CS449/549: Learning for Robotics
Bilkent University
Fall 2024
Due date: October 20

Homework 1

Submission

Please follow this instruction to submit your work.

- The coding will be done using the RAI (robotics) package introduced in the tutorial.
- In the necessary parts, upload the ".g" files that you created or updated along with the Python files (.py and .ipynb can be used).
- For the written answers, you should upload a PDF file containing your solutions.
- You should submit a single ".zip" file, which involves all of the files in the working form. **Please ensure that all the files run smoothly without any need for modification when executed.**
- For your ."zip" files, please use this name convention "StudentNumber_HW1.zip"

1 Frames

In this question, you should calculate the pose of objects **by hand** according to the operations below, and additionally, you should create environments in the code. Show and explain your calculations step by step clearly. As a brief review, we have covered two rules of spatial transformation in class.

$$ {}^{A}X^{B}\,{}^{B}X^{C} = {}^{A}X^{C} \tag{1} $$

$$ [{}^{A}X^{B}]^{-1} = {}^{B}X^{A} \tag{2} $$

Note that the rules of transforms are based on the rules of transforming positions and rotations that are listed below.

- Addition of positions in the same frame

$$ {}^{A}p_{F}^{B} + {}^{B}p_{F}^{C} = {}^{A}p_{F}^{C} \tag{3} $$

- The additive inverse

$$ {}^{A}p_{F}^{B} = -{}^{B}p_{F}^{A} \tag{4} $$

- Rotation of a point

$$ {}^{A}p_{G}^{B} = {}^{G}R^{F}\,{}^{A}p_{F}^{B} \tag{5} $$

- Chaining rotations
$$ {}^{A}R^{B}\,{}^{B}R^{C} = {}^{A}R^{C} \tag{6}$$

- Inverse of rotations
$$ [{}^{A}R^{B}]^{-1} = {}^{B}R^{A} \tag{7}$$

Applying these rules will yield the same result as the ones computed by the former two rules.
In this question, we will use the kitchen environment already provided in RAI. You can create it using the code below.

```
C.addFile(ry.raiPath("../rai-robotModels/objects/kitchen.g"))
```

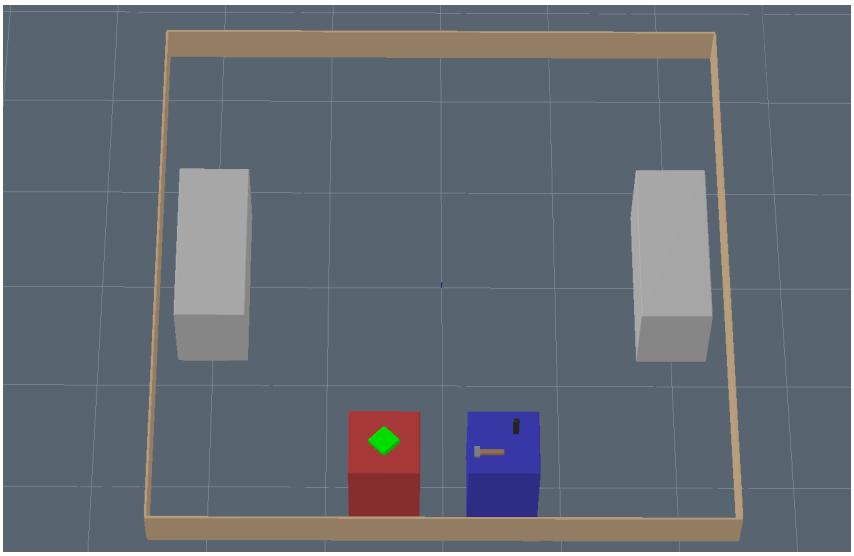
Part A.

- Create a simple hammer object as a **child of** 'sink1' and add it into the environment.
- The hammer object can be composed from two 'ssBox' elements: one representing the handle, sized [0.3, 0.05, 0.03, 0.005], with the color [0.65, 0.50, 0.39], and the other representing the head, sized [0.1, 0.05, 0.03, 0.005], with the color [0.55, 0.57, 0.58], positioned [0.125, 0.0, 0.0].
- The hammer's head should be the child of the hammer's handle.
- You should position the handle [0.1, 0.1, 0.415] according to 'sink1'. (Hint: You can use a quaternion to properly orient the hammer's head during its creation.)

What is the pose (position and orientation) of the hammer's handle according to the World frame?

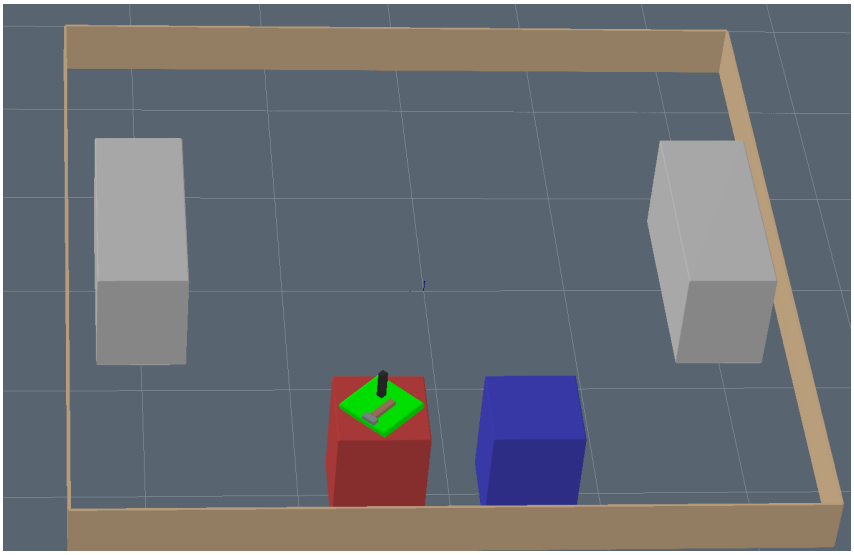
Part B.

- Add an object named "mug" as a child of "sink1" to the environment.
- The "mug" object is a "ssBox" object with the size [0.05, 0.05, 0.2, .01] and the color [0.2, 0.2, 0.2]. Also, the position is [-0.1, -0.1, 0.5].
- Add an object named "tray" as a child of "stove1" to the environment.
- The "tray" object is a "ssBox" with the size [0.2, 0.2, 0.05, 0.02] and the color [0.0, 1.0, 0.0]. The position is [0.0, 0.0, 0.42]. Also, the tray should be rotated by 45 degrees in one of the axis. So, the environment should now be in the configuration below.

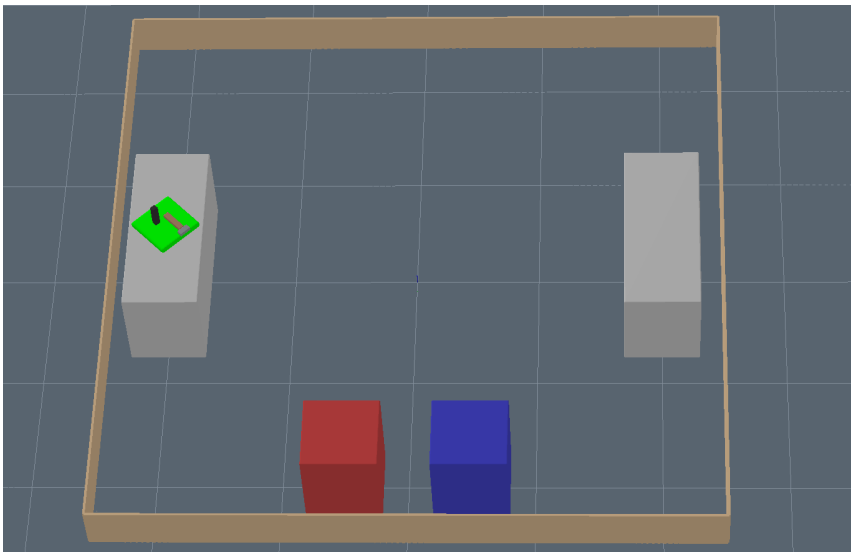


Part C.

- Now, you should make the tray bigger to carry both of the items.
- Place both of the items just above the tray so they should not be in the air or be buried in the tray.
- We call the current state of the environment "State1". State1 should look like below.



- Now you should place the tray above "table1" instead of "stove1".
- We call the current state of the environment "State2". State2 should look like below.



We assume that it is a pure translation from stove1 to table1 for our tray. In this case (during the transition between State1 to State2), how is the relative position of "item1" changed according to "tray" and "item2"? Explain briefly.

Part D. Calculate the relative pose of the mug according to sink1. Also, you can check your results via simulation. Note: In order to understand the kitchen environment better, you can inspect "kitchen.g" file.

2 Two Link Manipulator

In this question, you will use the two-link planar manipulator which we created in the first tutorial.

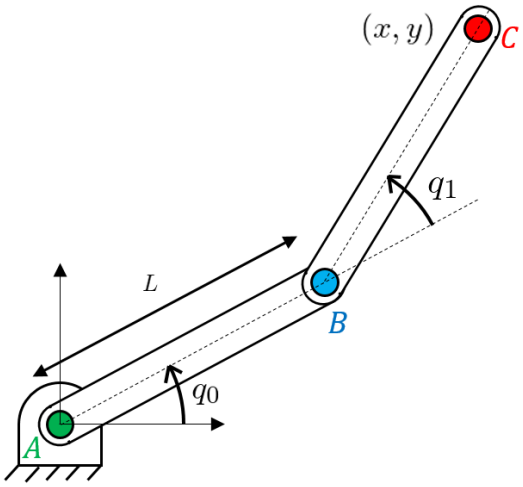


Figure 1: Two link planar manipulator

- In the monogram notation introduced in the slides, the forward kinematics of the manipulator refers to writing down the 2D position of the red point in space with respect to the green point, ${}^{A}p^{C}$, as a function of joint angles (q_0, q_1).

- **In this question, your job is to write down the forward kinematics of the manipulator using trigonometry.** You may use NumPy's trigonometric functions, such as np.sin() and np.cos(), for the implementation.
- You may assume both arm lengths are equal, and their length is given by L = 1.0m
- HINT: If you can write down the 2D position of the blue point with respect to the green point ${}^{A}p^{B}$ as a function of q_0, and the position of the red point with respect to the blue point ${}^{B}p^{C}$ as a function of q_1, then can you write down what ${}^{A}p^{C}$ should be?

Part A. First, create a **.g** file, which is the modified version of the two-link manipulator that is shown in the tutorial. You should change the size of the two links (link1 and link2) and make them 1m long in the Z axis. (You shouldn't change the radius of the links. Change the size of spheres (zero and end-effector) to 0.6.) **Then find the correct joint configurations.** Finally, add the target object using the code below.

```
target: {
    X: "t(0 1 1) d(0 0 0 1)",
    shape: box,
    size: [0.2 0.2 0.2 0.5],
    color: [1 1 0],
    mass: .1,
    contact: true
}
```

The final state of the system should be similar to the figure below:

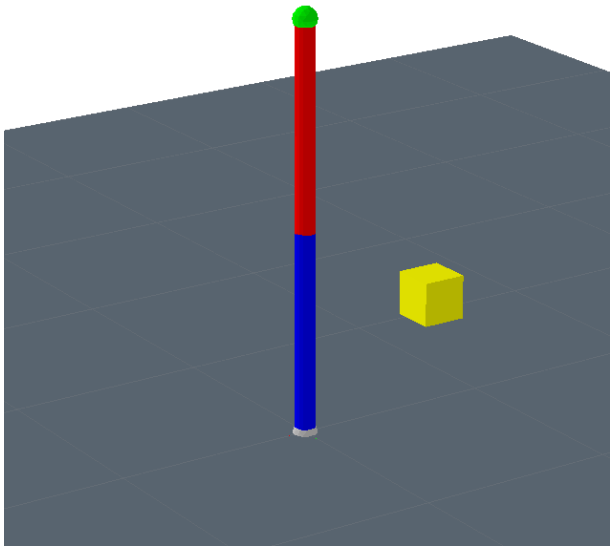
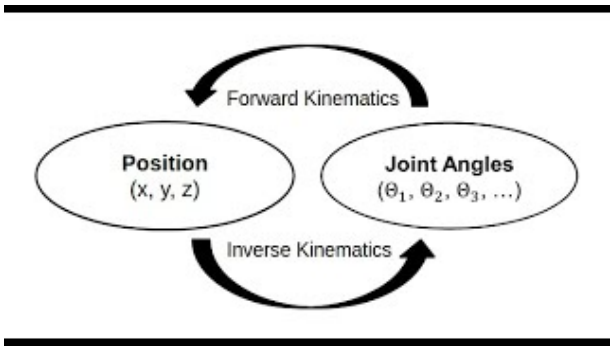


Figure 2: This is the final state of the system. The green sphere is the end effector, yellow box is the target object.

Part B. Write the forward kinematics function to estimate the position of the end effector.



You should fill in the missing spaces in the code below.

```
#You should fill the missing spaces in the code
target = ...
joint_angles = 2 * np.pi * np.random.rand(3)

def forward_kinematics(q):
    q0 = ...
    q1= ...
    y = ...
    z = ...
    return np.array([0,y,z])

pos_target= forward_kinematics(joint_angles)
... #Set the target position in this line
... #Set the joint_angles in this line
K.view()
#Restart the joint configuration for next run
q = np.zeros(K.getJointDimension())
K.setJointState(q)
target.setPosition([0,0,0])
```

When you run your code, you should see that in every single run, the end effector should be placed inside of the target. (You can use "always on top" to fix the ConfigurationViewer window for convenience.)

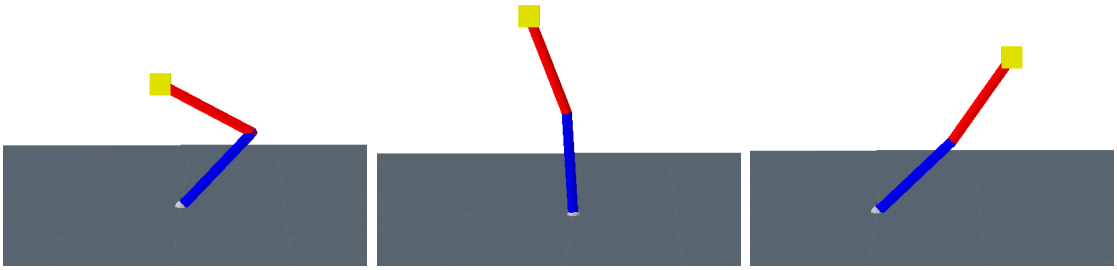


Figure 3: Some example results.

Part C. Write down the 2x2 Jacobian matrix based on the forward kinematics you have derived above. Now that we have the forward kinematics function that gives us our end-effector position given the joint angles.

$$^{A}p^{C} = f(q) \tag{8}$$

Our job now is to derive the translational Jacobian of this simple manipulator. As observed in the lecture, the translational Jacobian is a matrix relating how a change in end-effector position is related to changes in joint angles.

$$d^A p^C = \frac{\partial f(q)}{\partial q} dq = \mathbf{J}(q) dq \tag{9}$$

If you are not familiar with vector calculus, you can write it down even more explicitly as:

$$\mathbf{J}(q) = \frac{\partial f(q)}{\partial q} = \begin{bmatrix} \partial x/\partial q_0 & \partial x/\partial q_1 \\ \partial y/\partial q_0 & \partial y/\partial q_1 \end{bmatrix} \tag{10}$$

NOTE: Don't forget, in our simulation, the z-axis equals the x-axis of the above figure, and the x-axis = 0.
Use these values below to check your result (if you correctly implement the Jacobian function, results should be the same) joint-angles= [1.95205226, 3.15753276, 0]

```
joint_angles= [1.95205226, 3.15753276, 0]
J= Jacobian(joint_angles)
print(J)
```

✓ 0.0s

```
[[ 0.01474768  0.3868342 ]
 [-0.00604877  0.92214929]]
```

Part D. There is one insightful analysis we can do on this Jacobian - when can we invert the Jacobian to successfully recover joint velocities from end-effector velocities? We've seen we can analyze the kinematic singularities of the manipulator through the Jacobian - your job will be to explicitly reason about what they are.

What are the values of (q0, q1) for which we cannot invert the Jacobian? (i.e. What are the kinematic singularities of this manipulator?)

HINT: You should be able to identify two classes of configurations.

3 Spatial Velocity for Moving Frame

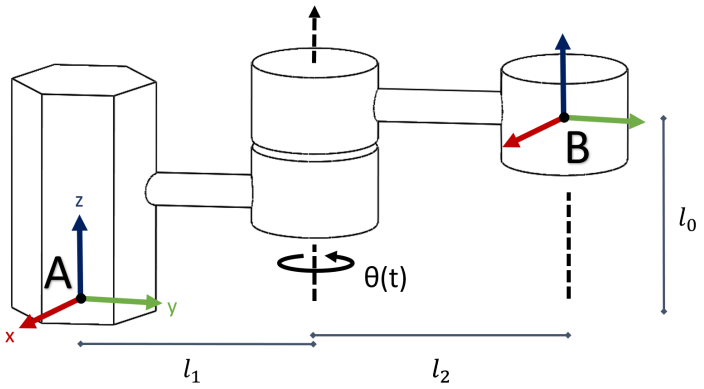


Figure 4: A manipulator with one moving joint. Frame A is the robot base, while frame B is on the end-effector.

A manipulator with one joint is shown above. The joint is rotating over time, with its angle as a function of time, $\theta(t)$. When $t = 0, \theta(t) = 0$ and the link(arm) is aligned with the y-axis of frame A and that of frame B. For this exercise, we will explore the spatial velocity of the end-effector.

Part A. For this manipulator, given ${}^{B}p^{C}(0) = [1, 0, 0]^{T}$, write ${}^{A}p^{C}(0)$. For a generic point C, derive 3 x 3 rotation matrix ${}^{A}R^{B}(t)$ and translation vector ${}^{A}p^{B}(t)$, such that ${}^{A}p^{C}_{A}(t) = {}^{A}R^{B}(t){}^{B}p^{C}_{B} + {}^{A}p^{B}(t)$. Your solution should involve l_0, l_1, l_2, and $\theta(t)$.

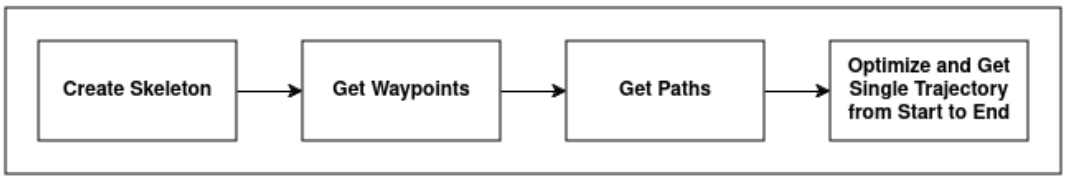
Part B. Prove that for any rotation matrix R, $\hat{\omega} \equiv \dot{R}R^{-1}$ satisfies $\hat{\omega} = -\hat{\omega}^{T}$. Plug in $R =^{A} R^{B}(t)$ to verify your proof ($\dot{R}$ is the derivative of rotation matrix w.r.t. t).

Part C. Solve for ${}^{A}V^{B}(t)$ for the manipulator, as a function of $l_0, l_1, l_2, \theta(t)$, and $\dot{\theta}(t)$.

4 Cargobot

In this question of the homework, you will work on the Cargobot robot. Cargobot is a mobile-based robot designed for cargo transportation. You are responsible for building a motion planning model for this robot. Your task is transporting the object to desired locations with the Cargobot in the kinematic world. For this task, you do not need a dynamic engine (BotOp); you can use Skeleton and KOMO classes.
NOTE: In your solutions, the robot should not penetrate the walls.

Cargo Delivery. A Panda Franka robot has been integrated into the mobile base. You should pick up the cargo from the initial configuration and place it in the goal (blue) area. While doing this, you should utilize the Skeleton object/class. One of the most important aspects here is creating a Skeleton list that is robust. In other words, in the Skeleton, the task should be accomplished, intermediate points that KOMO can handle should be placed, and the correct features should be selected. First, please import "cargobot.g", "cargo.g" and "maze.g" files.
In this question, you are given two checkpoints and one final destination. The robot should pass through these checkpoints. You can find these checkpoints and the final destination in the "cargo.g" You do not need to change anything in "cargobot.g" and "maze.g" files.
Hint: You can add more intermediate points (similar to checkpoints) to robustify your solution in your Skeleton.
To solve the question, you can follow the structure below.



As an output, an optimized, single trajectory should be seen without any collision. The solution can be achieved in a kinematic environment and visualized via "KOMO.view_play()" function.