EECS C106B: Lab 1 - Trajectory Tracking with Baxter *

Due Date: Feb 12, 2018

Goal

Implement closed-loop PD control on Baxter or Sawyer and compare with the default controller.
The purpose of this lab is to utilize topics from Chapters 2-4 of MLS to implement closed-loop control with baxter. You'll have to implement workspace, joint velocity, and joint torque control and compare the speed and accuracy of the trajectory following with each. The following diagram shows an example state feedback scheme:

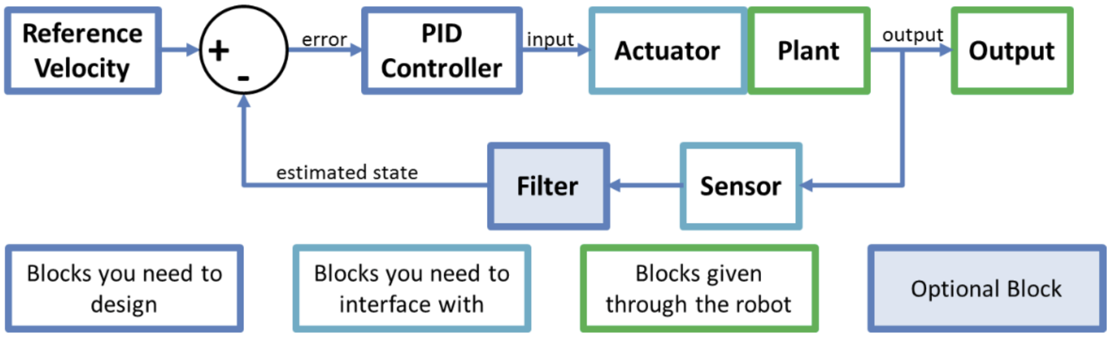

Figure 1: Block diagram of control scheme to be implemented.

Contents

*Developed by Jeff Mahler, Spring 2017. Expanded and edited by Chris Correa and Valmik Prabhu, Spring 2018

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, or acceleration in 3D space. Also called Cartesian Control.

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

Visual Servoing: Also known as vision-based robot control. You use feedback from a vision sensor (camera) to control motion.

2 Logistics and Lab Safety

We expect that all students in this class are mature and experienced at working with hardware, so we give you much more leeway than in EECS 106A. Please live up to our expectations.

2.1 Groups, Robots and Accounts

Remember that groups must be composed of 2-3 students, at least one of whom must have completed EECS 106A. Robots should be reserved on the robot calendar https://calendar.google.com/calendar/selfsched?sstoken=UUpnbjNSMlRpaF9Cf
The rules are

1. **Groups with a reservation have priority.** You can go in and use (or continue using) the hardware whenever you like if no one has a reservation. However, you must pass along the hardware once the next reservation starts. Please be respectful and try to plan ahead; it's not cool to take the first twenty minutes of another group's section winding down.

2. **Do not reserve more than two hours at a time.** This is shared hardware and we want to ensure that all groups have enough time to complete the lab.

3. **One computer per group (if needed)** We only have ten lab computers, and have around thirty students. If more than ten students want to use the computers, please try to limit yourself to one computer per group. In addition, groups with robot reservations have priority on the computers next to their respective robots. Groups can accomplish working in parallel by using personal computers, file-sharing software like git, remoting into the lab computers, and/or using simulators like Gazebo or RViz.

2.2 Safety

Remember to follow the lab safety rules:

1. **Never work on hardware alone.** Robots are potentially dangerous and very expensive, so you should always have someone on hand to help if something goes wrong. This person should be another 106B student, though in cases when group schedules are highly incompatible, you may talk to a TA to arrange an alternate solution, such as bringing a friend (note that you would be entirely responsibe for this person's conduct).

2. **Do not leave the room while the robot is running.** Running robots must be under constant observation.

3. **Always operate the robot with the E-Stop within reach.** If Baxter is going to hit a person, the wall, or a table, press the E-Stop.

4. **Power off the robot when leaving.** Unless you're trading off with another group, the robots should be powered down for safety. To power on/off the robot, press the white button on the back of the robot.

5. **Terminate all processes before logging off.** Leaving processes running can disrupt other students, is highly inconsiderate, and is difficult to debug. Instead of logging off, you should type

```
killall -u <username>
```

into your terminal, where `<username>` is your instructional account username (`ee106b-xyz`). You can also use `ps -ef | grep ros` to check your currently running processes.

6. **Do not modify robots without consulting lab staff.** Last semester we had problems with students losing gripper pieces and messing with turtlebots. This is inconsiderate and makes the TAs' lives much more difficult.

7. **Tell the course staff if you break something.** This is an instructional lab, and we expect a certain amount of wear and tear to occur. However, if we don't know something is broken, we cannot fix it.

3 Project Tasks

For this lab you must implement closed-loop workspace, joint velocity, and joint torque control. The tasks are formally listed below:

1. *Comparing Control Schemes* Follow constant-velocity trajectories using your custom PD implementations for workspace, joint velocity, and joint torque control. Optionally compare their performance to default controllers in MoveIt.
   (a) A straight line to a goal point
   (b) A circle in the table plane centered on an AR tag.
   (c) The outline of a rectangle in the talbe plane with corners marked by AR markers and a different height above the table at each point.

   The line and circle are illustrated in Figure 3. You can choose the radius, line width, and height above the squares, just make these parameters clear in your report.

2. *Visual Servoing* Implement and test a control scheme to make Baxter or Sawyer follow an AR marker with its gripper. For example, you can make the gripper always stay a constant offset away from the marker.

4 Deliverables

To demonstrate that your implementation works, deliver the following:

1. Videos of your implementation working. Provide links to your videos in the report.

2. Provide the code in a zip file, to be uploaded with your writeup. Ideally, zip your full package or workspace so that we can run your code with minimum effort if needed.

3. Submit a report with the following:
   (a) Describe your approach and methods in detail
   (b) Summarize your results in both words and figures. In particular, show:
      - Plots comparing the desired and true end effector position and velocity as a function of time for each of the control methods and trajectories (including AR marker following). Describe any differences that you see.
      - Compare the control methods in terms of total time taken to follow each trajectory.
      - **BONUS:** Compare both the speed and accuracy of your controllers with using the default controller through MoveIt!

      As a reminder, the desired trajectories are:
      - Line
      - Circle
      - Rectangle
      - AR marker following

      And the desired control methods are
      - Workspace (velocity) control
      - Joint Space velocity control

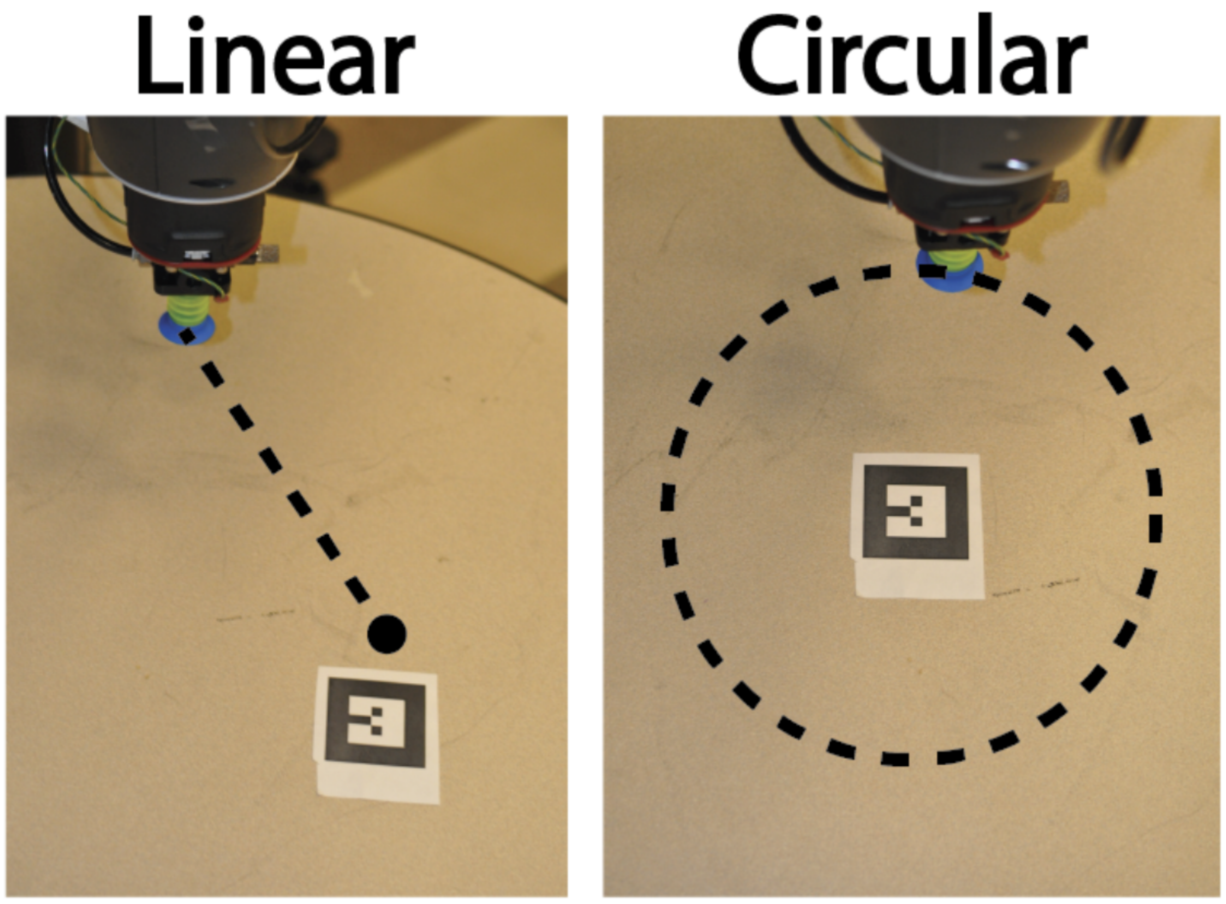

Figure 2: (Left) Example straight-line trajectory. (Right) Example circular trajectory

- Joint Space torque control

(c) For each control method, give one application where you think the method best applies.

(d) Provide any plots, tables, numbers, or videos you think would be valuable.

4. **BONUS:** In addition to your report, write a couple paragraphs on the difficulties you had performing the lab, paying particular attention to how the lab documentation could be improved. This course is evolving quickly, and we're always looking for feedback.

5 Getting Started

The lab machines you're using already have ROS and the Baxter robot SDK installed, but you'll need to perform a few user-specific configuration tasks the first time you log in with your class account, as well as setting up the package for lab 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*

First, open `.bashrc` in your home directory, and add the following three lines to the end of the file:

```
source /opt/ros/indigo/setup.bash
source /scratch/shared/baxter_ws/devel/setup.bash
export ROS_HOSTNAME=$(hostname --short).local
```

Restart your terminal, then type the following commands:

```
mkdir ~/ros_workspaces
mkdir ~/ros_workspaces/lab1_ws
mkdir ~/ros_workspaces/lab1_ws/src
cd ~/ros_workspaces/lab1_ws/src
catkin_init_workspace
git clone https://github.com/RethinkRobotics/baxter_pykdl.git
cd ..
catkin_make
source devel/setup.bash
```

Now download `Lab1_Resources.zip` from bcourses and unzip it. Put the `lab1_pkg` package into the `src` folder of `lab1_ws`, and rebuild.

5.2 Working With Robots

For this lab, you'll be working with Baxter or Sawyer. The initial setup steps are the same. First link the `baxter.sh` file to your workspace. Type

```
ln -s /scratch/shared/baxter_ws/baxter.sh ~/ros_workspaces/lab1_ws/
```

To setup your robot, run `./baxter.sh [name-of-robot].local` in your folder (where `[name-of-robot]` is either `asimov`, `ayrton`, or `archytas` for the Baxters; or `alan` or `ada` for the Sawyers). Then run `source devel/setup.bash` so the workspace is on the `$ROS_PACKAGE_PATH`.

Unfortunately, Rethink Robotics has not yet put out a Kinematics and Dynamics Library for Sawyer; therefore you will only be able to use Baxter for this lab.

To test that the robot is working, run the following commands:

1. Enable the robot

   ```
   rosrun baxter_tools enable_robot.py -e
   ```

2. Start the trajectory controller

   ```
   rosrun baxter_interface joint_trajectory_action_server.py
   ```

3. Check that MoveIt! works (if desired).

   ```
   roslaunch baxter_moveit_config move_group.launch right_electric_gripper:=true
                   right_electric_gripper:=true
   ```

   omit the last argument if the robot lacks a gripper

4. Test motion

   ```
   rosrun baxter_tools tuck_arms.py -u
   ```

For velocity and torque control, you'll likely need the Jacobian and Manipulator Inertia Matrix. For this you should use the Baxter Kinematics Dynamics Library.

```
rosrun baxter_pykdl baxter_kinematics.py
```

5.3 Object Tracking with AR Markers

Baxter can use AR markers to determine the trajectory to follow using a calibrated transformation from the AR marker to the object. To set up object tracking, first configure the Baxter cameras to be the correct resolution for AR tracking:

```
rosrun baxter_tools camera_control.py -o left_hand_camera -r 1280x800
rosrun baxter_tools camera_control.py -o right_hand_camera -r 1280x800
```

.py

/src

Then launch the AR tracking with

```
roslaunch lab1_pkg baxter_left_hand_track.launch
```

cd scr/lab1_pkg/launch

You can also run the right hand version if you prefer. The script publishes reference frames to the tf tree so you can look up the object pose with respect to any coordinate frame. The name of the reference frame in tf is `ar_marker_0` for the AR marker pictured. If the script is working and the AR marker is in view of the camera you should see some output from

```
rosrun tf tf_echo base ar_marker_0
```

5.4 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 addition, note that the lab infrastructure has evolved a lot in the past year. While we have verified that the code works, remember that some things might not work, and that debugging hardware/software interfaces is a useful skill that you'll be using for years.

5.4.1 Part A: Path Infrastructure

The linear and circular paths inherit the `MotionPath` class. They define the functions $p_d(t)$, $v_d(t)$, $a_d(t)$ (the desired postion, velocity and acceleration at timestep t). These will be used by the controllers to determine the errors in position, velocity, and acceleration: $(p(t) - p_d(t))$, $(v(t) - v_d(t))$, $(a(t) - a_d(t))$.

The `MultipleLinearPaths` object is used for the rectangle task. You may choose to implement it as its own type like `LinearPath` and `CircularPath`, and have it only take in the four AR marker positions. However, we chose to implement it by passing in an arbitrary number of path varibales, and make it choose when to switch from one path variable to the next.

You may also try making a path instance for visual servoing. However, this may not be as natural, since the goal state may change as the controller is executing the path.

5.4.2 Part B: Controller Implementation

The `PDWorkspaceVelocityController`, `PDJointVelocityController`, and `PDJointTorqueController` inherit the `Controller` class. They look at the current state of the robot and the desired state of the robot (defined by the Paths above), and output the control input required to move to the desired position. Your job is to implement the `step_path()` function, which takes in the timestep and path object and outputs the control to follow the path.

The `PDWorkspaceVelocityController` compares the robot's end effector position and desired end effector position to determine the control input. Conversely, the `PDJointVelocityController` compares the robot's joint values and desired joint values to generate the control input. Meanwhile the `PDJointTorqueController` uses the methods we explored in lecture to generate the torque to reach the desired end effector position.

You may find the `inverse_kinematics()`, `jacobian_pseudo_inverse()`, and `inertia()` methods in the `baxter_pykdl` package useful. You may also find useful the `set_joint_velocities()` and `set_joint_torques()` methods in `baxter_interface.Limb()`.

Kin

(ldub)

If you set `log` to `True`, it will plot the end effector positions and velocities, along with their target values.

5.4.3 Part C: Visual Servoing

For visual servoing, you may find it easier to create a new Controller class which overrides the `execute_path` method to include the changing position of the AR marker. You could try to create a `LinearPath` object towards the goal at every timestep, but this may be awkward. Alternatively, you could have the controller generate the target position.

5.4.4 Notes

A couple notes:

- You may have to edit the `baxter_left_hand_track.launch` file if you use a different-sized AR tag.
- Make sure to always source `<path to ws>/devel/setup.bash`.
- A big part of this lab is getting you to explore lower-level control and compare different methods intelligently. Tuning controllers takes a very long time, especially for systems this complex, so remember that your output doesn't have to be perfect if you discuss its flaws intelligently.

6 Scoring

For each task listed in Section 4, you'll be graded on a 0-5 point score as explained in Table 1:

Table 1: Grading Rubric for Lab 1

Score	Meaning
0	Did not attempt
1	Attempted but missing more than 3 deliverables
2	Complete with major error or missing two or three deliverables
3	Complete with moderate error or missing one deliverable
4	Complete with minor errors
5	All tasks completed satisfactorily with deliverables

In addition, you'll earn 5 points for submitting your code. The bonus writeup will be worth the same amount of points as one of the tasks, as will the MoveIt! task, and both will be graded on the same scoring system. See Table 2 for more info.

Table 2: Point Allocation for Lab 1

Section	Points
Video	5
Code	5
Methods	5
Results: Workspace	5
Results: Jointspace Velocity	5
Results: Jointspace Torque	5
Application	5
Bonus Comparison with MoveIt	5*
Bonus Difficulties section	5*

Summing all this up, this mini-project will be out of 35 points, with an additional 10 points possible.

7 Submission

Youll submit your writeup(s) on Gradescope and your code through bCourses as a single .zip file. Your code will be checked for plagiarism, so please submit your own work. Only one submission is needed per group, though please add your groupmates as collaborators.