# MEAM 5200 Lab 3 Report

John D'Ambrosio, Cedric Hollande

April 10, 2024

## 1 Methods

For this lab, I developed python scripts and functions to solve the inverse kinematics problem for the 7 Degrees of Freedom (DOF) Franka Emika Panda robotic arm using a secondary `IK`, which houses several methods aimed at calculating joint velocities to guide the end effector towards a specified target pose. This process involves a series of mathematical and geometrical computations to ensure accurate and feasible joint movements that respect the robotic arm's physical constraints. Below, I explain the specifics of mathematical techniques used and a breakdown of the code within the `IK` class, focusing on the equations and mathematical operations utilized.

1. The first step I took was to compute the displacement between the current and target frames and the axis of rotation between them. To calculate the displacement, I extracted the position vectors from their respective homogeneous transformation matrices, and calculated the displacement vector, $\vec{d}$, as:

   $$\vec{d} = \vec{t}_d - \vec{c}_d$$

   where $\vec{t}_d$ and $\vec{c}_d$ are the position vectors, respectively. To get the axis of rotation, $\vec{a}$, and the angle of rotation, $\theta$, are derived using the rotation matrices from the target and current poses, denoted as $R_t$ and $R_c$, through the function `calcAngDiff`, which internally computes the axis-angle representation.

2. The next step I took was to calculate the euclidean distance and angle between two homogenous transformations. To find the the distance, $d$, I took the norm of the difference between the origins of $G$ and $H$:

   $$d = \|G_d - H_d\|$$

   where $G_d$ and $H_d$ are the translational components of $G$ and $H$, respectively.

   The angle, $\alpha$, is calculated from the rotation part of $G$ and $H$, denoted as $R_G$ and $R_H$, using the trace method:

   $$\alpha = \arccos\left(\frac{\text{trace}(R_G^T R_H) - 1}{2}\right)$$

3. I then created a function, `is_valid_solution`, that checks if the IK solution is within the upper and lower joint limits, and returns a boolean, `success`, if the values are within the limits.

4. To actually perform the IK, I created a function that calculates the joint velocity ($\dot{q}$) to minimize the error between the target and current end effector poses. This involves computing the Jacobian matrix ($J$) and applying either its pseudo-inverse ($J^+$) or transpose ($J^T$) based on the selected method, to solve for $\dot{q}$:

$$\dot{q} = J^+ \begin{bmatrix} \vec{d} \\ \vec{\omega} \end{bmatrix}$$

or

$$\dot{q} = J^T \begin{bmatrix} \vec{d} \\ \vec{\omega} \end{bmatrix}$$

where $\vec{d}$ is the displacement vector, and $\vec{\omega}$ is the angular velocity vector derived from the axis of rotation.

5. A secondary task to keep the joints near their centers was formulated. This task created a normalized offset of all joints from -1 to 1 within their allowed range, which was calculated using

$$2 * \frac{q - center}{upper - lower}$$

where center is the center of the joint limits, and upper and lower are the respective limits. Then, an implied quadratic cost was calcuated by multiplying the negative of this offset by a rate at which we want the joints to return to center.

$$\dot{q}_{\text{center}} = \text{rate} \cdot -\text{center}$$

6. To perform the secondary task, I had to utilize linear algebra and null-space projection, by creating a projection matrix $P$, where

$$P = I - J^+ \cdot J$$

I then dotted this projection matrix with the secondary task vector so that it would not affect the primary task's objective, thus successfully utilizing the properties of nullity.

7. To perform the inverse kinematics, I implemented a gradient descent algorithm to iteratively adjust the joint angles, blending the primary and secondary task velocities to achieve the target pose while maintaining joint configurations within their limits. The iteration continues until the movement is below a minimum step size or a maximum number of steps is reached. The IK returns a $q$, which is the closest-guess joint angles, rollout, which is an iteration history of joint configurations, and success a boolean indicating if the target pose is achieved within tolerances.

## 1.1 Code Explanation

We used Cedric Hollande's code for testing. The code base consists of 8 scripts that work in tandem with each other, with `calculateFK.py` and `calcJacobian.py` forming the computational backbone. The `calculateFK.py` script is tasked with deriving homogeneous transformation matrices and joint positions using the DH parameters. This foundational work supports the `calcJacobian.py` script, which calculates the Jacobian matrix crucial for both forward and inverse kinematics analyses.

Central to our inverse kinematics solution are the `IK_position_null.py` and `IK_velocity_null.py` scripts. The `IK_position_null.py` script is engineered to solve the inverse kinematics problem focusing on achieving a most precise desired end-effector positions. It optimizes joint configurations based on a secondary task in the null space to precisely reach target positions, incorporating constraints such as joint limits and potential singularity avoidance. Conversely, `IK_velocity_null.py` targets the velocity aspect of inverse kinematics. It computes the necessary joint velocities that smoothly direct the end-effector towards its target, facilitating real-time movement adjustments and enhancing the arm's responsiveness to dynamic tasks.

To complement these inverse kinematics solvers, the `calcManipulability.py` script plays a vital role in assessing the robot arm's operational versatility. It calculates the manipulability index, a measure of the arm's ability to execute movements in various directions from a particular configuration, and helps us identify potential singularities and non-feasible solutions.

# 2 Evaluation

## 2.1 Part 1: Secondary Task using Null-Space Projection

For the first part of our evaluation, we chose to analyze the `ellipse` trajectory. We ran this configuration with the null-space projection turned on and off, and observed our results. When removing the `null` variable, and thus removing the joint-centering secondary task, we began to notice large deviations and joint configurations reaching suboptimal positions, which makes sense as the least-squares error would accumulate. These deviations were verified using print statements during RViz testing. When added the null-space projection back, the robot cleanly tracked our trajectory, remaining very close to the neutral position, thanks to the joint-centering secondary task. Here is a video implementing `follow.py`.

## 2.2 Part 2: Testing Different Poses

For testing on the real robot, we created three new poses, and evaluated their performance in terms of convergence time and number of iterations to reach convergence. To see our experimental results testing different poses, you can look at this video. The resulting homogeneous transformations for the configurations tested and their solve statistics are in the table below.

Table 1: Performance metrics for custom trajectory targets.

| Target | Location | Solution t (s) | Iterations |
|---|---|---|---|
| 0 | $\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0.2 \\ 0 & 1 & 0 & 0.4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ | 0.53 | 73 |
| 1 | $\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0.4 \\ 1 & 0 & 0 & 0.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ | 1.21 | 182 |
| 2 | $\begin{bmatrix} 0.707 & 0.707 & 0 & 0 \\ 0.707 & -0.707 & 0 & -0.4 \\ 0 & 0 & -1 & 0.6 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ | 0.58 | 87 |

```
targets = [
    transform( np.array([0., 0.2, .4]), np.array([pi/2,-pi/2,0])           ),
    transform( np.array([0., .4, .6]),  np.array([0,-pi/2,pi]) ),
    transform( np.array([0., -0.4, .6]),    np.array([0,pi,-5*pi/4])          ),
]
```

Figure 1: Targets

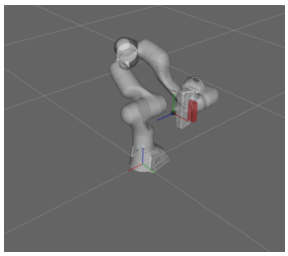

Figure 2: First Config

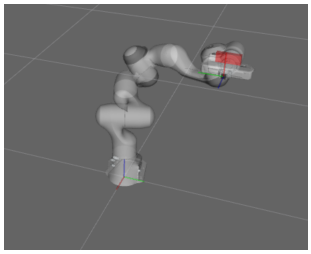

Figure 3: Second Config

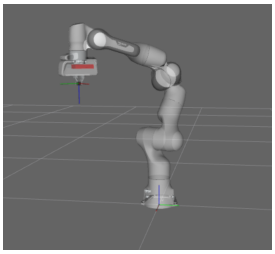

Figure 4: Third Config

To further test the robustness of our solvers, we evaluated their performance on 10 unique poses in simulation. Table 2 summarizes the key metrics, including the time taken to solve, the number of iterations required, and the success rate.

Table 2: Comparison of Jacobian Pseudo-Inverse and Transpose Methods

| **Metric** | **J_pseudo** | **J_trans** |
|---|---|---|
| Number of configurations tested | 10 | 10 |
| Mean elapsed time (s) | 0.36 | 2.09 |
| Median elapsed time (s) | 0.31 | 1.94 |
| Maximum elapsed time (s) | 0.77 | 3.38 |
| Mean iterations | 92.10 | 563.90 |
| Median iterations | 83.50 | 551.00 |
| Maximum iterations | 180 | 1001 |
| Success rate | 1.00 | 0.80 |

The Jacobian pseudo-inverse method outperformed the Jacobian transpose method in terms of both speed and reliability. The pseudo-inverse method converged faster, required fewer iterations, and achieved a 100% success rate across all tested configurations. In contrast, the transpose method exhibited slower convergence, needed more iterations, and had a lower success rate of 80%.

The superior performance of the pseudo-inverse method can be attributed to its ability to provide a least-squares solution, handle redundancy, and exploit the null space of the Jacobian matrix. These properties allow for more accurate and efficient convergence towards the target pose while optimizing secondary tasks. On the other hand, the transpose method fails to explicitly consider the null-space and may struggle in the presence of redundancy or singularities, leading to sub-optimal solutions and potential instabilities.

# 3 Analysis

## 3.1 Part 1: Trajectory Design and Performance Evaluation

### 3.1.1 Custom Trajectory Analysis

For the analysis of inverse kinematics solutions, a custom trajectory was designed to test the algorithm's performance under varying conditions. We chose to implement an ellipse trajectory in the $x$-$y$ plane rather than the $y$-$z$ plane. We observed that the IK solver performed equally as well in this configuration, tracking the trajectory precisely, in real time, while maintaining joint-centering in accordance with the secondary task.

### 3.1.2 Secondary Task Analysis

Adjusting the parameters k_0 to 0.5 and k_r and k_p both to 0.01, it was observed that the robot maneuvered towards the target a bit more slowly, ensuring smoother motion. This adjustment also resulted in the robot more effectively maintaining its neutral position, demonstrating the impact of tuning these parameters on the behavior of the inverse kinematics solver. Specifically, a slower approach allowed for more precise control and adherence to the line-following function, emphasizing the trade-offs between speed and precision in robotic motion control. Here is a video implementing our ellipse code from `follow.py` **without** a secondary task.

## 3.2 Part 2: Algorithmic Analysis

### 3.2.1 Comparison of Two Methods

The Jacobian pseudo-inverse method demonstrates superior performance compared to the Jacobian transpose method. It converges faster, requires fewer iterations, and achieves a higher success rate in finding valid solutions. The pseudo-inverse method's ability to handle redundancy, exploit the null space, and provide least-squares solutions contributes to its optimality and robustness. In contrast, the transpose method's lack of explicit null space consideration and potential struggles with singularities lead to slower convergence and suboptimal results.

### 3.2.2 Uniqueness of Solution

After testing the IK solver with random initial guesses generated using `np.random.uniform` within the allowed joint limits of the Franka Emika Panda robot, I observed that the solver consistently converged to the same or very similar final joint configurations for each target pose, regardless of the initial seed. This suggests that the implemented IK solver is robust and capable of finding valid solutions from various starting configurations within the feasible joint space. The solver's convergence behavior can be attributed to the effectiveness of the Jacobian pseudo-inverse method in handling redundancy and finding optimal

solutions. However, further testing with a wider range of target poses and more diverse initial guesses would provide additional insights into the solver's robustness and potential limitations.

### 3.2.3 Algorithmic Completeness

During testing, we encountered some target end-effector poses for which my IK solver failed to find a solution. However, this does not necessarily mean that the robot is physically incapable of reaching those poses or that my algorithm is incorrectly coded. It is important to distinguish between the limitations of the robot's reachable workspace and the solver's algorithmic challenges. The Franka Emika Panda robot has physical constraints and joint limits that define its range of achievable end-effector poses. If a target pose lies outside this workspace, the solver's inability to find a solution accurately reflects the robot's limitations. On the other hand, if a pose is within the reachable workspace but the solver still fails, it suggests that the solver's limitations, such as getting stuck in local minima or handling singularities, are preventing it from finding a valid solution. Further analysis, such as workspace mapping and forward kinematics validation, along with algorithmic enhancements, can help assess and improve the solver's completeness while acknowledging the inherent challenges of the IK problem.

### 3.2.4 Warm Start

When computing IK solutions for many different poses clustered closely together in the reachable workspace with similar orientations, we can leverage the concept of a warm start to decrease the total runtime and iterations required. Instead of starting the IK solver from scratch for each pose, we can use the solution from the previous pose as the initial guess (seed) for the next pose. Since the poses are clustered closely together and have similar orientations, the previous solution is likely to be a good approximation for the next pose, reducing the number of iterations needed for convergence. By using a warm start, we can take advantage of the spatial proximity and similarity of the poses, allowing the solver to start closer to the desired solution and reducing the overall computation time compared to starting from scratch for each pose. However, it's important to note that while a warm start can improve efficiency, it does not guarantee finding the global optimum solution for each pose, especially if the poses are not sufficiently close or if the initial guess is far from the desired solution.

### 3.2.5 Manipulability

The manipulability index serves as a quantitative measure reflecting the robot arm's capability to execute precise movements across all directions from a given pose. An elevated manipulability index signifies a robust ability for the robot to maneuver and apply forces in a diverse array of directions. In contrast, a diminished index indicates restrictions in the robot's agility and force exertion capabilities.

In the context of approaching a singularity, our observations reveal a pivotal characteristic: the manipulability index tends towards 0. This phenomenon underscores a significant reduction in the robotic arm's dexterity and its ability to initiate movements in certain directions. Singularities in robotic arms are configurations where the arm loses one or more degrees of freedom, effectively constricting its operational envelope.

Table 3: Performance metrics and manipulability index near a singularity.

| **Location** | **Time (s)** | **Iterations** | **Manipulability** |
|---|---|---|---|
| $\begin{bmatrix} 1 & 0 & -0 & 0.33 \\ -0 & -1 & 0 & 0 \\ -0 & 0 & -1 & 0.8 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ | 0.35 | 86 | 0.0677622067 |

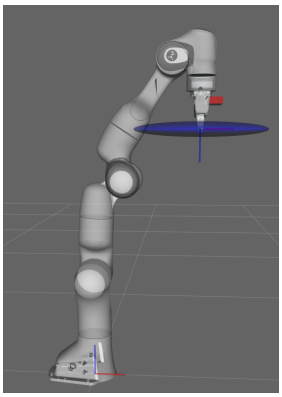

Figure 5: Manipulability near singularity

# 4 Appendix

Video of the robot in different configurations
Video of the robot implementing `follow.py` using secondary task
Video of the robot implementing `follow.py` without secondary task