EECS C106B: Project 1 Part A - Trajectory Tracking with Baxter *

Due Date: Jan 31, 2022

Goal

Implement closed-loop PD control on Baxter and compare with the default controller.

The purpose of this project is to utilize topics from Chapters 2-4 of MLS to implement closed-loop control with Baxter. This project will be split into two parts:

- In Part A, you will define a few trajectories for the robot to execute. You will also implement a feedforward workspace velocity controller and a feedforward jointspace torque controller.
- In Part B, you will incorporate feedback into a jointspace velocity controller, a workspace velocity controller, and a jointspace torque controller. You will compare the performance of the trajectory execution of each of these controllers as well as the default MoveIt! controller.

Note: This document is only for Part A of this project. We will release a separate document for Part B later.

Contents

*Adapted for Spring 2022 by Jay Monga and Josephine Koe. Originally developed by Jeff Mahler, Spring 2017. Expanded and edited by Chris Correa, Valmik Prabhu, and Nandita Iyer, Spring 2019. Further expanded and edited by Amay Saxena and Tiffany Cappellari, Spring 2020. Adapted for Spring 2021 (the year of the plague) by Amay Saxena and Jay Monga.

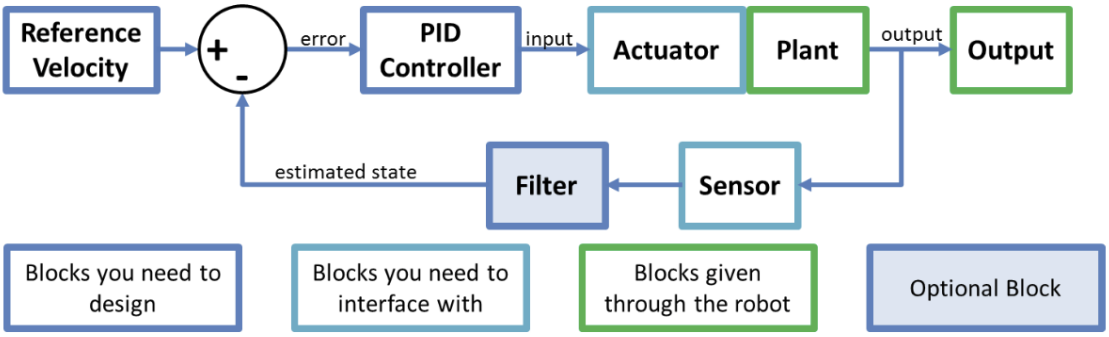

Figure 1: Block diagram of control scheme to be implemented.

1 Theory

A large part of this course is developing your ability to translate course theory into working code, so we won't be spelling out everything you need here. However, we believe that defining some terms here will help a lot in getting started with the lab as quickly as possible. Here's a definition of terms:

Workspace Control: Control wherein you look at the end effector's position, velocity, acceleration, or force in 3D space. Also called Cartesian Control.

Jointspace Control: Control wherein you look at the joints' angles, angular velocities, angular acceleration, or torque. Remember that you're still controlling the end effector position; your feedback is just on the joint states.

2 Part A Tasks

For Part A of this project you must implement various trajectories and use feedforward controllers to execute them on a simple 2-link manipulator in simulation. The tasks are formally listed below. The starter code section of this document further elaborates on the various provided files and how they all fit together.

2.1 Defining Trajectories

Specify 3 different types of trajectories that you want the robot to execute:

1. A straight line to a goal point
2. A circle in a plane parallel to the floor (constant z value) around a goal point.
3. A polygonal path in 3D space that traces straight lines between points in an arbitrary list of 3 or more goal points.

An example of each trajectory is illustrated in Figures 2, 3, and 4. Notice that the trajectories start and end with zero velocity. We recommend that you define your trajectories with this behavior for better performance on the real robots.

For Part A, keep in mind the reachable workspace of the simulated 2-link manipulator when specifying the goal points of your trajectories. The first link is of length 8, and the second link is of length 6. For Part A, you may manually specify goal points in 3D space to define your trajectories, but keep in mind that in Part B you will use AR tags to specify goal points in 3D space to define your trajectories. You can choose your own parameters such as start point, circle radius, and trajectory duration, but make sure to clearly specify the trajectory parameters that you use for Part B in your Part B report.

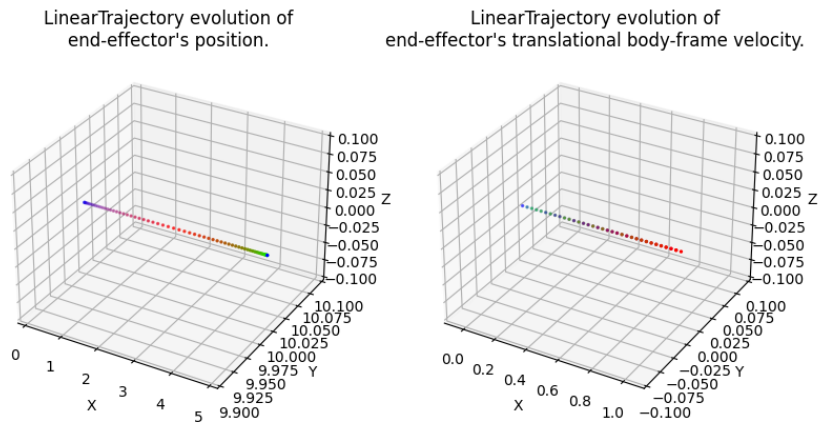

Figure 2: Example straight-line trajectory.

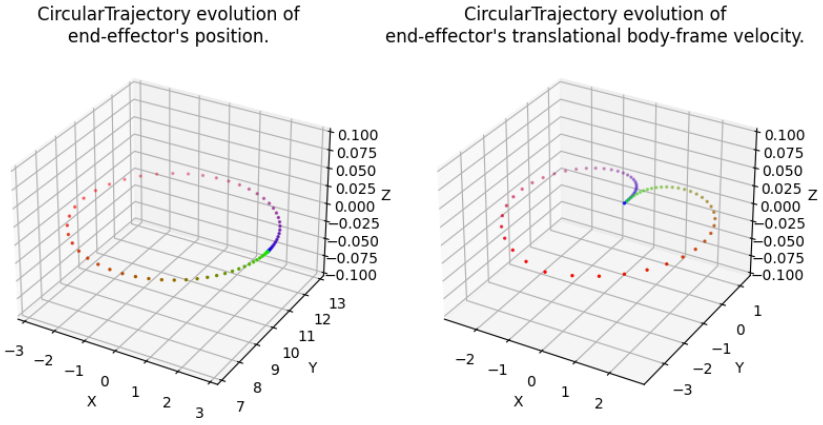

Figure 3: Example circular trajectory.

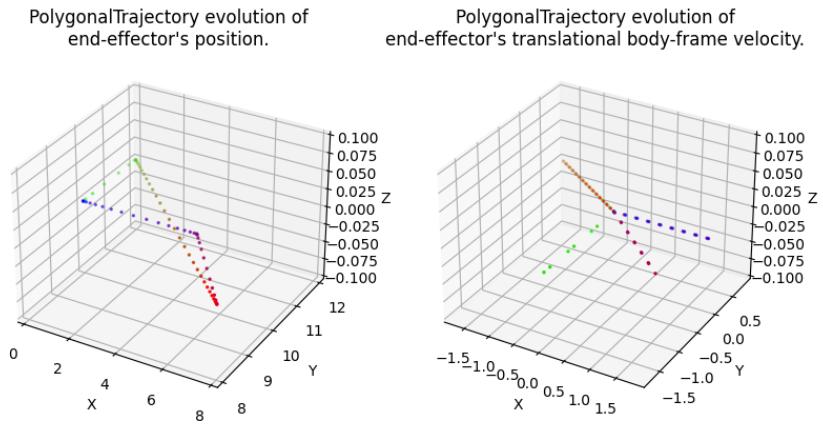

Figure 4: Example polygonal trajectory with 4 goal points.

2.2 Feedforward Controllers

Once you have defined these trajectories, you will implement two feedforward controllers: the workspace velocity controller and the jointspace torque controller.

2.2.1 Jointspace Velocity Control

This will be the simplest controller in Project 1, and the feedforward version is already implemented for you as an example. Here, the desired position, velocity, and acceleration will all be given in jointspace coordinates, as $\theta_d(t)$, $\dot{\theta}_d(t)$ and $\ddot{\theta}_d(t)$. Recall that in driving Baxter, we can only specify an input set of joint velocities or torques. Here, our control input will be a set of joint velocities. In the feedforward case this is simply the desired joint velocities, so it has already been implemented for you in the starter code.

2.2.2 Workspace Velocity Control

In workspace control, our target trajectory is no longer given as a trajectory in jointspace, but is rather given as a trajectory in workspace, $g_d(t) \in SE(3)$ that we want the end effector of the robot to track. Once again, recall that we can only supply jointspace commands to the robot. Our strategy will be to first compute a body workspace velocity that we would like our end effector to perform, $V^B(t)$. This should be the $se(3)$ velocity corresponding to the desired $SE(3)$ trajectory for the manipulator. Recall that the relationship between the end-effector workspace velocity and the corresponding joint velocity that will produce that workspace velocity is:

$$V^B = J(\theta)\dot{\theta} \tag{1}$$

where $J(\theta)$ is the body manipulator jacobian at the current joint configuration. With the desired workspace velocity $V^B(t)$, the second step is to compute our control input of joint velocities as

$$\dot{\theta}(t) = J^\dagger(\theta)V^B(t) \tag{2}$$

where $J^\dagger$ is the *Moore-Penrose Pseudoinverse* of the body jacobian. $\dot{\theta}$ is then sent to the robot as a control input.

2.2.3 Jointspace Torque Control

In jointspace torque control, our target positions, velocities and accelerations are still given in jointspace coordinates, exactly like in jointspace velocity control, but now we construct a control input of joint *torques*, instead of joint velocities. Recall from Chapter 4 Section 3.2 of MLS that the vector differential equation for the motion of a manipulator is:

$$M(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + N(\theta, \dot{\theta}) = \tau \tag{3}$$

where $M(\theta)$ is the inertia matrix, $C(\theta, \dot{\theta})$ is the Coriolis matrix, and $N(\theta, \dot{\theta})$ is the "gravity" matrix. τ is then sent to the robot as a control input.

3 Deliverables

For Part A, you will need to provide videos of all 3 of the trajectories listed above (line, circle, and polygon) being executed by the 2 controllers you implemented (workspace velocity and jointspace torque) on the simple manipulator (6 videos in total). **You will create a Google Drive folder that contains these videos and submit the folder's link to Gradescope.** You can use screen recorders such as OBS to record the simulation window.

4 Getting Started

4.1 Configuration and Workspace Setup

Remember that groups must be composed of 2-3 students, at least one of whom must have completed 106A. In 106B, projects are going to take a while and will benefit more from collaboration compared to labs in 106A. Thus, we will expect you to set up a **private** GitHub repo for each project. You can do this by forking our starter code repository.

```
git clone --bare https://github.com/ucb-ee106/proj1a_pkg.git
cd proj1a_pkg.git
git push --mirror https://github.com/yourname/private-repo-name.git
cd ..
rm -rf proj1a_pkg.git
```

Now you can clone your own repository

```
git clone https://github.com/yourname/private-repo-name.git
```

You may make any changes to your private repo, then commit and `git push origin main` as normal. If there are any updates to the starter code that you wish to pull you may do so with the commands

```
cd private-repo-name
git remote add public https://github.com/ucb-ee106/proj1a_pkg.git
git pull public main
git push origin main
```

Make sure to install the dependencies required for the project

```
cd private-repo-name
pip install -r requirements.txt --upgrade
```

4.2 Starter Code

We've provided some starter code for you to work with, but remember that it's just a suggestion. In addition, note that the project infrastructure has evolved a lot in the past few years. While we have verified that the code works, remember that some things might not work perfectly, so **do not assume the starter code to be ground truth**. Debugging hardware/software interfaces is a useful skill that you'll be using for years.

The starter code for Part A contains 4 files of interest:

- `trajectories.py` defines the trajectories. You will need to edit this file to define the trajectories.
- `controllers.py` defines the controllers. You will need to edit this file to implement the feedforward controllers.
- `SimpleArm_dynamics.py` defines the dynamics for the simple 2-link manipulator. You should not edit this file.
- `SimpleArm_sim.py` runs a trajectory and a controller on the simple manipulator. You will only need to edit the trajectory instantiations in this file.

4.2.1 The `trajectories.py` script

You can execute the script `trajectories.py` with the appropriate arguments to visualise your defined trajectories. The script will produce a visualization of the trajectory's path in terms of spatial position and translational body velocity as seen in Figures 2, 3, and 4.

Start by reading through `trajectories.py` to get an idea of how the `Trajectory` objects are structured. You will fill in the `target_pose` and `target_velocity` functions of the `LinearTrajectory`, `CircularTrajectory`, and `PolygonalTrajectory` as well as the `define_trajectories` function. The script takes two command line arguments:

- `-task` allows you to specify the kind of trajectory you wish to execute. The options are `line`, `circle`, or `polygon`.
- `--animate` allows you to animate the trajectory so you can see how it behaves over time.

4.2.2 The `controllers.py` file

The `controllers.py` file contains the implementation of the different controllers. Start by reading through `controllers.py` to get an idea of how the `Controller` objects are structured. You will fill in the `step_control` functions of the `WorkspaceVelocityController` and the `JointTorqueController` classes. We have provided `JointVelocityController` as an example.

4.2.3 The `SimpleArm_sim.py` script

You can execute the script `SimpleArm_sim.py` with the appropriate arguments to visualise a simple 2-link manipulator executing the defined trajectories and controllers. You only need to fill in the `define_trajectories` function.

The script takes two command line arguments:

- `-task` allows you to specify the kind of trajectory you wish to execute. The options are `line`, `circle`, or `polygon`.
- `-controller_name` allows you to specify which controller you want the manipulator to use. The options are `jointspace`, `workspace`, or `torque`.

You can run the `JointVelocityController` to see what the visualization should look like for the `WorkspaceVelocityController` and the `JointTorqueController`.

4.3 Notes

A couple notes:

- **When implementing your paths, you should try to respect boundary conditions.** In particular, we know that the robot must begin and end at rest, so you should ensure that your paths also begin and end with zero velocity, accelerating and decelerating as needed.
- **When simulating the jointspace torque controller, the manipulator may not trace the path as accurately as time increases.** This is due to the way the joint velocities and accelerations are computed in the simulator. It may not be as accurate as the jointspace velocity controller, but it should still clearly be trying to trace the trajectory.

5 Scoring

Table 1: Point Allocation for Project 1A

Section	Points
Trajectory Videos (Part A)	12

This project will be scored out of 12 points. Each video you submit (a total of 6) will be scored out of 2 points.

6 Submission

For Part A, you will submit a Google Drive folder link to Gradescope. This Google Drive folder should contain 6 videos, one for each combination of the 3 trajectories and 2 controllers. Please clearly name each video file with the names of trajectory and controller that are being executed. In addition, **make sure that the folder and videos are publicly viewable**. You may not receive credit if the course staff cannot view your videos.

Only one submission for Part A is needed per group, though please add your groupmates as collaborators on Gradescope.