measuring 0.1m on each side.

We are looking for a motion that both rotates and raises the cube up and down through the configurations



The motion starts at a known $(p_0, R_0) = \text{fkin}(q_0)$, then cycles through $(p_{\text{low}}, R_{\text{right}})$, $(p_{\text{high}}, R_{\text{mid}})$, $(p_{\text{low}}, R_{\text{left}})$, $(p_{\text{high}}, R_{\text{mid}})$, and ever repeating. It uses inverse kinematics to determine all joint values after the initialization.

To simplify the code, similar to Problem 3, we use the path variable s_0 during the initial phase, and s_p and s_R during the cyclic phase. We can compute these with splines segments or as sinusoids.



This allows us to compute the desired position, velocity, orientation, and angular velocity as follows. For $t < 3$:

$$p_d(t) = p_0 + (p_{\text{low}} - p_0)s_0 \qquad R_d(t) = \text{Rot}_z(-\frac{\pi}{2}s_0)$$

$$v_d(t) = \quad (p_{\text{low}} - p_0)\dot{s}_0 \qquad \omega_d(t) = \; -e_z \quad \frac{\pi}{2}\dot{s}_0$$

For $t \geq 3$, with $t_1 = (t-3)$, implicitly repeating every 8s:

$$p_d(t) = \frac{1}{2}\,(p_{\text{high}} + p_{\text{low}}) + \frac{1}{2}\,(p_{\text{high}} - p_{\text{low}})s_p \qquad R_d(t) = \text{Rot}_z(\frac{\pi}{2}s_R)$$

$$v_d(t) = \frac{1}{2}\,(p_{\text{high}} - p_{\text{low}})\dot{s}_p \qquad \omega_d(t) = e_z\,\frac{\pi}{2}\dot{s}_R$$

For this problem, we provide the desired trajectory, both to offer another example, as well as to allow you to focus on the inverse kinematics code.

Please update the code to initialize and implement the inverse kinematics. This will combine the position and orientation elements from past problems. If you are unfamiliar with NumPy, you will likely want to review the commands

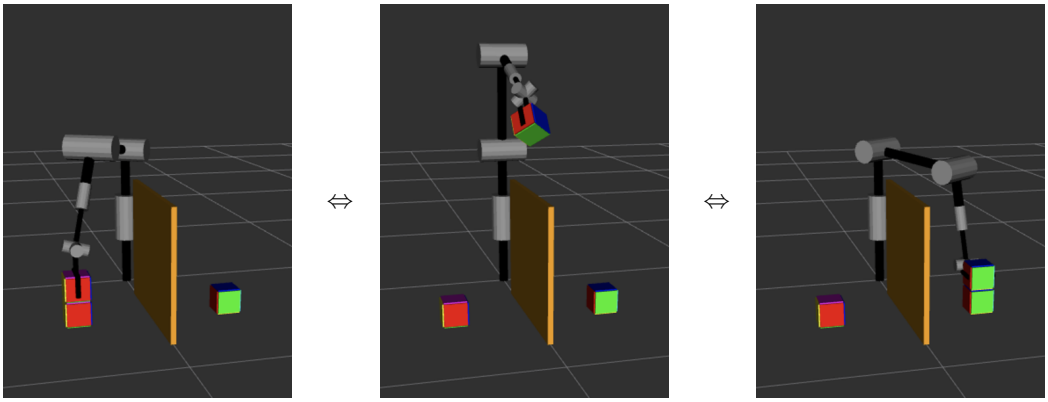
```
vec6x1   = np.vstack((top3x1, bottom3x1))    to vertically stack vectors
mat6x6   = np.vstack((top3x6, bottom3x6))    to vertically stack matrices
Jinv     = np.linalg.inv(J)                  to invert a matrix
Jpinv    = np.linalg.pinv(J)                 to compute a pseudo-inverse
u, s, vT = np.linalg.svd()                   to compute an SVD decomposition
n        = np.linalg.norm(vec)               to compute a vector norm
```

Please submit a plot of the joint movements (recorded in a ROS bag) for the first 11s as well as the relevant code (Trajectory Class).

Problem 5 (Touch and Go Movements) - 20 points:

Now let's make the motion for the 6 DOF robot a little more interesting. This will reuse the inverse kinematics code from the last problem. But trajectory will be new!

Use/load `sixDOF_Obstacle.urdf` in place of `sixDOF.urdf`. Again you may want to load into RVIZ (perhaps with the GUI) to take a look. This is the same mechanism, but with two targets and a central wall obstacle.



Both targets are also cubes, placed on the $z = 0$ plane. With all cubes being 10cm in size and the tip being in the middle of the moving cube, the target height is 15cm. More precisely, the two target

positions are:

$$p_{\text{left}} = \begin{bmatrix} 0.3 \\ 0.5 \\ 0.15 \end{bmatrix} \qquad \text{and} \qquad p_{\text{right}} = \begin{bmatrix} -0.3 \\ 0.5 \\ 0.15 \end{bmatrix}$$

As the robot "touches" either target, it should hold the cube in an orientation that matches the target. Recall that RVIZ and the cubes are color-coded, such that red (arrow or cube face) points in the positive x direction, green in y, and blue in z. The central wall is 0.5m high above the y axis at $x = 0$.

For the trajectory, start at the known (non-singular!) location

$$q_0 = \begin{bmatrix} 0^\circ \\ 90^\circ \\ -90^\circ \\ 0^\circ \\ 0^\circ \\ 0^\circ \end{bmatrix} \qquad p_0 = \text{fkin}(q_0) = \begin{bmatrix} 0.0 \\ 0.55 \\ 1.0 \end{bmatrix} \qquad R_0 = \text{fkin}(q_0) = I = \text{Reye}()$$

(knowing the joint and task coordinates), then

(i) over 3s, moving to the target,

(ii) thereafter move back and forth between the two targets, crossing over the wall. Each back and forth cycle should complete in 5s.

The only hard requirement is that the trajectory be (C^1) smooth, that is continuous in velocity. You may construct the back-and-forth movement from a single move or use however many segments or pieces you believe appropriate. You may utilize the `TrajectoryUtils.py` functions, sinusoidal movements, or any other functions. This may or may not include intermediate path variables. Indeed, I could imagine moving up, then over, then down. Or a more circular or parabolic shape. Think of what might be easiest to implement?

(a) What are the matching orientations R_{left} and R_{right}, expressed in world coordinates?

(b) What trajectory did you use *between the two targets*?

Please give the expressions for the full trajectory, including the position $\vec{p}(t)$, orientation $R(t)$, translational velocity $\vec{v}(t)$, and angular velocity $\vec{\omega}(t)$.

If you express these via a path variable s, in general as $\vec{p}(s)$, $R(s)$, $\vec{v}(s, \dot{s})$, $\vec{\omega}(s, \dot{s})$, please also give specify how the path variable evolves over time as $s(t), \dot{s}(t)$.

(c) Please also submit plots of the joint movement data (from 0s to 8s) and the relevant code.

Problem 6 (Time Spent) - 4 points:

Once again, may we know how much time you spent? In particular on P1-2 vs P3-5? Did you encounter any particular difficulties?