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

Due Date: Feb 1, 2021

Goal

Implement closed-loop PD control on Baxter or Sawyer 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. The first, part A, will be to define a trajectory manually for the robot to execute. You will test these trajectories by executing them with the default controller in MoveIt!. Part B will be to define your own controllers to execute the trajectory. You'll have to implement workspace velocity, joint-space velocity, and joint-space torque control and compare the speed and accuracy of the trajectory following with each.

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 2021 (the year of the plague) by Amay Saxena and Jay Monga. 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.

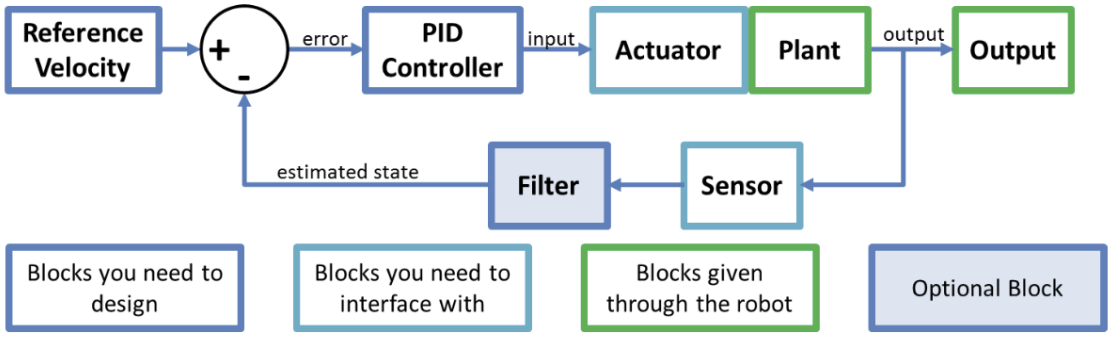

Figure 1: Block diagram of control scheme to be implemented.

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 Groups

Remember that groups must be composed of 2-3 students, at least one of whom must have completed EECS 106A.

3 Part A Tasks

For part A of this project you must implement various trajectories and show that you can get MoveIt!'s built in controllers to execute them on the robot. The tasks are formally listed below. The starter code section of this document further elaborates on the various provided files how they all fit together.

1. *Defining Trajectories*

   (a) A straight line to a goal point

   (b) A circle in a plane parallel to the floor around a goal point.

   (c) An arbitrary path in 3D space.

   The line and circle are illustrated in Figure 2. You can choose the circle radius, line length, and height above the goal points, just make these parameters clear in your report.

   Once you have defined these trajectories, execute them through MoveIt! to test the trajectories.

   In particular, for part A of this project, you will:

   (a) Implement the interface for a straight line trajectory and circle trajectory in the file `paths.py`. The "arbitrary" trajectory will be supplied by the MoveIt planner. You will supply a target end-effector configuration (as a position and a quaternion, in spatial coordinates) which will then be passed to the MoveIt planner. MoveIt will then provide you with a trajectory from the arm's current configuration to this target configuration. **For the arbitrary path, make sure that the target configuration you provide has a different end-effector orientation than the initial configuration.** Since, in part A, you will be executing these trajectories through MoveIt itself, the arbitrary trajectory should be trivial to get working. In part B, you will write custom controllers, and you will use these MoveIt-generated trajectories to show that you can track arbitrary paths in 3D space with your controllers.

(b) Fill in parts of the file `main.py` that select a trajectory to run.

(c) Record videos of each of the three types of trajectories being executed using the MoveIt controller on a Baxter robot.

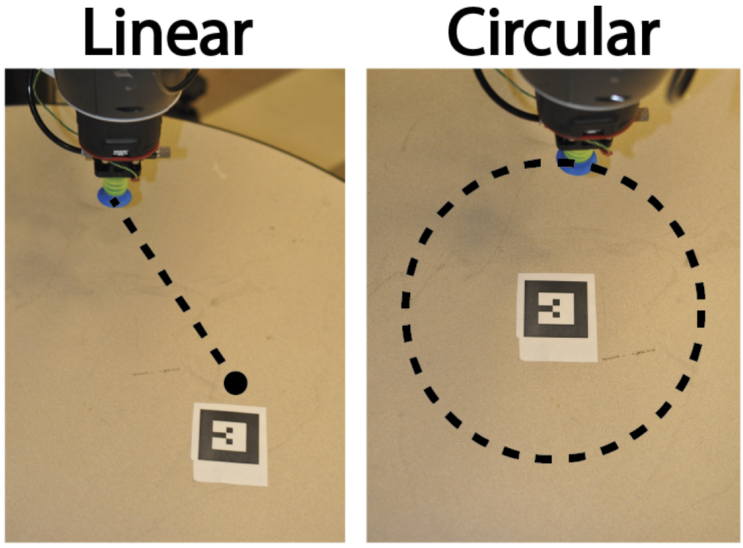

Figure 2: (Left) Example straight-line trajectory. (Right) Example circular trajectory. In the in-person version of this lab, we used AR tags to specify points in 3D space to define our trajectories. For the simulation lab, you may manually define points in 3D space for your trajectories.

4 Deliverables

For part A, you will need to provide a video of all 3 of the trajectories listed above being executed on the robot. You must upload a file with links to videos of each of the 3 trajectories working (with MoveIt).

5 Getting Started

Hopefully you have already setup ROS and the Baxter robot SDK on your personal machines in Lab 0. If not, please do so and then come back to here. A small bit of additional setup is needed for project 1.

5.1 Configuration and Workspace Setup

Note: *we expect all students to already be familiar with configuration procedures, either through EECS 106A, Lab 0, or some other place. Therefore, we won't be providing any explanations in this section. Please consult Lab 0 if you require a refresher*

To get started, first create a workspace for your project

```
mkdir -p ~/ros_workspaces/proj1_ws
cd ~/ros_workspaces/proj1_ws/src
catkin_init_workspace
```

In 106B, projects are going to take a while and will benefit more from collaboration compared to projects in 106A. Thus, we will expect you to setup a **private** github repo for each project. You can do this by forking our starter code repository.

```
cd ~/ros_workspaces/proj1_ws/src
git clone --bare https://github.com/ucb-ee106/proj1_pkg.git
```

```
cd proj1_starter.git
git push --mirror https://github.com/yourname/private-repo.git
cd ..
rm -rf proj1_pkg.git
```

Now you can clone your own repository

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

Note that no matter what you name your repo or the folder you clone into, your ROS package will always have the name `proj1_pkg`, so use this name for any ROS commands.

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
git remote add public https://github.com/ucb-ee106/proj1_pkg.git
git pull public main
git push origin main
```

5.1.1 Install dependencies

First, install some dependencies from package managers if you don't already have `pip` installed, install it

```
sudo apt-get python3-pip libnlopt-cxx-dev swig
pip3 install lxml
```

Now, we will clone two external packages using a `.rosinstall` file. A `.rosinstall` file is essentially a list of Github repos, and an easy interface to clone and pull from all of them with minimal effort.

```
cd ~/ros_workspaces/proj1_ws/src
wstool init
yes | wstool merge https://raw.githubusercontent.com/ucb-ee106/proj1_pkg/main/proj1.rosinstall
yes | wstool update
```

One package you will clone is the `baxter_pykdl` package. This is a wrapper around the OROCOS Kinematics and Dynamics Library for the Baxter robot. Note that you will be cloning a fork of this library that the course staff has adapted to work with Python 3 and ROS Noetic. The other is TRAC-IK, an inverse kinematics solver.

Finally, build all the packages we have cloned.

```
cd ~/ros_workspaces/proj1_ws
catkin_make
```

5.2 Working With Robots

For this lab, you'll be working with Baxter. To test that the simulation is working, run the following launch file:

```
roslaunch proj1_pkg baxter_gazebo_rviz.launch
```

This will launch Gazebo, enable the robot, tuck in its arms, and then launch Rviz allowing you to send commands to the Baxter through MoveIt!, all in one terminal window. Take a look at nodes being started from this launch file if you would like to run the processes in separate terminal windows.

Recall that you can use the `tuck_arms.py` script to bring the robot's arms to a convenient position from where it is easy to access most of its workspace. We recommend using this liberally.

```
roslaunch baxter_tools tuck_arms.py -u
```

5.3 Starter Code

We've provided some starter code for you to work with, but remember that it's just a suggestion. Feel free to deviate from it if you wish, or change it in any way you'd like. In fact, our staff solution features several changes in function signatures from the starter code. In addition, note that the project infrastructure has evolved a lot in the past year. 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 package contains many files, but the most important one's you should be looking at for Part A are: `main.py`, `paths.py`, and `utils.py`. The others are somewhat auxiliary, or there only if you have problems.

5.3.1 The `main` script

You will be executing the script `main.py` with the appropriate arguments to run the project. Start by reading through `main.py` to get an idea of the starter code pipeline. The script takes several command line arguments. Some of the most salient ones for part A of the project are:

- `-task` allows you to specify the kind of trajectory you wish to execute. The options are `line`, `circle` or `arbitrary`. This argument is read by the function `get_trajectory` in `main.py`. You should instantiate your paths in this function, as documented in the starter code.
- `-goal_point`. This argument allows you to pass in a float vector of length 3, which you can use to pass an (x, y, z) coordinate to the script. This is provided for your convenience, and you can use it as you please in your code (most likely, for initializing your paths in `get_trajectory`). For instance, you could use this to pass in a final location for the line trajectory, or the center point for your circle trajectory.
- `-arm` lets you specify which arm of the Baxter you wish to control. Options are `left` or `right`.
- `--moveit` is a command-line flag. When this flag is present, the built-in MoveIt controller is used to execute the trajectory. When it is absent, one of your custom controllers are used instead. For part A, you will always execute trajectories using the MoveIt controller, so this flag should always be present.

In the `main` method of this file, you also have the option of deciding which inverse kinematics solver you would like to use. We have provided the ability to use either the IK solver from the OROCOS Kinematics and Dynamics Library (KDL) exposed through the `baxter_pykdl` package (this is the default solver and is also used internally by MoveIt), or the TRAC-IK solver. You can pick between the two by setting the `ik_solver` variable. By default, the code sets this variable to None, in which case the KDL solver is used. If you instead uncomment the line that sets this variable, then TRAC-IK will be used. Two solvers are provided to you so that you can try out another IK solver if one of them is giving you undesirable performance.

5.3.2 The `paths` infrastructure

The other file that you will need to edit for part A is `paths.py`. The linear and circular paths inherit the `MotionPath` class. To specify a path, you need to implement the functions `target_position` and `target_velocity` in each of the defined classes. These functions specify the desired end-effector configuration (as an $SE(3)$ pose given in position quaternion format) and desired end-effector velocity (as an $\mathfrak{se}(3)$ body-frame velocity) at a given time t in the execution of the trajectory. You will implement the line trajectory and circle trajectory by implementing the respective classes in `paths.py`.

You will not need to implement anything for the arbitrary trajectory. Rather, you will simply pick an appropriate final configuration (as a position and quaternion), and specify it in the `get_trajectory` method of `main.py`. Then, the MoveIt motion planner will supply the trajectory.

You'll need to call the `MotionPath.to_robot_trajectory()` method on your custom trajectories to convert them into a RobotTrajectory. We have already included the code to do this and pass the trajectory to MoveIt!

The `MotionPath` interface also implements a number of visualization methods that you can use to sanity-check your implementations. You can play with these methods in the executable section of `paths.py`. Then, you can run it by directly running the python file.

5.3.3 Utility functions in `utils.py`

Finally, the file `utils.py` implements a number of functions that you might find useful in your implementations. Some of these functions you will implement yourself as part of part B of the project, but most are provided to you.

5.3.4 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.
- Make sure to always `source <path to ws>/devel/setup.bash`. You need to do this in every new terminal window. If you attempt to run a file and it says it does not exist, try sourcing.
- Gazebo often takes along time to close. The fastest way to safely quit it is to open a new terminal, run `killall gzclient && killall gzserver`, and then ctrl+c your Gazebo terminal.

5.4 Common Problems

- **IK returning None**
  1. Make sure you're passing position and orientation (in quaternion)
  2. Try using the other arm
  3. Check to make sure the robot's arm can actually reach the position you want it to go to.
  4. Try switching to a different IK solver.
  5. Add a seed to the `baxter_kinematics` IK. You can use the current position as the seed.
- **"Solution found but controller failed during execution"**
  1. With MoveIt you may just need to run it a couple of times for it to work so you may get this error a few times before the robot is able to perform the desired trajectory.

6 Scoring

Table 1: Point Allocation for Project 1

Section	Points
Trajectory Videos (Part A)	10
Video (Part B)	3
Code (Part B)	3
Methods (Part B)	10
Results: Workspace (Part B)	10
Results: Jointspace Velocity (Part B)	10
Results: Jointspace Torque (Part B)	10
Comparison with MoveIt! (Part B)	2
Application (Part B)	2
Bonus Difficulties section (Part B)	5*

See table 1 for a break-down of the various graded components of this project. Summing all this up, this project will be out of 60 points, with an additional 5 bonus points possible.

7 Submission

For Part A you will need to submit a file containing links to videos of all 3 of your trajectories working with MoveIt. For Part B you will need to submit a writeup and a link to your code (we recommend using GitHub). Your code will be checked for plagiarism, so please submit your own work. Only one submission for each part is needed per group, though please add your groupmates as collaborators.

8 Improvements

We've provided you with well over 1500 lines of starter code and comments, as well as an 8-page lab document. Three years ago, there were around 30 lines of starter code, and only 3 pages of lab. Our infrastructure is improving by leaps and bounds, but most of it has only been read by two (sleep-deprived) TAs, so we expect lots of little typos and errors. If you notice any typos or things in this document or the starter code which you think we should change, please let us know in your submissions (either in the file you submit for part A or in your writeup for part B).

8.1 Larger improvements

If you feel like being extra helpful, there are a couple things we didn't have time to incorporate into the starter code, yet might be useful to you in completing this lab. If you choose to do any of these things, let us know. We can't give you any extra credit, but we'll likely wrap your work into next year's lab.

1. Adapt our plotting code so that it works on paths executed through MoveIt!