# Assignment 4: Inverse Kinematics, Trajectories, and Control

Robot Kinematics and Dynamics

Prof. Jeff Ichnowski
Shahram Najam Syed
Yuemin Mao

## Contents

1 Overview 3

2 Background 4
- 2.1 Analytical IK . . . . . 4
- 2.2 Numerical IK . . . . . 4
  - 2.2.1 Advantages . . . . . 5
  - 2.2.2 Disadvantages . . . . . 5
- 2.3 Trajectories . . . . . 5
  - 2.3.1 Bang-Bang Control (No Trajectories) . . . . . 6
  - 2.3.2 Constant Velocity (Better Than Nothing) . . . . . 6

3 Instructions 7

4 Code Questions 11

5 Hands-On Questions 17

6 Submission Checklist 19

## 1 Overview

This assignment reinforces the following topics:

- Analytic IK
- Numerical IK
- Trajectories for Robot Control

# 2 Background

## 2.1 Analytical IK

One approach to IK is to solve for joint angles given a workspace end effector position manually, via algebraic methods or geometric analysis, for each different type of manipulator. This is very different from FK, which is pretty much formulaic, going through the transforms of each link to get the end effector position. The range of arms for which analytical solutions exist are only arms that are fully constrained, and solutions generally have to be further manually inspected to check for an arm exceeding joint limits or other practical considerations.

Analytical IK has the advantage that it is precise, fast (once you figure out the solution), and global, meaning that you can always find a solution for joint angles given an end effector position that is within the arm's workspace. So in short, with analytical IK:

- If there is a solution(s), you will find it.
- If there is no solution, you will know that.

A more flexible method we will discuss below is numerical IK, which can work around most of the limitations posed by things like redundancy and joint limits. However, it is approximate and local, meaning that you are not always guaranteed to find a 'good' solution.

## 2.2 Numerical IK

An alternative approach to analytical IK is numerical IK, or optimization-based IK. In this case we use the forward kinematics and a cost function to try to find a solution to the inverse kinematics. The general idea is to design a function that takes in a desired end effector pose and produces a scalar 'cost' (often an error based on distance to the desired pose) that an optimizer will try to minimize by varying the input variables. In this case the input variables are joint angles which are used to calculate the forward kinematics as part of the cost function.

Overall the process works like:

1. Pick a desired end effector pose
2. Pick an initial arm configuration (this is important detail; usually you are already in a configuration that you wish to start from)
3. Evaluate the forward kinematics at the current configuration
4. Evaluate the update rule for the input variables (e.g., gradient descent, a method based on the gradient of the cost function)
5. Check for 'stopping conditions', often based on the change in cost or the number of iterations (e.g., for gradient descent this can be related to the magnitude of the gradient); if we meet these stopping conditions, then return the current value of the input variables as our solution
6. Update the input variables and go back to step 3

Different optimizers have different methods of picking the next configuration to try, but most are based on gradient methods. Essentially, they try to figure out the slope of the cost function and trying to go 'downhill' until they reach a minimum.

There are distinct advantages and disadvantages of numerical IK compared to analytical IK.

### 2.2.1 Advantages

- Numerical IK is a 'black box'. As long as you have forward kinematics for an arm, you can make a cost function, pick an optimizer, and solve the inverse kinematics. It's general and relatively straightforward to implement.
- Numerical IK can handle arms that are under-constrained (redundant) or over-constrained.
- You can modify the cost function for numerical IK to include whatever you want. Joint limits, knowledge of obstacles, other high-level requirements, etc. can all be added relatively easily.

### 2.2.2 Disadvantages

- Numerical IK is a 'black box'. You do not necessarily get any intuition into the underlying structure of the problem. There might be multiple solutions to a desired pose, or none at all. You have to be careful with how you interpret and use the result that an optimizer spits out.
- Solutions are local, and subject to initial conditions. Optimizers all need an initial guess on where to start, and this guess can dramatically affect the solution the optimizer finds. The most reliable way to use numerical IK for arms is to start in a configuration that is as close as possible to the desired configuration, essentially 'warm-starting' the optimizer instead of searching over large areas of the configuration space.
- Numerical IK is usually less efficient computationally than analytical IK. Frequently this is not the limiting factor for controlling a robot. However, there are some cases where you might want to sample the workspace a huge number of times (one example of this in robotics research is the *RRT*, or Rapidly exploring Random Tree) and in such cases the speed of your IK might be the limiting factor.

## 2.3 Trajectories

When controlling a robotic arm, it is usually important to handle things in a way that is more sophisticated than just 'banging' the arm from one configuration to another. The general idea is to not command anything a robot physically do. Although we have not yet discussed dynamics in the class, dynamics affect the ability to control the position or velocity of any manipulator in the physical world. In short:

- Real robots have mass, $m$
- $F = ma$
- $F$ is limited, so we should be careful about our desired $a$.

One way to mitigate effects of the dynamic limits of an arm is to use *trajectories* to smoothly move the arm from one configuration to another. (When thinking about trajectories, you can pose them as trajectories in work space or configurations space. Here we'll just think of trajectories in configuration space, but in the code problems you'll work with workspace trajectories as well.)

### 2.3.1 Bang-Bang Control (No Trajectories)

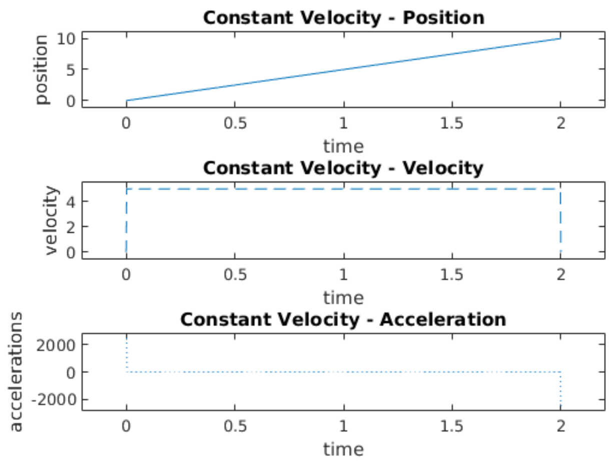
The simplest method is to ignore the pesky bounds of reality and rely on the underlying controller of the robot's actuators to 'catch up' to your commands. Depending on the specific application, and they way the low-level controllers are tuned, this can actually work. However, it has the disadvantage that the robot's motion is less predictable since the low-level controllers dictate the details of how the robot actually moves between the start and goal position – which is something that you often want to have control over! Additionally, with this approach you are usually 'jerking' the arm aggressively which can be dangerous and put unnecessary wear and tear on the arm's joints.

### 2.3.2 Constant Velocity (Better Than Nothing)

The next-simplest method you might think of would be to respect the reality that a robot's joints can't move with infinite speed. Looking at the case of moving a single DOF between two angles, we can choose an amount of time to take to move between the angles and command a corresponding constant velocity.

$T_{\text{move}} = (\theta_{\text{end}} - \theta_{\text{start}})/V_{\text{cruise}}$

This is a step in the right direction, but it has issues where we start and stop. To see the problem, let's think about the position, velocity, and acceleration.



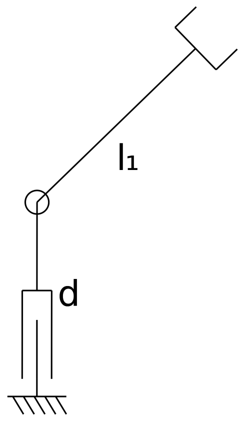
Because we jump from stop to a constant velocity and back again, at the ending steps we see big spikes in acceleration. In theory, these spikes are infinite acceleration, and in practice we will see some large finite value depending on how finely we discretize time (size of our $\Delta t$ in the simulation or arm controller). Large accelerations are undesirable because if the arm has any mass, then we are commanding large torques that are usually outside the capabilities of the robot. Again, the goal of using trajectories is to always command things the arm can physically do, so that it is playing 'catch up' as little as possible.

## 3 Instructions

- The deadline for this project is 3rd October, 2024 09:00 P.M.
- Zip your code into a single file named `<AndrewId>.zip`. See the complete submission checklist at the end, to ensure you have everything. Submit your PDF file to Gradescope.
- Each question (for points) is marked with a **points** heading.
- **Start early!** This homework may take a long time to complete.
- **During submission indicate the answer/page correspondence carefully when submitting on Gradescope.** If you skip a written question, just submit a blank page for it. This makes our work much easier to grade.
- If you have any questions or need clarifications, please post in Piazza or visit the TAs during the office hours.
- Unless otherwise specified, **all units are in radians, meters, and seconds**, where appropriate.

1) Analytical IK: PR robot

Given the PR Arm illustrated below from HW1.



a. [12 points] Analytically solve for $d$ and $\theta$, which result in an end-effector pose of $\left[\begin{smallmatrix} x_e \\ y_e \end{smallmatrix}\right]$.

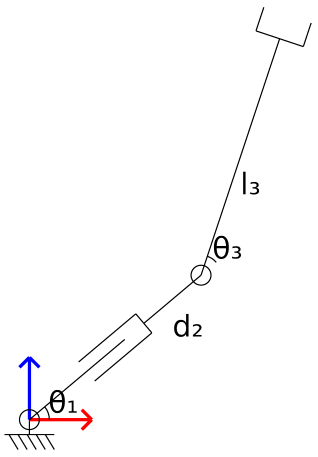
b. [8 points] Assume prismatic joint $d$ can go from 0 and 1 and $l_1$ is 1. Using your analytic IK from above, solve for $d$ and $\theta$ that results in an end-effector position of $\left[\begin{smallmatrix} 0.25 \\ 1.5 \end{smallmatrix}\right]$.

c. [8 points] Assume prismatic joint $d$ can go from 0 and 1 and $l_1$ is 1. Using your analytic IK from above, solve for $d$ and $\theta$ that results in an end-effector position of $\left[\begin{smallmatrix}0.25\\0.5\end{smallmatrix}\right]$.

d. [8 points] Assume prismatic joint $d$ can go from 0 and 1 and $l_1$ is 1. Using your analytic IK from above, solve for $d$ and $\theta$ that results in an end-effector position of $\left[\begin{smallmatrix}-0.25\\-0.75\end{smallmatrix}\right]$.

2) Analytic IK: RPR robot

Consider the following robot



(Define the entire distance between the revolute joints as $d_2$).

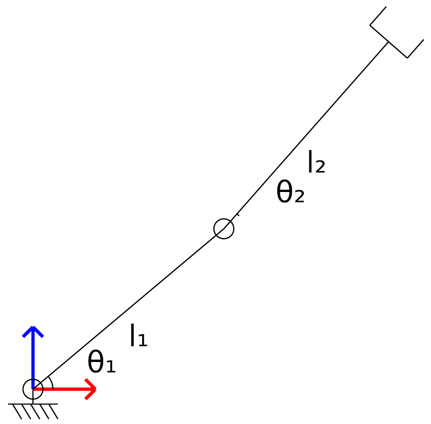
a. [21 points] Geometrically solve for $\theta_1$, $d_2$, and $\theta_3$ which result in an end-effector pose of $\begin{bmatrix}x_e\\y_e\\\theta_e\end{bmatrix}$. Show your work! HINT: Break this problem down into an easier subproblem;

are there known workspace positions of other parts of the robot, given the end effector pose?

b. [9 points] Assume there are no joint limits, except that $d_2 >= 0$. What (if any) limitations are there on achievable end-effector poses $\begin{bmatrix} x_e \\ y_e \\ \theta_e \end{bmatrix}$. Explain your answer in 1-2 sentences.

3) Numerical IK: Cost function

Consider the following robot



a. [10 points] Let $p_e = \begin{bmatrix} f_x(\theta) \\ f_y(\theta) \end{bmatrix}$ (for forward kinematics $f$ and joint configuration $\theta$) be the end effector position as a function of the configuration $\theta$. The goal is to find $\theta$ which achieves an end effector position as close as possible to the desired position $p_d = \begin{bmatrix} x_d \\ y_d \end{bmatrix}$. Let cost function $h$ be the square of the distance between $p_e$ and $p_d$. Write this cost function in terms of the forward kinematics function $f(\theta)$ and $p_d$ (e.g. you may use $f_x$ and $f_y$, but do not use $l_1$ or $l_2$).

b. [8 points] Write the gradient of $h$, $\nabla h(\theta) = [\frac{\partial h}{\partial \theta_1}, \frac{\partial h}{\partial \theta_2}]^T$, in terms of $f$, $p$, and partial derivatives of $f$ with respect to $\theta_i$. Note: do not actually write out the derivatives of $f$; just use, for example, $\frac{\partial f}{\partial \theta_1}$. HINT: if you are not familiar with vector differentiation, first write out $h(\theta)$ as a scalar expression by multiplying the individual vectors, and then compute the required partial derivatives as given above in the definition of $\nabla h(\theta)$.

c. [10 points] Reorder these terms to write this expression in terms of the end effector Jacobian, $J$, along with $f$ and $p$.

d. [6 points] What would the expression for the gradient from the previous part look like for a 100 link serial chain of revolute modules? Is there any difference? Describe why or why not.

## 4 Code Questions

Copy the Code Handout folder to some location of your choice. Open your favorite code IDE and navigate to that location. Whenever you work on the assignment, go into this directory and run scripts.

We will extend the Robot class you created in the last assignment; begin by copying this `Robot.m` file into the root of the Code Handout folder.

1) [20 points] Numerical Jacobian
   Follow the question below:

   a. **Implementation**

   The first task in this assignment is to write the last Jacobian you'll ever need – a Jacobian that can work for any frame generated by the forward kinematics. You will fill in the `jacobians` function in the `robot.py` file; copy and paste from the template provided in the `robot.py` file.

   This function computes a $3 \times n$ Jacobian (where $n$ is the number of joints) for each of the $n + 1$ frames in the diagram.

   The code describes the conventions used for the arguments and result.

   Whereas in previous assignments you have calculated the analytic derivative for each element of the Jacobian, here you will use a numerical approach.

   Given a general function $g(x)$, one can compute a numerical approximation of the derivative $g'(x)$ by computing $(g(x + \Delta x) - g(x - \Delta x))/(2\Delta x)$.

   In the code, we will take the same approach for each element of the Jacobian. (Note – as with the analytical approach, the rotation fields are computed by inspection for now). For example, for the element of the Jacobian corresponding to the $x$ value for frame $i$ and joint $j$ (given a joint configuration $\mathbf{Q}$), one can evaluate the forward kinematics $f_x$ at $\mathbf{Q} + \Delta\mathbf{q}_j$ and at $\mathbf{Q} - \Delta\mathbf{q}_j$, and divide the difference by $2\epsilon$ (where $\Delta\mathbf{q}_j$ is a vector of zeros with $\epsilon$ for element $j$).

   b. **Verification**

   To verify the results, run the `sample_velocities.py` function, and verify that:

   (i) On the first plot, the paths match *and* the end effector velocity vectors are tangent to the path that it takes.

   (ii) On the second plot, the $x$ and $y$ velocities match.

   Also, use this function to find the Jacobian for an RRR arm at configuration $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$. Look at the third row of the Jacobian for each frame – the effect of joint angle velocity on the orientation of the frame. For the first frame (recall, attached to the first link), only the first joint angle should have an effect (e.g., this row should be $[1, 0, 0]$). For the second frame, the first and second joint angles should have an effect. The third and fourth frames (third link and end effector, respectively) should each be affected by the motion of every joint.

   **Note: You do not have to submit anything for this part; however, you should use this to detect any problems with your code before continuing!**

c. **Application**

Because we have a Jacobian for any planar revolute-joint based arm, we can now address problems that would have been much more difficult before. Imagine a RRRRRRRRRR arm (10 R's), with links each equal to $5$. Using the `Robot` class you have written, compute the joint torques necessary to apply a wrench[1] of $\begin{bmatrix} 0 \\ 5 \\ 2 \end{bmatrix}$ with the end effector when the robot is at the following configuration:

$$\mathbf{\Theta} = \left[\, 0, 0.1, 0.2, 0.3, 0.4, 0, -0.1, -0.2, -0.3, -0.4 \,\right]^T$$

The python expressions to compute the result (e.g., creating a robot object, calling the `jacobians` function, and multiplying the result) is already implemented in the file titled `jacobian_example.py` in the `ex_01` directory.

(Just for fun, imagine the number of terms for the analytic Jacobian for this arm. If you could write that out the very first time without making a mistake or a sign error, you might be the first person to ever do so...)

2) [40 points] Numerical Inverse Kinematics (IK)

Follow the question below:

a. **Implementation**

The next extension to the `Robot` class will be a function to compute inverse kinematics using gradient descent, as discussed in lecture. You will fill in an `inverse_kinematics` function in the `robot.py` file; copy and paste from the template provided in the `robot.py` file.

This function accepts a vector of joint angles for initial conditions and a desired end effector location. It returns the resulting joint angles that are the solution of the gradient descent process.

The code describes the conventions used for the arguments and result.

For this problem, assume the cost function is the sum of squared errors between the end effector and the goal point, ignoring joint angle limits. Use the solution to the second written problem to help compute the gradient of the cost function, and use a loop to continue updating the joint angles until no significant progress is made. **REMINDER**: You use the *negative* gradient in this process, multiplied by a small step size $\alpha$.

Your code should handle two cases:

- If the goal is a $2 \times 1$ vector, it should only consider the $[x, y]^T$ position of the end effector.
- If the goal is a $3 \times 1$ vector, it should consider the $[x, y, \theta]^T$ pose of the end effector.

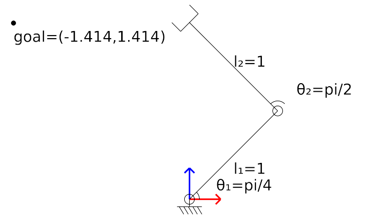
**HINT 1**: Note that the `end_effector` method in the `Robot` class returns the end effector's $[x, y, \theta]^T$ pose. This will be very useful in coding the equation!

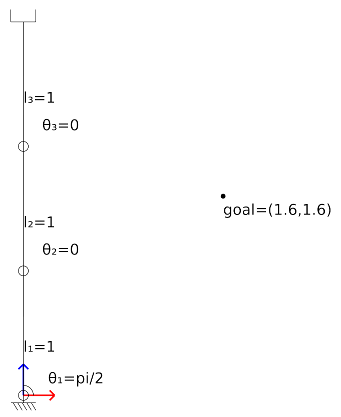
**HINT 2**: For the $3 \times 1$ goal vector case, note that the form of the solution does not change; you must simply use an additional component of the pose and update your calculations accordingly.

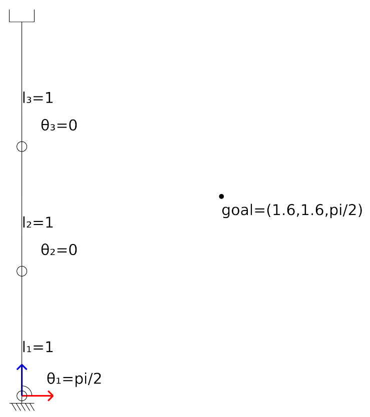
[1]Recall this is a combined force and torque vector; the first and second terms are forces in $x$ and $y$, and the third term is a moment.

b. **Verification**

To verify the results, solve the examples illustrated in the following figures:







**Note**: You do not have to submit anything for this part; however, you should use this to detect any problems with your code before continuing!

Use the listed initial configurations and compute the joint angles. For example, for the first problem, this would be:

```
robot = Robot(
    link_lengths=np.array([1, 1]),
    link_masses=np.array([1, 1]),
    joint_masses=np.array([1, 1]),
    end_effector_mass=1
)
initial_angles = np.array([np.pi / 4, np.pi / 2])
goal = np.array([-np.sqrt(2), np.sqrt(2)])
final_angles = robot.inverse_kinematics(initial_angles, goal)
```

For these three problems, the resulting joint angles should achieve the goal end effector position when passed into the forward kinematics function. You can test by running:

```
robot.end_effector(final_angles)
```

Note that for examples 2 and 3, the resulting joint angles are different because of the inclusion of the end effector orientation constraint.

A caution here is that numerical methods for IK are *local* – for example, the first problem above has two solutions (elbow up/elbow down), but numerical IK only finds the closest one to the initial configuration (or in the case of constraints such as joint angles, may not find any valid solution). This is important to consider for real-world applications.

This local approach can be a benefit over analytical methods, though: for example, in the first problem, if the goal was $[-2, 2]^T$, analytical IK would return no solution, whereas numerical methods would still minimize the cost as best as possible and result in a "close" solution.

Another note: you can make the optimization more accurate by decreasing the step size and stopping condition; however, for our 'plain' gradient descent implementation, this can result in long runtimes. There are built-in numerical optimization methods in Python (e.g., from `scipy.optimize`) that are much more efficient than our simple approach here, resulting in more accurate results in less time.

3) [40 points] Trajectories

When controlling a robot, one option is to just command a destination joint angle configuration and allow the robot's internal joint angle controllers to move the joints towards this goal. However, depending on the starting position of the robot, this can result in large dangerous motions and will not give you any control over *how* the robot gets to this destination. Are there obstacles you want the robot to avoid? Do you want the resulting motion to be smooth?

Instead, one can define a *joint angle trajectory* for the robot's joints to follow through this motion to the goal point and then iteratively command points along this trajectory to result in a smooth, controlled motion along a well-defined path.

a. **Straight Line Joint Trajectories**

Given a starting joint configuration and a goal joint configuration, one option is to connect the start and ending joint angles with a straight line in *joint space*. This means that you linearly interpolate between the start and end joint angles[2].

In this problem, you will fill in the `linear_joint_trajectory` function in the `linear_joint_trajectory.py` file. This function takes a NumPy array of start angles, a NumPy array of end angles, and a trajectory resolution (number of points). It

[2]In Python, the function `numpy.linspace! will linearly interpolate between two numbers with a given number of points`

linearly interpolates between the angles to reach the end points at the given number of points. It will return a matrix of $n$ angles by $t$ timesteps.

b. **Verification**

For an RRR arm with unit link lengths, interpolate between the start state

$$\boldsymbol{\Theta}_{\text{start}} = \begin{bmatrix} \pi/4 \\ \pi/2 \\ \pi/2 \end{bmatrix}$$

and the end state

$$\boldsymbol{\Theta}_{\text{end}} = \begin{bmatrix} 0 \\ \pi/2 \\ 0 \end{bmatrix}$$

with 100 steps.

Run the `visualize_trajectory.py` function with the relevant robot and resulting trajectory matrix as the argument. The joint angle plots should be straight lines, as this is a linear interpolation in joint space. What does the end effector path look like? Does this make sense?

**Note: You do not have to submit anything for this part; however, you should use this to detect any problems with your code before continuing!**

c. **Linear Workspace Joint Trajectories**

Now assume that you have a start joint configuration and want to reach a workspace goal position $\mathbf{g}$ by moving in a straight line in *the workspace*. This can be useful for moving between obstacles or approaching obstacles for manipulation.

In this problem, you will fill in the `linear_workspace_trajectory` function in the `linear_workspace_trajectory.py` file. This function takes a `Robot` object, a NumPy array of start angles, a workspace goal position (it should accept points in both $\mathbb{R}^2$ and SE(2)), and a trajectory resolution (number of points). This function should return a matrix of $n$ angles by $t$ timesteps which result in a straight line workspace trajectory between the inputs.

The general approach to this problem is to first linearly interpolate between the initial and goal workspace positions and then solve the inverse kinematics (IK) for each of these positions in turn. Note that when solving for the $i^{\text{th}}$ column of the resulting trajectory matrix, you should use the $(i-1)^{\text{th}}$ column as the initial condition for the numerical IK, as this will ensure the resulting trajectory is smooth.

d. **Verification**

We will repeat the verification from the last part of this problem, except we use the workspace position for the end point instead of the joint angles:

For an RRR arm with unit link lengths, interpolate between the start state

$$\boldsymbol{\Theta}_{\text{start}} = \begin{bmatrix} \pi/4 \\ \pi/2 \\ \pi/2 \end{bmatrix}$$

and the end workspace position

$$\mathbf{g}_{\text{end}} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

with 100 steps.

Run the `visualize_trajectory.py` function with the relevant robot and resulting trajectory matrix as the argument. Is the end effector workspace path a straight line? What about the joint angle plots? Does this make sense?

**Note: You do not have to submit anything for this part; however, you should use this to detect any problems with your code before continuing!**

**Additional Notes: End Goal** You may have noticed that the `linear_joint_trajectory` function takes a vector of joint angles, and the `linear_workspace_trajectory` function takes a workspace position.

Although this could be done in different ways, often the first function would be used when moving between joint "waypoints," rather than between a current robot position and a desired workspace position. You will see an example of this in the "teach and repeat" problem in the hands-on part of this assignment.

The workspace trajectory would be difficult to complete if it were between two joint configurations because one would have to enforce a continuous constraint on the joint angles in addition to the linear workspace constraint. This is not in general possible (as a concrete example, imagine an RR arm moving from an "elbow up" to an "elbow down" configuration where the end effector could not move, e.g., a straight line between two identical points in the workspace).

## 5 Hands-On Questions

For this lab, you must complete the tasks in on a real robot in the REL.

1) [100 points] Gravity Compensation

In this section of the hands-on question you will implement and verify the of gravity compensation for the Franka Arm's joints 5 and 6. By computing the Jacobian matrix and calculating the necessary gravitational torques, the goal is to ensure that the robotic arm maintains its position against gravitational forces. The verification process involves comparing the calculated torques with the actual torques provided by the Franka Arm's API and then if the test is passed, your calculated joint torques will be passed into the joint torque controller.

The implementation involves two primary functions:

1. `compute_jacobian(theta5, theta6)`: Calculates the Jacobian matrix for joints 5 and 6 based on their current angles.
2. `compute_gravitational_torques(joint_5_angle, joint_6_angle)`: Computes the gravitational torques required at joints 5 and 6 to maintain the arm's position against gravity.

The `compute_jacobian` function calculates the Jacobian matrix for the RR (Revolute-Revolute) arm based on the current joint angles $\theta_5$ and $\theta_6$. This matrix captures how small changes in joint angles affect the position of the end effector.

The `compute_gravitational_torques` function utilizes the Jacobian to determine the necessary torques at joints 5 and 6 to counteract gravity. By mapping the gravitational forces into joint torques, the robot can maintain its posture without unwanted movements.

The verification process involves comparing the calculated gravitational torques with the actual torques provided by the Franka Arm's API. The steps are as follows:

1. Initialization: Reset the robot to a known configuration and retrieve the initial joint angles.
2. Sweep: Iteratively increment joint 6's angle by 30 degrees, moving the robot to the new configuration each time.
3. Torque Calculation**: For each new configuration, calculate the gravitational torques and retrieve the actual torques from the robot's API.
4. Comparison: Compare the calculated torques with the actual torques, determining if they are within a specified tolerance.
5. Results: Print out the results, indicating whether the gravity compensation was successful (PASS) or not (FAIL) for each step.

**Note**: Replace the `XX.XX` placeholders with actual measured values after running the script.

The results indicate whether the implemented gravity compensation accurately matches the actual torques required by the robot to counteract gravity at joints 5 and 6. A "PASS" result signifies that the torque difference is within the specified tolerance, ensuring reliable gravity compensation.

Table 1: Gravity Compensation Verification Results

| Joint 6 Angle (deg) | Calculated Torque (Nm) | Actual Torque (Nm) | Difference (Nm) | Result |
|---|---|---|---|---|
| 30.00 | XX.XX | XX.XX | XX.XX | PASS |
| 60.00 | XX.XX | XX.XX | XX.XX | PASS |
| 90.00 | XX.XX | XX.XX | XX.XX | PASS |

The successful verification of gravity compensation for the Franka Arm demonstrates the effectiveness of the implemented Jacobian-based torque calculation. By accurately computing the necessary torques to counteract gravitational forces, the robot can maintain its posture and perform precise movements without unintended deviations. This capability is fundamental for tasks that demand high precision and stability, such as delicate assembly operations or intricate manipulation tasks.

2) Submission

To submit, run `creats_submission.py`. It will first check that your submitted files are present and correctly organized, and perform a small sanity check. Note, this is not going to grade your submission! The script will create a file called `handin.zip`.

## 6 Submission Checklist

- ☐ Create a PDF of your answers to the written questions with the name `writeup.pdf`.
- ☐ Make sure you have `writeup.pdf` in the same folder as the `create_submission.py` script.
- ☐ Run `create_submission.py` in Python.
- ☐ Upload `handin-4zip` to Gradescope.
- ☐ Upload `writeup.pdf` to Gradescope.