Lab 7: Fullstack Robotics: Perception, Planning, and Control *

EECS/ME/BIOE C106A/206A Fall 2023

Goals

By the end of this lab, you should be able to:

- Use the Sawyer camera to detect an AR tag location
- Implement planning to generate custom trajectories for our Sawyer to follow
- Implement a trajectory-tracking controller through open-loop and closed-loop control.

Contents

Introduction

In Lab 5, you explored the inverse kinematics and path planning functionalities available for the Sawyer. **Inverse kinematics** can produce joint configurations that position a robotic manipulator at a specified end effector position, but it doesn't tell us how to *move* the manipulator between a specified start and end position. Then, **path planning**

*Written by Mingyang Wang and Shrey Aeron, Fall 2023. Name stolen from Lab 8, written by Eric Berndt, Karim El-Refai, Charles Xu, Kirthi Kumar, and Martin Zeng, Fall 2023. Partially adapted from 106B Project 1 (developed by Jeff Mahler, Spring 2017)

solves the problem of choosing a joint trajectory that moves a manipulator between a given start and end configuration while obeying a set of constraints, which you used to execute a simple pick and place task.

Labs 7 and 8 will both culminate with the exploration of **full-stack robotics** – implementing perception, planning, and control algorithms to develop your own fully functional robotics application.

Before every lab, you should pull the updated starter code. Navigate to the `ros_workspaces` directory and run

```
git pull starter main
```

to pull the lab 7 starter code.

1 Overview

1.1 Code Overview

The starter code for this lab is more complicated than usual, so we will provide a brief overview.

The primary files you will be interacting with are listed below:

```
Lab7
  src
    ar_track_alvar
    baxter_pykdl
    sawyer_full_stack
      launch
        custom_sawyer_tuck.launch
        sawyer_camera_track.launch
      scripts
        main.py
      src
        controllers
            controllers.py
        paths
            paths.py
            trajectories.py
            path_planner.py
```

The primary file you will be running in this lab is `sawyer_full_stack/scripts/main.py`, which utilizes the following command line arguments:

- `-ar_marker {number}`
- `-task {task}`: *line* or *circle*.
- `-controller {controller}`: *moveit*, *openloop*, or *pid*

You will be tasked with implementing the following functionality:

- Perception: use the Sawyer camera to locate the specified AR tag
- Planning: `main.py` will generate a target trajectory for the Sawyer by creating a `LinearTrajectory`, or `CircularTrajectory` object from `sawyer_full_stack/src/paths/trajectories.py` (depending on the provided task). You will be tasked to implement these trajectories.
- Control: `main.py` will execute the generated trajectory using either the default MoveIt controller or one of the controllers you will be writing in `sawyer_full_stack/src/controllers/controllers.py`.

We additionally provide the following `.launch` files:

- `sawyer_camera_track.launch`: uses the `ar_track_alvar` package to detect `ar_tags` on the Sawyer camera.

- `custom_sawyer_tuck.launch`: positions the Sawyer so the camera can view the entire table. This is optionally run at the beginning of `main.py`

While you do not need to understand all of the code, we recommend reading some of it for understanding.

1.2 Spinning up the robot files

Before running code for this lab, you will run the following:

Reminder: All programs interacting with the Sawyer robot should be run while connected to the robot!

- Start the intera action server:

```
roslaunch intera_interface joint_trajectory_action_server.py
```

- Start MoveIt to enable inverse kinematics and usage of the MoveIt controller:

```
roslaunch sawyer_moveit_config sawyer_moveit.launch electric_gripper:=true
```

- Launch the camera AR tracking node from Section 2.

2 Perception: Tracking AR tags

2.1 Sawyer Camera Setup

When executing your pick-and-place task, you manually recorded and programmed the Sawyer to move to the start position. While this worked for a very simple case, it's generally not a good strategy. Rather than have a hard-coded start position, we would like our Sawyer to dynamically detect and move to a target location.

Luckily, the Sawyer comes equipped with two cameras that allow it to observe its surroundings: a `head_camera`, located on its "head", and a `right_hand_camera`, located on its right wrist. Run the following two commands to view the output of the two cameras:

```
rosrun intera_examples camera_display.py -c head_camera
rosrun intera_examples camera_display.py -c right_hand_camera
```

Due to bandwidth issues, only one camera can be enabled at a time. In practice, we've found that the head camera will occasionally fail to recognize smaller AR tags, so we will be using the wrist camera in this lab. The wrist camera does come with a few downsides:

- Unlike the head camera, the wrist camera image feed is black and white. This isn't too much of an issue since we'll only be using it to detect AR tags.
- The wrist camera is located on the side of the wrist, its view is dependent on the robot's current position. We've provided the `custom_sawyer_tuck.launch` file to position the camera in an ideal configuration. Run this every time you need to start the tracking scripts, being aware of your surroundings :).

Like in Lab 4, we will use the `ar_track_alvar` package to track AR tags. Run

```
roslaunch sawyer_full_stack sawyer_camera_track.launch
```

to initialize AR tag tracking.

In a new terminal, run `rviz`. Set the fixed frame to `base`, add a `RobotModel` and `tf` display, and verify that the AR tag frame is being published and at a reasonable location. You may also add an `Image` display and set the topic to the corresponding camera topic to view the Sawyer's camera feed.

Ensure you are using the larger wood AR tags, which are defined as 15cm wide in the `sawyer_camera_track.launch` *file; without any depth perception capabilities, the AR tracker needs to know how large the tags are to know how far away they are.*

2.2 Transformation lookup

Task 1: In `main.py`, implement the `lookup_tag` function to, given an AR tag number, lookup the position of the AR tag relative to the robot base frame.

Hint: this is very similar to what you did in Lab 4 with the TurtleBots

3 Planning: Trajectory Tracking

Now that we've generated a target position from the AR tag, we want our robot to use this target position to plan a trajectory. You will implement 2 different types of trajectories in `paths/trajectories.py` for the Sawyer to execute:

- `LinearTrajectory`: a straight line to the goal point
- `CircularTrajectory`: a circle around the goal point parallel to the floor

Each Trajectory has two functions you will have to implement:

- `target_pose`: returns the translation and quaternion of where the end effector should be at some *time*.
- `target_velocity`: returns the velocity of the end effector at some some *time* as a twist.

3.1 Linear Trajectory

The `LinearTrajectory` constructor takes in the following arguments:

- `start_position`: the starting endpoint of the trajectory
- `goal_position`: the ending endpoint of the trajectory
- `total_time`: the total duration over which to execute this trajectory.

Task 2: Implement the `LinearTrajectory` `target_pose` and `target_velocity` functions. Since we want our trajectory to be smooth, you'll implement your trajectory to maintain a constant acceleration for the first half of the trajectory and constant deceleration for the latter half; we've provided this to you in the `self.acc` variable, as well as `self.v_max`, the maximum velocity the trajectory will reach.

3.2 Circular Trajectory

The `Circular Trajectory` constructor takes in the following arguments:

- `center_position`: the center of the circular trajectory
- `radius`: the radius of the circular trajectory
- `total_time`: same as before

Task 3: Implement the `CircularTrajectory` `target_pose` and `target_velocity` functions. Unlike the `LinearTrajectory` case, you'll be making your calculations in angular space, so we instead provide you with the constant `self.angular_acceleration` and `self.angular_v_max` variables. In the starter code, we've provided you with the logic of converting angular positions and velocities to their linear counterparts.

3.3 Testing

The `__main__` function of `trajectories.py` contains some testing code to test your trajectory implementations. Uncomment the corresponding trajectory you wish to visualize and run `python trajectories.py` to display/visualize the trajectory.

Once you've verified that your trajectories look as expected, you can execute your trajectory on the Sawyer via:

```
python main.py -task {line/circle} -ar_marker {number} --log
```

which will execute your trajectory via the default MoveIt controller.

Warning: Be especially vigilant on the E-Stop when running code with a custom trajectory. You lose many of the safety and reliability guarantees provided by the internal implementations.

Checkpoint 1

Submit a checkoff request at tinyurl.com/fa23-106alab2 for a staff member to come and check off your work. At this point you should be able to:

- Show that rviz tracks the correct AR Tag in the UI.
- Explain how you calculated the trajectory target positions and velocities.
- Show the debug line and circle trajectory visualizations
- Show that the line task works correctly to servo to the AR Tag
- Show that the circle task first goes to the start position and then revolves around the AR Tag pointing down

---

4 Control: PID

In Lab 5, we introduced the action server, which consists of:

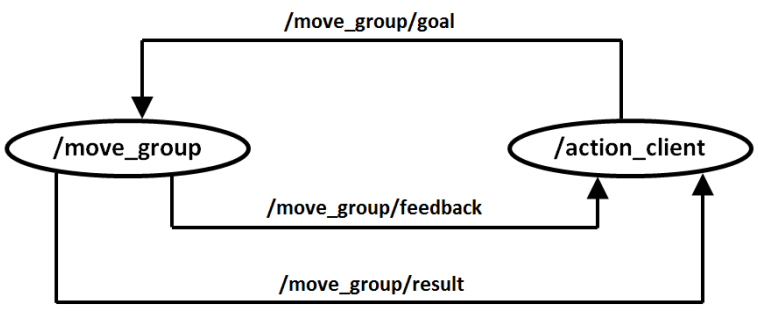

Figure 1: Action server topics

- **Goal:** Specifies the goal of the action. In our case, the goal includes the start and end states for a motion plan, as well as any constraints on the plan (like obstacles to avoid).
- **Result:** The final outcome of the action. For a motion plan, this is the trajectory returned by the path planner, as well as the actual trajectory measured from the robot's motion.
- **Feedback:** Data on the progress of the action so far. For us, this is the current position and velocity of the arm as it moves between and start and end states.

Previously, the task of executing trajectories was left to MoveIt. In this section, you'll be replacing MoveIt's default execution with a controller of your own. You will utilize the action server feedback data to implement your own PID controller, the most-used controller in the industry, to make the robot's end effector track the target trajectory.

`controllers.py` contains the following Controllers:

- `FeedforwardJointVelocityController`
- `PIDJointVelocityController`

For each controller, your task will be to implement the `step_control` function, which takes in a target position, velocity, and acceleration (in jointspace coordinates), and sets the Sawyer's limb velocities to the appropriate values determined by the controller. Note that each controller may not necessarily use all of these values!

Recall that in driving Sawyer, we can only specify an input set of joint velocities or torques.

4.1 Feedforward Control

The feed-forward (or open loop) joint velocity controller is the simplest controller. Fill in the controller to set the joint velocities.

Run the feedforward controller via:

```
python main.py -task {line/circle} -ar_marker {number} -controller open_loop --log
```

After each execution, the controller displays a plot of each joint value over time, with the target in gold and the measured value in blue. How does the performance look in the plot? Use `tf` (either `tf_echo` or do it programmatically) to check the final end effector pose against the goal. How does the open-loop controller perform?

Warning: Be especially vigilant on the E-Stop when running code with a custom controller. You lose many of the safety and reliability guarantees provided by the internal implementations. Note that hitting `Ctrl+C` should also stop any movement of the robot, but if the robot looks like it will collide with something, use the E-stop first.

4.2 What is PID Control

As you may have learned/noticed, feedforward control is unable to account and correct for errors. Luckily, with feedback from our action server feedback, we can implement closed-loop control to track and attempt to correct any errors as we execute our trajectory. The following introduction to PID control is shared between Labs 7 and 8.

PID (Proportional-Integral-Derivative control) is a type of feedback control system widely used in industrial control systems and various other applications requiring continuously modulated control. A PID controller continuously calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms. Watch the PID video from MathWorks if you want to learn more. PID (proportional integral derivative) control is ubiquitous in industry because it's both intuitive and broadly applicable.

4.2.1 Components of PID Control

1. **Proportional (P)**:
   - The proportional term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant known as K_p, the proportional gain constant.
   - Formula:
   $$P_{\text{out}} = K_p \times \text{error} \tag{1}$$
2. **Integral (I)**:
   - The integral term is concerned with the accumulation of past errors. If the error has been present for an extended period of time, it will accumulate (integral of the error), and the controller will respond by changing the control output in relation to a constant K_i known as the integral gain.
   - Formula:
   $$I_{\text{out}} = K_i \times \int \text{error}\, dt \tag{2}$$
3. **Derivative (D)**:
   - The derivative term is a prediction of future error, based on its rate of change. It provides a control output to counteract the rate of error change. The contribution of the derivative term to the overall control action is termed the derivative gain, K_d.
   - Formula:
   $$D_{\text{out}} = K_d \times \frac{d(\text{error})}{dt} \tag{3}$$

The combined feedback output from all three terms is computed as:

$$\text{Output} = P_{\text{out}} + I_{\text{out}} + D_{\text{out}} \tag{4}$$

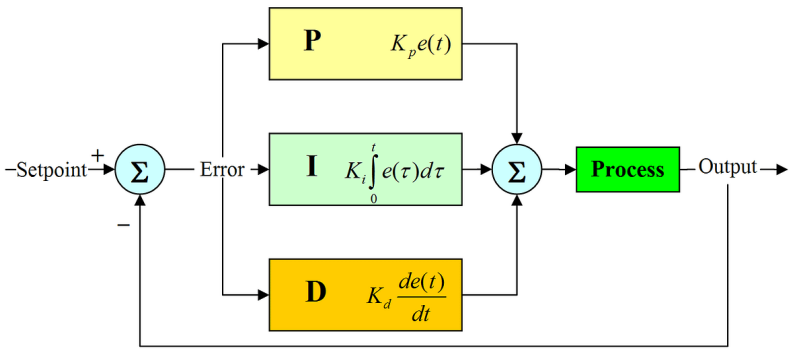

Figure 2: When will I ever use this again?

4.2.2 Advantages of PID Control

- Versatility: PID controllers can be used for a wide range of applications.
- Stability: Properly tuned PID controllers can provide stable control for many processes.
- Improved transient response: The controller can reduce the overshoot and settling time of a system.

4.2.3 Challenges

- Requires tuning: The K_p, K_i, and K_d values need to be properly set for optimal performance, which can sometimes be a complex task.
- Not suitable for all processes: Some systems might not benefit from one or more of the PID terms.

4.3 Implementing PID

These PID feedback terms can be used to stabilize our feedforward controller. Our new control law is defined as:

$$u = u_{ff} + K_p e + K_i \int_0^t e\,dt + K_d \dot{e} \tag{5}$$

where

- u_{ff}: the feedforward term (desired velocity)
- e: the state error $q_d - q$
- q_d: the target position/joint angles
- q: the current position/joint angles
- K_p: the proportional term which pulls the error towards zero
- K_d: the derivative term which damps the controller and reduces oscillation
- K_i: the integral term which compensates for constant error sources (like gravity)

Computing the integral in practice is actually quite difficult, as it would require storing and summing over the entire previously executed trajectory. Instead, we approximate this integral through an exponential moving average, where

$$\text{integral error} = \underbrace{K_w}_{<1} \cdot \text{previous integral error} + \text{current error} \tag{6}$$

The integral error term is stored in `self.integ_error`.

Once you've implemented your PID controller, test it by running:

```
python main.py -task {line/circle} -ar_marker {number} -c pid --log
```

How does this compare to the original MoveIt controller and your FeedForward controller.

Warning: Be especially vigilant on the E-Stop when running code with a custom controller. You lose many of the safety and reliability guarantees provided by the internal implementations. Note that hitting `Ctrl+C` should also stop any movement of the robot, but if the robot looks like it will collide with something, use the E-stop first.

5 ξ Around and Find Out

Now that you have a fully functioning system, you can toy around with a few parameters to see how the robot trajectories behave.

Warning: Be especially vigilant on the E-Stop

- In `main.py:get_trajectory`, try

  - changing the LinearTrajectory `goal_position` and `total_time` parameters.
  - changing the CircularTrajectory `radius` and `total_time` parameters.
- In `main.py:get_controller`,
  - We currently define a set of PID K_p, K_d, and K_i values. Try changing some of these values and see if your controller exhibits any noticeable behavior changes.
  - Try setting K_w to all 0s. What type of controller is this?

- Email Tarun (tarunamarnath@) a google maps directions link to MoMo Masalas with the subject "[106a help]"

Checkpoint 2

Submit a checkoff request at tinyurl.com/fa23-106alab2 for a staff member to come and check off your work. At this point you should be able to:

- Explain how you implemented PID control
- Execute paths using the Feedforward and PID controllers
- Discuss the performance of the open loop and PID controllers against the default controller
- Explain the purpose of the K_w term
