16-384 Capstone: Light Painting

Jeff Ichnowski

Fall 2025

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.

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 Picking / Grasping

The picking/grasping task in the capstone is the same as in the hands-on section of Assignment 7: moving the robot to a config where the target object is at the center of the gripper and closing the gripper.

2.3 Franka Kinematics

Most robot manufacturers, including Franka Robotics, use DH parameters to represent the robot kinematics. DH parameters of the Franka Emika Panda robot see Fig. 1 and Table 1. (Ref: Franka DH parameters, note that Panda and Research 3 have the same DH parameters.)

You will implement a forward kinematics function using the method introduced on Slide 16 of DH parameter slides. While the given DH parameters allow you to compute the transformation from the robot's base frame to its flange frame, you can implement your forward kinematics function to operate to any arbitrary point by applying additional transformations. For instance, the distance between the center of grasp and the flange frame is 0.1034 m in the z direction, so you can apply an additional transformation using $a = 0, d = 0.1034, \alpha = 0, \theta = 0$ to compute the transformation from the robot's base frame to its center of grasp. Similarly, you can also get the transformation from the robot's base frame to the tip of a grasped light source. Assuming you follow the slides, you will compute forward kinematic map in the form:

$$H_8^0 = H_1^0 H_2^1 \cdots H_8^7.$$

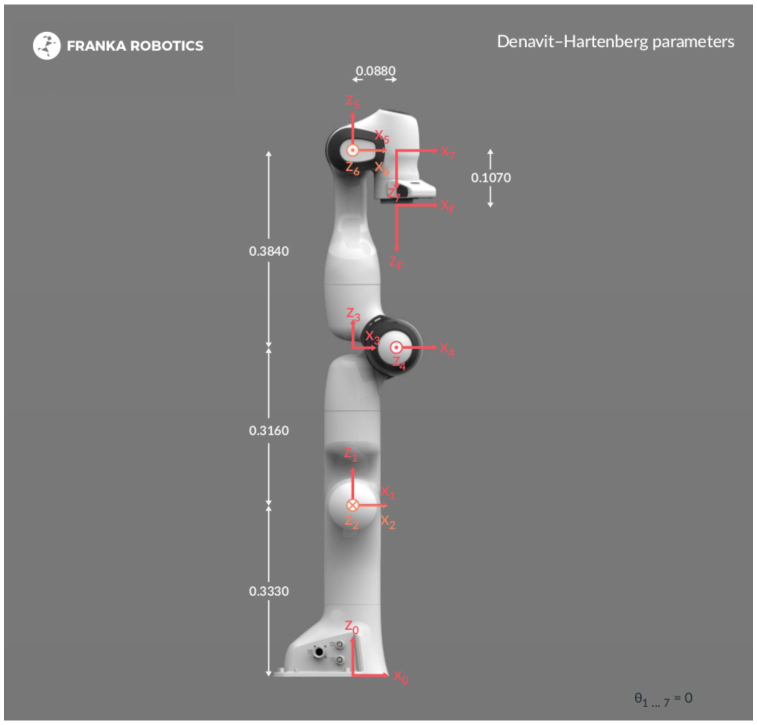


Figure 1: Franka Emika Panda robot joint frames

We will define $\mathrm{fk}(q)$ to be the forward kinematic function that, given a robot configuration $q \in \mathbb{R}^7$, outputs H_0^8.

Joint	a_i (m)	α_i (rad)	d_i (m)	θ_i (rad)
1	0	0	0.333	θ_1
2	0	$-\pi/2$	0	θ_2
3	0	$+\pi/2$	0.316	θ_3
4	0.0825	$+\pi/2$	0	θ_4
5	-0.0825	$-\pi/2$	0.384	θ_5
6	0	$+\pi/2$	0	θ_6
7	0.088	$+\pi/2$	0	θ_7
Flange	0	0	0.107	0

Table 1: DH parameters of Franka Emika Panda robot

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

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

Table 2: **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]	$\dot{q}_{\max}$ $[\frac{\text{rad}}{\text{s}}]$	$\ddot{q}_{\max}$ $[\frac{\text{rad}}{\text{s}^2}]$	$\dddot{q}_{\max}$ $[\frac{\text{rad}}{\text{s}^3}]$	$\tau_{j_{\max}}$ [N m]	$\dot{\tau}_{j_{\max}}$ $[\frac{\text{N m}}{\text{s}}]$
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`

Table 3: **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 $\frac{\text{m}}{\text{s}}$	2.5 $\frac{\text{rad}}{\text{s}}$	2.1750 $\frac{\text{rad}}{\text{s}}$
$\ddot{p}_{\max}$	13.0 $\frac{\text{m}}{\text{s}^2}$	25.0 $\frac{\text{rad}}{\text{s}^2}$	10.0 $\frac{\text{rad}}{\text{s}^2}$
$\dddot{p}_{\max}$	6500.0 $\frac{\text{m}}{\text{s}^3}$	12 500.0 $\frac{\text{rad}}{\text{s}^3}$	5000.0 $\frac{\text{rad}}{\text{s}^3}$

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

where x_{target} is the target pose in $SE(3)$, q_{init} is the initial guess, and $\lambda \in \mathbb{R}^+$ is a gradient step size. We will often omit q_{init} and λ in most of the following text. You will have to tune λ, and have a strategy for selecting q_{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. 2. The easiest way to do this is to clamp/clip the configuration in every iteration of the gradient descent.

2.5 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 ($x_0 = \text{fk}(q_0)$) 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.

2.5.1 Joint-Space Trajectories

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

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

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.

2.5.2 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.

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

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 ϕ 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:

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

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

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

where q_0 is the start of the interpolation.

2.5.3 Trajectory Timing and Execution

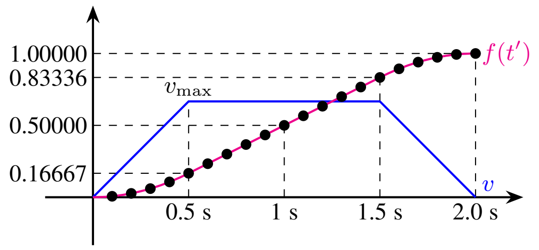
When computing an interpolation-based trajectory, the next thing to consider is velocity and acceleration, thus timing. Robots cannot instantaneously accelerate, and the Franka robot has velocity and acceleration limits, both in joint space (Table. 2) and end-effector/Cartesian space (Table. 3). You 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:

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

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.

2.6 Placing

Placing means positioning an object with the robot, which can have different levels of precision. Dropping an object roughly within an area, as in the hands-on section of Assignment 7, is a simple placing task with low precision requirements. In contrast, inserting a peg into a hole is difficult due to the noise in actuation and sensing, thus requiring a feedback loop to deal with imprecision. In the capstone, you will complete placing tasks with multiple levels of precision without a feedback loop.

2.7 Avoiding Collisions

When planning robot motions, it's essential to consider the risk of collisions, especially during joint-space interpolation. Joint-space interpolation causes the robot to sweep arcs through space as the joints rotate, which may inadvertently sweep through an obstacle, such as the table, the whiteboard, the bin, or the robot itself.

To minimize collision risk:

- **Use smaller increments** This allows for more control when the waypoints are close to obstacles.
- **Use tested interpolations** You can find a safe trajectory first and optimize it by applying minor changes to its waypoints and speed.

2.8 Avoiding Singularities

Singularities can result in erratic or unpredictable motions, which occur more frequently with Cartesian interpolation. When the end effector moves along waypoints defined in Cartesian space, the robot may pass through its singular configurations as the inverse kinematics solver only guarantees each individual configuration is within joint position limits. If you are interpolating in Cartesian space, singularities will show up as the inverse kinematics solver not finding a solution (within the specified tolerance).

To avoid singularities:

- **Avoid high-risk end-effector positions** Avoid moving the end effector to positions where the arm has to fully extend or fold.
- **Use tested interpolations** Similar to collision avoidance, using tested trajectories with small variations can ensure smoother transitions and avoid singularities.

3 Instructions

- The deadline for this project is in-class on Dec. 5, the last day off class. There is no way to extend this.
- See the submission section for details on milestones and final submission.
- Pay attention to the grading rubric.
- **Start early!** The capstone has many parts, and may take a long time to complete.
- 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**.

4 Capstone

The robot must pick up a switched-on light source from a holder located at a known position. It will then move the light source to a predefined start pose, which will be provided by the TAs before the demonstration (and will differ for each group). Once the light source is in the start position, the team will begin recording using a camera mounted on a tripod in front of the robot. The robot will then use the light source to "draw" light patterns in the air, creating long-exposure images. This sequence must be repeated at least three times, each time using a different light source to produce distinct light drawings.

Some important points:

- The robot must complete several tasks that involve picking, placing, forward kinematics and inverse kinematics.
- Everyone on the team must participate in the capstone. It is up to the team to determine how to distribute the effort, but **all team members must know all parts of the code**.
- During the capstone presentation, **one team member, selected at random** by the instructor staff, will run the demo—thus every team member must be prepared to demo from **their own account**.
- Be careful about collisions! Before running any program, have your hand hovering over the e-stop button. Stop the robot as soon as you think something might go wrong—rerunning the program takes a minute, fixing a broken robot takes months.

4.1 Forming Teams

Use the following spreadsheet to form your team. Team names are first-come first serve. Not all team names will be used. Each team should have 3 members. We may need to form a team or two with 4 members. Other team sizes can only be formed with permission from the instructors. `https://docs.google.com/spreadsheets/d/1ILGacVf14mC_ezogwvcBn0_4SwQNdbYmReoEiDCu7b4/edit?usp=sharing`

4.2 Recovery policies and "Sensors"

A.k.a. At first you don't succeed, try, try again. Throughout this explanation, there will be times where something fails. The approach we will take is to allow the robot to try again. Since sensing is not part of this class, the "sensor" input will be someone running the robot code. On a failure, the robot operator can run tell the robot to try again. It is up to you to determine when failures are likely to happen and what recovery steps to take.

You may also provide “sensor” input to tell the robot what to draw (e.g., if you want the robot to play an interactive drawing game). Just make sure you hit all parts of the rubric.

The “sensor” input must be either an observation or a simple command. For example, you can tell the robot that “the light source did not drop” to have it try to drop the light source again. You can also tell the robot to “pick up light source 1” if it failed to pick up the planned light source and you want to have it try another. You can also tell the robot to “draw an X at 0,0.”

If in doubt, ask the TAs on Piazza.

4.3 Home configuration

The robot should first be placed in a home configuration (important, this is not a home pose, it should be specified in joint angles!). This configuration will allow for predictable/repeatable motions from home to the first pick. We suggest using a linear interpolation in configuration space from the home to a pre-pick configuration.

4.4 Picking up a light source

When setting up the environment, we will place a light source holder at a known location. Note: the TAs will provide a range of locations where the light source holder will be during the final demo. The Teams may decide which light source color to place in each light source holder.

To pick up a light source, first ensure that the gripper is on a light source. Then move the gripper to place it vertically over the light source so that when the gripper closes, the light source will be between the fingers. **Note: when the robot approaches the light source, it can collide with things!** Plan a motion that will not.

After securing the light source in the gripper, the robot should lift the light source vertically out of the light source holder. This will require a linear interpolation in Cartesian space using inverse kinematics to resolve the robot configurations along the interpolation.

Note: The light source holder’s orientation is constant but its position varies. You must make sure your code can adapt to different light sourceholder positions.

4.5 Moving the light source to the start point

Once the light source is above the light source holder, determine the coordinate of the point where to start/restart the drawing from. You can try to do a joint-space interpolation, however, be careful, this can fail with a collision. You may instead want to explore strategies that include: (a) moving the robot to a known safe (e.g., “home 2”) position, or (b) moving to a pre-draw position, or (c) some combination or other idea.

Note: The start position of the drawing will vary for each team. You must make sure your code can adapt to different start positions.

4.6 Drawing a line

Drawing a line requires tracing a linear interpolation in world space and converting to joint configurations along the way using inverse kinematics. You’ll want to interpolate the motion at a smooth light source drawing rate.

Between drawing lines, you’ll need pause the long exposure image and therefore your robot must wait for user input before starting new drawing, picking or placing the light source.

4.7 Drawing an arbitrary curve

Once you've got the hang of drawing a line, the next thing to try is to trace a curve, such as a circle, an arc, a polynomial, etc. It's much like tracing a line, but the biggest difference is timing the path. Keep the drawing motion as even as possible. Don't go too fast or too slow.

Between drawing curves, you'll need pause the long exposure image and therefore your robot must wait for user input before starting new drawing, picking or placing the light source.

4.8 Placing a light source in a drop location

After you're done drawing with a single light source, the next thing you'll need to do is put down the light source. Unfortunately, placing the light source back in the holder is likely not to work—go ahead and try if you're interested! So, instead, you should place the light source in a drop bin.

Note: The drop bin's orientation is constant but its position varies. You must make sure your code can adapt to different drop bin positions.

Bonus Point: if you can successfully place the light source back in the holder, that will count as a successful drop and earn bonus points, however you must demonstrate dropping in the drop bin at least once.

4.9 Human-in-the-loop

Humans must not touch or interact with the robot or the environment with the following exceptions:

- Human input on command line for a recovery policy or "sensor" input/command is okay.
- A human can pick up a dropped light source and replace it in the holder or drop bin.

4.10 Painting

When you're done light painting, the long exposure image will show a what you drew. The figure below is an example created by one of the TAs (can you guess who?).

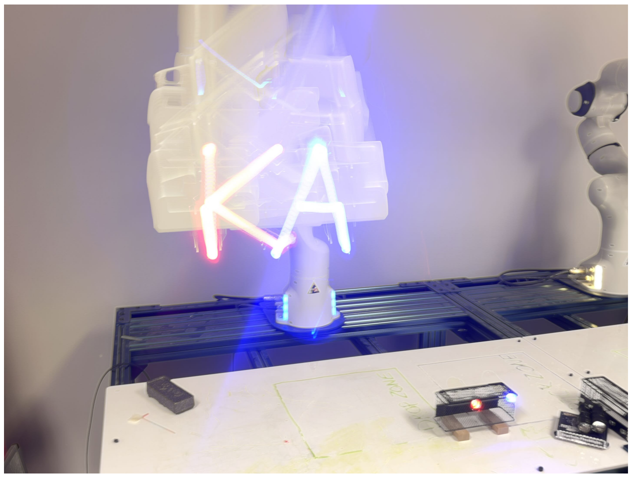


Figure 2: Example robot light painting

5 Milestones/Checkpoints

We will work on the capstone over 3 milestones, due weekly. Detailed instructions on the milestones will be provided in separate documents. This section only provides an overview.

5.1 Milestone 1: Forward Kinematics & Robot

Due: **Thurs, Nov. 13, 2025**

This milestone is meant to (re)familiarize yourself with the Franka in 3D. Using the DH Parameters defined in Table 1 above, develop a forward kinematic map for the robot.

The Forward Kinematic map will be a python function `fk`, that takes a single argument: a list of the 7 joint angles. It should return the 3D homogeneous transform of the end effector. You will test this function with known configurations and end-effector poses.

Once the code works, you will then test it on the real robot. After putting the robot into free-drive or "teach" mode, validate that your forward kinematics works, by moving the robot to a known position, record the joint angles and validate that your forward kinematics map gives you the expected result.

Also in teach mode, move the robot to get it as close as possible to a specific transform. Use the `get_config` function to record the joint angles. Then move the robot into a different configuration while keeping the same end-effector pose and record another set of joint angles. (Hint: have one team member hold the end-effector still, while another moves the arm, and a third records the joint angles.)

5.2 Milestone 2: Trajectories

Due: **Thurs, Nov. 20, 2025**

In this milestone, you will get the robot to interpolate between two configurations smoothly.

In code, write a function `compute_joint_trajectory(q1, q2, time_step, num_steps)`, where the `q1` is the starting configuration, `q2` is the ending configuration, and `time_step * num_steps` is the duration (or end time) of the trajectory. The trajectory it returns should be a list of waypoints where each waypoint is separated by `time_step` seconds (for the robot, this should usually be 0.001 (i.e., 1 ms), matching the robot's control frequency), though larger values may be useful for testing and debugging.

Once the code works, test it on the real robot. For the robot, use teach mode to move the robot to a pose and record the joint configuration, and repeat 3 more times. Generate trajectories that move between successive pairs of recorded configurations in sequence and then close the loop. Get the robot to complete a loop motion within 10 seconds. You will need to call the `follow_trajectory(traj)` method 4 times to close the loop. Use `goto_joints(q1)` to get to the starting configuration, but do NOT use it for anything else.

Note: be careful about the motions between points when collecting them—if you do to much motion, the trajectory you generate could cause a collision.

5.3 Milestone 3: Inverse Kinematics & End-effector interpolations

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

In this milestone you will create a numeric inverse kinematics implementation, then compute and follow trajectories in the end-effector space. To compute your inverse kinematics, you'll first need a Jacobian. To test the Jacobian, you will first create a finite-difference version of it. You will know when your analytic Jacobian implementation matches the numeric Jacobian within a few decimal points.

To then implement the inverse kinematic function, you will implement gradient descent using your Jacobian and the cost function defined in lecture. The `ik(q0, target)` function will take two arguments: `q0` is the initial point from which you will perform the gradient descent (usually you'll use the robot's current configuration as this value). The `target` is the inverse kinematic 3D homogeneous transform target.

Test this on a real robot by having the robot move around in response to keyboard commands (using scripts provided by the teaching staff).

In part II if this milestone, create another function to generate an IK (or end-effector) trajectory. This function will take as an argument, a "trajectory" in homogenous coordinates—i.e., a list of transforms that follows a desired path. The end-effector trajectory will you generate will repeatedly call your ik function to convert the path to joint angles that you can run.

The final part of this milestone is to create the end-effector paths for desired shapes to draw.

6 Demo and Grading Rubric

Each team will have to book a 15 min slot (slot times will be released by the TA's). Each team will have 10 minutes to run their demo. As mentioned above, one team member will be chosen at random to be the robot operator. The operator will have to log in and run all the code from their account. Other team members can give advice and root on the team. The main task and points are listed in bold below. The subtasks are not bold and sum to the main tasks. This rubric is evaluated 3 times, once for each cycle with a different light source. Note that you must draw at least 3 lines and 3 curves, but it does not matter with which light source (but each light source must draw at least one line or curve). So you can draw 10 lines with light source 1, 1 curve with light source 2, and 1 curve with light source 3, and still satisfy the rubric. The sum of lengths of all lines must be 30 cm or more. The sum of lengths of all curves must be 30 cm or more.

The light source holder and the drop bin will be placed at arbitrary locations on the top board of the robot workcell. We will provide the necessary positions and orientations during the final demo—you must make sure your code can adapt to small variations from what you tested your code on.

Task	**Points**
1. Go to home	**5**
2. Go to pre-pick location	**5**
Avoid collisions	2
Place gripper in correct location	3
3. Close gripper and grasp light source	**5**
4. Lift light source vertically from holder	**10**
light source holder does not move	3
light source tip clears holder	5
5. Move light source to start pose facing the camera	**10**
Avoids collisions	2
light facing the camera	8
6. Draw line	**10**
Robot goes on standby to start image capture	5
light source tip motion is straight	5
7. Lift and move light source to next drawing	**10**
No extraneous contact at lift point	3
Avoid collisions / contact	3
Place light source tip at start of next drawing	4
8. Draw curve	**10**
light source motion is smooth (no instantaneous accelerations/jerks)	10
9. Move light source to drop location	**10**
Avoid collisions	2
Over target location	3
At desired orientation	5
10. Open light source gripper to drop light source	**5**
Subtotal (passes $\times$ points)	**3×80**
Bonus point opportunities	**10**
Use four colors in your drawing	5
Place a light source in the holder after drawing (must complete 9 at least once)	5
Total (possible/subtotal)	**250/240**

7 Writeup

In the final submission, each team member must submit an individual 1-page writeup of their (1) contributions to the project, and (2) insights they gained during the capstone. Example insights:

- What did you learn about timing trajectories?
- How accurate/repeatable were particular parts of the process?
- What special trick did you do to avoid collisions or singularities?
- Was there something special you needed to do to tune inverse kinematics?

Submission

1. Upload your video to Youtube and attach the link in your writeup.
2. Include the script you write in your code submission.

Submission Checklist

- Upload your writeup `<andrew_id>.pdf` to Gradescope.
- Upload the code in a zipped folder name `<team-name>.zip` to Gradescope.

8 Capstone Steps

1. Run calibration script
   (a) Determine which point on the light source holder is its origin in its frame – calibrate to that point.
   (b) Determine which point on the whiteboard is the origin. Calibrate to that point first, then calibrate to two other points to determine the plane. Note that calibration will give you a z-axis that is normal to the whiteboard, an x-axis that is left-right on the board, a y-axis that will be the "up-down" (relative to the board, not the world!).
   (c) Determine which point on the drop bin is its origin, calibrate to that point.
2. Call `reset_joints()`.
3. Pick up a light source
   (a) For checkpoint
      (i) Determine pre-pick and lift configurations using the script from Assignment 7.
      (ii) Go to the pre-pick configuration
         (1) Generate linear interpolation in joint-space between two configurations.
      (iii) Close gripper.
      (iv) Go to lift-configuration
         (1) Generate linear interpolation in joint-space between pre-pick and lift configurations.
   (b) After checkpoint
      (i) Determine pre-pick and lift positions using the light source holder origin calibrated with `calibration_workspace.py`.
      (ii) Go to the pre-pick position
         (1) If using a Cartesian interpolation, use IK to compute the configuration for each point along the interpolation. (Remember, you have to provide trajectories in joint configuration over time.)
         (2) If using a joint-space interpolation, use IK once to determine the end configuration, and generate a trajectory between home and this configuration.
      (iii) Close gripper.
      (iv) Lift light source vertically to avoid moving the light source holder (hint: joint-space interpolation will have problems in some cases).
4. Take the light source to the first point on the board for drawing
   (a) Hint 1: Consider which interpolation you want to use here – is one better than the other?
   (b) Hint 2: If you can't get a single interpolation to work, consider interpolation to the home configuration first, before proceeding to the board (note: you are not allowed to call `reset_joints()` here! Or anywhere unless explicitly stated as an option).

5. Interpolate along the board with the light source to do the drawing.
6. Lift light source from the board (optionally repeat from Step 4).
7. Carry the light source to the drop bin oriented so that the light source tip does not hit.
8. Repeat from Step 2 (calling `reset_joints()` is okay here, optional: go directly to the next light source).
9. After the last light source, call `reset_joints()`.