Homework #5
due Wednesday 11/1/22 11:59pm
extensions provided if the delay caused issues

This week we leverage the kinematic chains, Jacobians, and splines to solve inverse kinematics and generate full robot movements in general.

This will involve more coding to build up the matrices and will force us to use Python's NumPy module. Please look at the attached NumPy notes (and the end of this document) and the provided examples, and certainly ask for help as needed! The sample code is collected in the `hw5code` package on Canvas.

As always, please submit the relevant code as well as the graphs/plots.

3 DOF Kinematics:

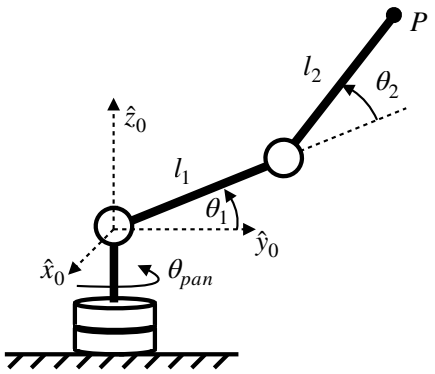
For the following problems, we will use the 3 DOF robot introduced in HW#2 with the URDF `threeDOF.urdf`. Assume $\ell_1 = \ell_2 = 1$m for all problems.

We will consider the tip position as the task or output coordinates. Meaning

$$x = \vec{p} = \begin{bmatrix} x_{\text{tip}} \\ y_{\text{tip}} \\ z_{\text{tip}} \end{bmatrix} \qquad q = \begin{bmatrix} \theta_{\text{pan}} \\ \theta_1 \\ \theta_2 \end{bmatrix}$$

leading to the 3x3 Jacobian

$$\dot{x} = \begin{bmatrix} \dot{x}_{\text{tip}} \\ \dot{y}_{\text{tip}} \\ \dot{z}_{\text{tip}} \end{bmatrix} = J(q)\ \dot{q} = J(q) \begin{bmatrix} \dot{\theta}_{\text{pan}} \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix}$$



From HW#2 (check the solutions) we know the analytic expression for the forward kinematics mapping and the Jacobian matrix (for $\ell_1 = \ell_2 = 1$m and writing $\theta_{12} = \theta_1 + \theta_2$):

$$x(q) = \begin{bmatrix} -\sin(\theta_{\text{pan}})\ (\cos(\theta_1) + \cos(\theta_{12})) \\ \cos(\theta_{\text{pan}})\ (\cos(\theta_1) + \cos(\theta_{12})) \\ (\sin(\theta_1)\ + \sin(\theta_{12})) \end{bmatrix}$$

$$J(q) = \begin{bmatrix} -\cos(\theta_{\text{pan}})\ (\cos(\theta_1) + \cos(\theta_{12})) & \sin(\theta_{\text{pan}})\ (\sin(\theta_1)\ + \sin(\theta_{12})) & \sin(\theta_{\text{pan}})\ \sin(\theta_{12}) \\ -\sin(\theta_{\text{pan}})\ (\cos(\theta_1) + \cos(\theta_{12})) & -\cos(\theta_{\text{pan}})\ (\sin(\theta_1)\ + \sin(\theta_{12})) & -\cos(\theta_{\text{pan}})\ \sin(\theta_{12}) \\ 0 & \cos(\theta_1) + \cos(\theta_{12}) & \cos(\theta_{12}) \end{bmatrix}$$

We also know we have **2 multiplicities**, leading to $\mathbf{4 = 2^2}$ **solution sets**

Elbow (θ_2) multiplicity:	
Elbow up	$\sin(\theta_2) < 0$
Elbow down	$\sin(\theta_2) > 0$

Front/Back (θ_{pan}) multiplicity:	
Front side	$\cos(\theta_1) + \cos(\theta_{12}) > 0$
Back side	$\cos(\theta_1) + \cos(\theta_{12}) < 0$

Problem 1 (3 DOF Jacobian - NumPy Warm-up) - 10 points:

Please write two python helper functions `fkin(q)` and `Jac(q)`, returning the above 3 DOF expressions as a NumPy 3x1 and 3x3 matrices. Note NumPy will try to flatten a 3x1 vector in a 3 element array, so make sure it is a column vector. See the skeleton code `hw5p1.py` in the posted package.

Please confirm that you satisfy the following test case:

$$q = \begin{bmatrix} 20^\circ \\ 40^\circ \\ -30^\circ \end{bmatrix} = \begin{bmatrix} 0.3491 \\ 0.6981 \\ -0.5236 \end{bmatrix} \quad x(q) = \begin{bmatrix} -0.5988 \\ 1.6453 \\ 0.8164 \end{bmatrix} \quad J(q) = \begin{bmatrix} -1.6453 & 0.2792 & 0.0594 \\ -0.5988 & -0.7672 & -0.1632 \\ 0.0000 & 1.7509 & 0.9848 \end{bmatrix}$$

Please submit the isolated `fkin()` and `Jac()` code, as well as the outputs for the second test case:

$$\theta_{\text{pan}} = 30^\circ \qquad \theta_1 = 30^\circ \qquad \theta_2 = 60^\circ$$

Problem 2 (Newton Raphson Algorithm) - 24 points:

With the above forward kinematics and Jacobian code, let's continue to use the 3 DOF robot to explore the Newton Raphson iterative algorithm

$$q(i+1) = q(i) + J(q(i))^{-1} \left(x_{\text{goal}} - x(q(i))\right)$$

Looking only at translation and having a 3 DOF robot, the 3x3 Jacobian will allow inversion. Program the Newton-Raphson algorithm. For each step, store both the distance $\|x_{\text{goal}} - x(q(i))\|$ and the resultant joint step size $\|q(i+1) - q(i)\|$. Plot these two versus the iteration count, for up to 20 iterations. Practically, work from the skeleton code `hw5p2.py` and see the NumPy notes at the end of this set.

For the initial guess, use something reasonably in the middle of the workspace, namely

$$q(0) = \begin{bmatrix} 0 \\ \pi/2 \\ -\pi/2 \end{bmatrix}$$

This has the "elbow up" and locates the output on the "front side" (with pan "facing the tip"). Execute the algorithm for the following seven different tip position targets

$$x_{\text{goal}} = \begin{bmatrix} 0.5 \\ 1.0 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 1.0 \\ 0.5 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 2.0 \\ 0.5 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.0 \\ -1.0 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.0 \\ -0.6 \\ 0.5 \end{bmatrix}, \begin{bmatrix} 0.5 \\ -1.0 \\ 0.5 \end{bmatrix}, \begin{bmatrix} -1.0 \\ 0.0 \\ 0.5 \end{bmatrix}$$

so that we may see different good and bad ways the algorithm converges (or doesn't). Indeed, in some cases the algorithm oscillates, switches solution branches (multiplicity), wraps by 360°, etc.

Create a table and for each of the seven targets, please report:

(a) Did the algorithm converge? If so, then:
(b) How many steps were required to reduce the distance $\|x_{\text{goal}} - x(q(i))\|$ below 10^{-12}?
(c) What were the final joint values q?
(d) Does this correspond to an elbow up or down, front or back side solution?
(e) How many times did each joint wrap by 360°?

Also please submit the relevant code and the last plot (target #7).

Problem 3 (3 DOF Joint Movement - NumPy/Spline/ROS Warm-up) - 14 points:

From here on, we focus on smooth movements over time. Consider the three joint positions

$$q_A = \begin{bmatrix} 0^\circ \\ 60^\circ \\ -120^\circ \end{bmatrix} \qquad q_B = \begin{bmatrix} -90^\circ \\ 135^\circ \\ -90^\circ \end{bmatrix} \qquad q_C = \begin{bmatrix} -180^\circ \\ 60^\circ \\ -120^\circ \end{bmatrix}$$

Review the skeleton code `hw5p3.py` which cyclically moves between q_A and q_B, using 2s for each leg. Please update the code to

(a) Use three legs: q_A to q_B, q_B to q_C, q_C to q_A, each taking 2s.

(b) Notice as the robot moves from q_A to q_B to q_C, the first (pan) angle changes from 0° to -90° to -180°. But rather than continuing at a smooth velocity, it comes to rest at q_B. Please update the two splines (boundary conditions) so the pan angle does not come to rest. What velocity should you impose at q_B, so that the velocity is smooth (no jumps in acceleration)? **Hint:** Imagine a single 4s spline from 0° to -180°. What velocity would it have midway? You can solve this analytically, or just call the `goto()` function to compute it numerically.

Please store the motion using a ROS bag and plot via Matlab. Submit the relevant code (`Trajectory` class), the plot for a 6second cycle the final case, and the pan velocity at q_B in (b).

Problem 4 (3 DOF Tip Movement - Inverse Jacobian) - 24 points:

Let us start moving in task space, and continue to plot the resulting motions. Again using the 3 DOF robot, with the kinematics code from Problem 1.

Your task is to move the robot between two points A and D. Point A is defined in joint space via q_A, so we can immediately determine the matching x_A. Point D is defined only in task space via x_D. Assume the matching joint values q_D are not known.

$$q_A = \begin{bmatrix} 0^\circ \\ 60^\circ \\ -120^\circ \end{bmatrix} \quad x_A = \begin{bmatrix} 0.0 \\ 1.0 \\ 0.0 \end{bmatrix} \qquad q_D = \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} \quad x_D = \begin{bmatrix} 0.5 \\ -0.5 \\ 1.0 \end{bmatrix}$$

The first leg (from A to D) will move the tip is a straight Cartesian line in 3D space. The second, return leg (from D to A) will move directly in joint space, irrespective of or ignoring the path of the tip. Use 3s for each leg and come to rest (zero velocity) at both points. Upon completion, start all over again and repeat indefinitely.

The challenges are:

(i) For the first leg, use the inverse kinematics

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

where $\lambda = 20$ should be a reasonable value (in 10-100 range). Numerically integrate $\dot{q}(t)$ every cycle to obtain $q(t)$ (this is where the `dt` argument in the `evaluate()` function will be useful).

(ii) For the second leg, we do not know and can't uniquely pre-compute the joint coordinates of D (without knowing which solution to select). So let the inverse kinematics in the first leg figure this out for you.

Three (hopefully useful) notes:

- Please see the skeleton code `hw5p4.py`.
- Keep in mind what you need to store between cycles (time steps). Anything you store and access in `evaluate()`, must be initialized (in `__init__()`).
- If you return `None` from `evaluate()` after 6 seconds (in place of `(q, qdot)`), the code will shut down and your plots will show exactly 6 seconds of data.

Please submit

(a) a screenshot of the robot at points A and D,
(b) the plot for one A to D to A cycle (6s), recorded via ROS bag as before,
(c) and your code (the `Trajectory` class).

Problem 5 (General Kinematic Chain Calculations) - 24 points:

Looking ahead, we need to be able to handle arbitrary URDFs. Please write (update/fix) the code to process a URDF into the kinematic chain, and ultimately output the position/orientation/Jacobian of the tip for specific joint value.

Specifically, please edit `KinematicChain.py` in the `hw5code` package. As you might expect, you'll see:

(i) The class `KinematicStep`. Please take a look - this stores the information of each step in the chain. In particular it contains

  (a) Permanent information, obtained from the URDF. This is "local", giving only the information how the frame relates to the previous frame, etc.

  (b) Transient information, computed as we walk up the chain. This store the frame's information with respect to the world frame and is recomputed every `fkin()` call.

(ii) The class `KinematicChain` with the initialization `__init__()`. This spends some time receiving the URDF from ROS (via the `robot_state_publisher` that we always run). From the XML string it extracts a list of steps that connect the base frame to the tip frame. Feel free to examine, but you shouldn't have to edit anything here.

(iii) In `KinematicChain`, the function `fkin(q)`

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

which is the core function "walking up the chain". As it traverses the chain, it needs to recompute all the transforms w.r.t. world for the current (given) set of joint angles (positions). It thus arrives at the tip's position and orientation. A second walk up the chain further populates the Jacobian matrix (split into translational and rotational parts).

You will find that some of this code is incomplete. The challenge of this problems is to complete `fkin()`, making it functional!

In general, please fix anything marked with `FIXME`. Also take a look at the `TransformHelpers.py` functions. And everything is done using NumPy arrays, so please check the attached notes.

In particular, you may want to use:

`T = TA @ TB`	multiply two T matrices
`T_from_Rp(R, p)`	Create T from R and p
`Rote(e, theta)`	Create a rotation about axis e
`Reye()`	Create an identity rotation
`(e * dist)`	Position along axis e, distance d
`pzero()`	Zero position vector
`(pA - pB)`	Difference between two positions
`cross(e, p)`	Cross product of e with p

Run this *under ROS* and remember to run the `robot_state_publisher` using the `threeDOF.urdf` so it gets the URDF information. To make sure the code is working correctly, we've pre-computed the following test cases. Please confirm they work, but only submit the code (being the `fkin()` function).

(a)
$$q = \begin{bmatrix} 20^\circ \\ 40^\circ \\ -30^\circ \end{bmatrix} \quad p_{\text{tip}} = \begin{bmatrix} -0.599 \\ 1.645 \\ 0.816 \end{bmatrix} \quad J_v = \begin{bmatrix} -1.645 & 0.279 & 0.059 \\ -0.599 & -0.767 & -0.163 \\ 0.000 & 1.751 & 0.985 \end{bmatrix}$$

$$R_{\text{tip}} = \begin{bmatrix} 0.940 & -0.337 & 0.059 \\ 0.342 & 0.925 & -0.163 \\ 0.000 & 0.174 & 0.985 \end{bmatrix} \quad J_\omega = \begin{bmatrix} 0.000 & 0.940 & 0.940 \\ 0.000 & 0.342 & 0.342 \\ 1.000 & 0.000 & 0.000 \end{bmatrix}$$

(b)
$$q = \begin{bmatrix} 30^\circ \\ 30^\circ \\ 60^\circ \end{bmatrix} \quad p_{\text{tip}} = \begin{bmatrix} -0.433 \\ 0.750 \\ 1.500 \end{bmatrix} \quad J_v = \begin{bmatrix} -0.750 & 0.750 & 0.500 \\ -0.433 & -1.299 & -0.866 \\ 0.000 & 0.866 & 0.000 \end{bmatrix}$$

$$R_{\text{tip}} = \begin{bmatrix} 0.866 & 0.000 & 0.500 \\ 0.500 & 0.000 & -0.866 \\ 0.000 & 1.000 & 0.000 \end{bmatrix} \quad J_\omega = \begin{bmatrix} 0.000 & 0.866 & 0.866 \\ 0.000 & 0.500 & 0.500 \\ 1.000 & 0.000 & 0.000 \end{bmatrix}$$

(c)
$$q = \begin{bmatrix} -45^\circ \\ 75^\circ \\ 120^\circ \end{bmatrix} \quad p_{\text{tip}} = \begin{bmatrix} -0.500 \\ -0.500 \\ 0.707 \end{bmatrix} \quad J_v = \begin{bmatrix} 0.500 & -0.500 & 0.183 \\ -0.500 & -0.500 & 0.183 \\ 0.000 & -0.707 & -0.966 \end{bmatrix}$$

$$R_{\text{tip}} = \begin{bmatrix} 0.707 & -0.683 & 0.183 \\ -0.707 & -0.683 & 0.183 \\ 0.000 & -0.259 & -0.966 \end{bmatrix} \quad J_\omega = \begin{bmatrix} 0.000 & 0.707 & 0.707 \\ 0.000 & -0.707 & -0.707 \\ 1.000 & 0.000 & 0.000 \end{bmatrix}$$

This, unfortunately, can not test the prismatic joints. But hopefully it will help catch most bugs.

Problem 6 (Time Spent) - 4 points:

Approximately how much time did you spend on this homework? What was the biggest bottleneck? These problems are definitely requiring more coding...

Python / NumPy Notes

I realize the Python learning curve can be steep. Following are a few thoughts on NumPy that will hopefully help avoid problems, especially if you are used to matrix operations in Matlab. See also the `NumPyExamples.py` and `TransformHelpers.py` files.

(a) NumPy arrays provide the mathematical tool for matrices and vectors. And are distinct from generic python lists!

(b) Be careful with the size of NumPy arrays! NumPy operations generally compute some answer, handling operands of various dimensions. But the results can be unexpected if the dimensions do not match your thinking.

In particular, we always want 2D arrays! A column vector should be Nx1, a row vector 1xN. But NumPy defaults to 1D vectors of length N. So my recommended (explicit) option is *to always reshape* into a 2D column when creating/initializing vectors:

```
x     = np.array([1, 2, 3]).reshape((3,1))
theta = np.array([1, 2, 3, 4]).reshape((-1,1))
```

where the "-1" means whatever height is needed.

(c) Consistent with Python, but contrary to Matlab, everything is zero-indexed. But when you extract an element from a 2D column vector, you need to specify two indices. So the second angle θ_2 in the above vector is the second row and first column:

```
theta2 = theta[1,0]
```

(d) When pulling out sub-vectors from matrices, NumPy tries to flatten and create 1D objects. You have to make sure that both indices remain a range. So:

`R[:,1]`	is the 2nd column, but encoded as a 1D vector! Bad!
`R[:,1:2]`	is the 2nd column, as a 2D 3x1 vector! Good!

(e) Also be careful when you are operating on a single θ (walking up the kinematic chain, computing inside the fkin() function) versus when you are operating on the $\vec{\theta}$ vector (calling fkin() and multiplying with or working with Jacobian matrices).

(f) NumPy also defaults to matrix element-by-element products, rather than the normal matrix products we want. So *always* use the "@" multiplication (or the np.matmul() function):

```
xdot = J @ qdot
```

(g) Other useful NumPy functions:

```
vec6x1   = np.vstack((top3x1, bottom3x1))      to vertically stack vectors
Jinv     = np.linalg.inv(J)
Jpinv    = np.linalg.pinv(J)                   pseudo-inverse
u, s, vT = np.linalg.svd()
n        = np.linalg.norm(vec)
```