# Milestone 3: Inverse Kinematics and Drawing

Robot Kinematics and Dynamics

Prof. Jeff Ichnowski
Kartik Agrawal
Aman Tambi

# Contents

# 1 Overview

In this capstone, you will form a team of 3 students to program a Franka Emika Panda robot to draw with lights to create a long-exposure image.

Table 1: **Franka Panda Joint Limits**—Documented limits for the robot joints. Configuration limits have an upper and lower bound that cannot be exceeded due to hardware limits. Velocity, acceleration, and jerk limits list the upper bound in any direction—these may be enforced by software limits in the robot, and exceeding them may result in unexpected operation. Torque limits are limited by the capabilities of the motors.

| Joint | $q_{\min}$ [rad] | $q_{\max}$ [rad] | $\dot{q}_{\max}$ [$\frac{\mathrm{rad}}{\mathrm{s}}$] | $\ddot{q}_{\max}$ [$\frac{\mathrm{rad}}{\mathrm{s}^2}$] | $\dddot{q}_{\max}$ [$\frac{\mathrm{rad}}{\mathrm{s}^3}$] | $\tau_{j_{\max}}$ [N m] | $\dot{\tau}_{j_{\max}}$ [$\frac{\mathrm{N\,m}}{\mathrm{s}}$] |
|---|---|---|---|---|---|---|---|
| 1 | $-2.8973$ | $2.8973$ | $2.1750$ | $15$ | $7500$ | $87$ | $1000$ |
| 2 | $-1.7628$ | $1.7628$ | $2.1750$ | $7.5$ | $3750$ | $87$ | $1000$ |
| 3 | $-2.8973$ | $2.8973$ | $2.1750$ | $10$ | $5000$ | $87$ | $1000$ |
| 4 | $-3.0718$ | $-0.0698$ | $2.1750$ | $12.5$ | $6250$ | $87$ | $1000$ |
| 5 | $-2.8973$ | $2.8973$ | $2.6100$ | $15$ | $7500$ | $12$ | $1000$ |
| 6 | $-0.0175$ | $3.7525$ | $2.6100$ | $20$ | $10\,000$ | $12$ | $1000$ |
| 7 | $-2.8973$ | $2.8973$ | $2.6100$ | $20$ | $10\,000$ | $12$ | $1000$ |

Source: https://frankaemika.github.io/docs/control_parameters.html#limits-for-panda

# 2 Introduction

## 2.1 Notation and Terminology

Throughout this document we will use the following terms consistently:

**Configuration** (or config for short) specifies the settable degrees of freedom of a Robot. For Franka robots, this is a 7-element vector where the first coefficient is the angle of the first joint, etc.

**Cartesian** coordinates refers to the $x, y, z$ position in workspace coordinates for translation, and a rotation relative to a fixed workspace identity rotation.

**Position** refers to the Cartesian translation of a an object.

**Orientation** refers to the rotation of an object.

**Pose** refers the the position and orientation of an object.

**Linear interpolation** given a start and end vector, this is an interpolation that starts at the start vector at $t = 0$ and ends at the end vector at $t = 1$. Mathematically $f(x_0, x_1, t) = x_0(1 - t) + x_1 t$, for $x_0, x_1 \in \mathbb{R}^n$.

## 2.2 Inverse Kinematics

Inverse kinematics is the process of finding joint angles that position the robot's end effector at a specific point in Cartesian space. Gradient descent for IK starts from an initial guess of a configuration and iteratively follows the negative gradient to minimize the cost/error between the current end effector position and the target position. A good initial guess can significantly speed

Table 2: **Franka Panda Cartesian Limits**—Documented limits of the robot's motion. Trajectories that exceed these limits may not run properly, so try to keep the trajectories within these limits.

| | Translation | Rotation | Elbow |
|---|---|---|---|
| $\dot{p}_{\max}$ | $1.7\ \frac{\mathrm{m}}{\mathrm{s}}$ | $2.5\ \frac{\mathrm{rad}}{\mathrm{s}}$ | $2.1750\ \frac{\mathrm{rad}}{\mathrm{s}}$ |
| $\ddot{p}_{\max}$ | $13.0\ \frac{\mathrm{m}}{\mathrm{s}^2}$ | $25.0\ \frac{\mathrm{rad}}{\mathrm{s}^2}$ | $10.0\ \frac{\mathrm{rad}}{\mathrm{s}^2}$ |
| $\dddot{p}_{\max}$ | $6500.0\ \frac{\mathrm{m}}{\mathrm{s}^3}$ | $12\,500.0\ \frac{\mathrm{rad}}{\mathrm{s}^3}$ | $5000.0\ \frac{\mathrm{rad}}{\mathrm{s}^3}$ |

Source: https://frankaemika.github.io/docs/control_parameters.html#limits-for-panda

up convergence in gradient descent. You will need to define an inverse kinematic function that returns a configuration $q$ based on this process as:

$$q = \mathrm{ik}(x_{\mathrm{target}}, q_{\mathrm{init}}; \lambda),$$

where $x_{\mathrm{target}}$ is the target pose in $SE(3)$, $q_{\mathrm{init}}$ is the initial guess, and $\lambda \in \mathbb{R}^+$ is a gradient step size. We will often omit $q_{\mathrm{init}}$ and $\lambda$ in most of the following text. You will have to tune $\lambda$, and have a strategy for selecting $q_{\mathrm{init}}$.

When computing the gradient descent, ensure that the joint configurations stay within the limits of the Franka Emika Panda robot. See the $q_{\min}$ and $q_{\max}$ columns of Table. 1. The easiest way to do this is to clamp/clip the configuration in every iteration of the gradient descent.

## 2.3 Trajectories

For the capstone, we recommend switching, as appropriate, between running joint-space or Cartesian-space interpolations. In almost all cases, the start configuration ($q_0$) or pose ($x_0 = \mathrm{fk}(q_0)$) will be given from the robot's current configuration or forward kinematics. The end configuration or pose will be task-based. You can also compute a sequence of interpolations by chaining the end-configuration of one interpolation into the start of the next.

### 2.3.1 Joint-Space Trajectories

Joint-space **l**inear int**erp**olation (LERP) is straight-forward. Compute

$$\mathrm{lerp}(q_0, q_1, t) = q_0(1-t) + q_1 t,$$

where $q_0 \in \mathbb{R}^7$ is the start configuration, $q_1 \in \mathbb{R}^7$ is the end configuration, and $t \in [0, 1]$. This hopefully seems simple, as it is just standard vector times scalar and a vector addition.

### 2.3.2 End-Effector Trajectories

Cartesian-space interpolation is a bit more complicated. If considering just the translation components, you can hold the rotation fixed and linearly interpolate from start to end translation, then use inverse kinematics to get the joint angles. If you want to include rotation too, you will need to interpolate both translation and rotation. Fortunately, there is a really good way to interpolation rotations: **s**pherical **l**inear int**erp**olation or SLERP. SLERP requires converting the rotations to quaternions, interpolating a quaternion, then converting the quaternion back.

$$\text{slerp}(p_0, p_1, t) = \frac{\sin((1-t)\phi)}{\sin\phi} p_0 + \frac{\sin(t\phi)}{\sin\phi} p_1,$$

where $p_0, p_1 \in \mathbb{R}^4$ are the coefficients of unit quaternions stacked into vectors, and $\cos\phi = p_0 \cdot p_1$ (dot product). SLERP can trace the short way or the long way. To ensure taking the shortest path, test if $\cos\phi < 0$, and if so, negate one of the quaternions (remember $p_0$ and $-p_0$ are the same rotation), and recompute $\cos\phi$. You can take a shortcut to compute $\sin\phi$ using the identity $\sin^2\phi = 1 - \cos^2\phi$. Note also that $\phi$ is the angular distance between rotations—you can use this to decide how many steps to take along the interpolation.

Putting together, interpolating in Cartesian space, given $x_0 = [d_0, p_0]$ and $x_1 = [d_1, p_1]$ are the translations (displacements) $(d_0, d_1)$ and rotations $(p_0, p_1)$. Note that $x_0, x_1 \in SE(3)$ and can be expressed as homogeneous matrices and converted to and from a [translation, quaternion] tuple (we leave these conversions as implicit here). Compute interpolation of a configuration $q_t$ at time $t \in [0, 1]$ as:

$$q_t = \text{ik}(x_t) = \text{ik}([d_t, p_t]) = \text{ik}([\text{lerp}(d_0, d_1, t), \text{slerp}(p_0, p_1, t)]).$$

When computing an interpolation, the most sensible setting for $q_{\text{init}}$ in the inverse kinematics computation is the previous configuration computed. Thus, switching the subscript $i$ to reflect the $i^{\text{th}}$ waypoint along the interpolation:

$$q_{i+1} = \text{ik}(x_{i+1}, q_i),$$

where $q_0$ is the start of the interpolation.

### 2.3.3 Trajectory Timing and Execution

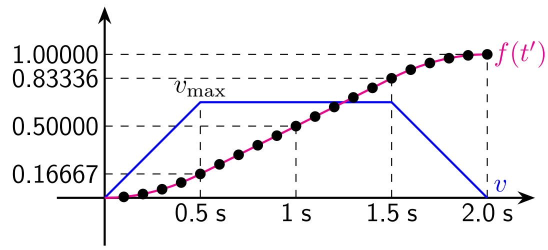
When computing an interpolation-based trajectory, the next thing to consider is velocity and acceleration, thus timing. Robots cannot instantaneously accelerate, and the Franka robot has velocity and acceleration limits, both in joint space (Table. 1) and end-effector/Cartesian space (Table. 2). You should thus smoothly vary the time that you pass to the interpolation function to smoothly accelerate and avoid instantaneous changes. You can usually stay within the documented limits in the tables by trial-and-error or rough estimations.

We recommend starting with a trapezoidal profile, but you are free to try other profiles. Example: suppose you want to interpolate a line in joint space from $q_0$ to $q_1$, where $q_0, q_1 \in \mathbb{R}^7$. You want to interpolate over 2 seconds, accelerating for 0.5 seconds and decelerating for 0.5 seconds. First, define the number of waypoints you want to compute. We'll say 20 waypoints, thus 5 for accelerating, then 10 at constant velocity, and 5 for decelerating.

Our goal is to modify our interpolation from:

$$\text{lerp}(q_0, q_1, t) \qquad \text{to} \qquad \text{lerp}(q_0, q_1, f(t')),$$

where $f(t')$ maps from $t' \in [0, 2]$ to $f(t') \in [0, 1]$. The interpolation will look something like the following:




With $f(\cdot)$ defined, you can then interpolate $t'$ from 0 to 2 seconds and send the waypoints to the robot to run. You can apply the same method to interpolate in Cartesian space.

**Note: The robot controller receives new waypoints every 0.02 seconds, so your interpolation function should ensure that all waypoints are spaced 0.02 seconds apart. Each successive waypoint must be reachable within 0.02 seconds from the previous one-the robot should not exceed its joint limits during the 0.02-second motion to avoid triggering errors. The final output of your interpolation function must be an $n \times 7$ array, where each row represents a waypoint defined by the 7 joint angles.**

## 2.4 Milestone 3: Inverse Kinematics & End-effector interpolations

Due: **Thurs, Dec. 4, 2025**

In this milestone, you will create a numeric inverse kinematics implementation and draw straight as well as curved lines.

# Codebase Files

The codebase contains the following five files:

1. `guide_mode.py` This script allows you to interactively control the robot:
   - Press 1 to get the end-effector pose.
   - Press 2 to get the joint angles.
   - Press 3 to stop the current skill.
   - Press 4 to move the robot to the home position.

2. `FrankaRobot16384.py` This is a wrapper class that you will use for most of the capstone to access different functions of the robot.

3. `example.py` This file contains examples demonstrating how to use the wrapper class

4. `demo.py` This file demonstrates how to incorporate user inputs to put the robot in a standby or recovery mode. The robot waits for a user signal to either retry the current action or move on to the next one.

5. `milestone-3.py` This file contains 3 new functions for you to implement:
   - `inverse_kinematics_numerical(q_init, T_target, step_size, max_iters)` – outputs the joint values to reach the target pose (homogenous transformation).
   - `draw_line(q_start, dp, num_steps` – To create a straight line trajectory in the cartesian space.
   - `draw_line(q_start, curve_fn, num_steps` – To create a curved trajectory in the cartesian space.

# 3 Rubric

**(1) Autograder** Your code will be tested automatically on `inverse_kinematics_numerical` Points will be assigned based on the number of test cases passed.

# 4 Submission Checklist

☐ Upload `all files` to Gradescope.