EECS C106B: Lab 1 - Trajectory Tracking with Baxter *

Due Date: Feb 17, 2019

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. This lab will be split into two parts. The first 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!. The second part 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. Graduate students have the additional task of implementing workspace and joint-space impedance controllers. Undergrads may complete these tasks for extra credit. Figure 1 shows an example state feedback scheme:

Contents

*Developed by Jeff Mahler, Spring 2017. Expanded and edited by Chris Correa, Valmik Prabhu, and Nandita Iyer, Spring 2019

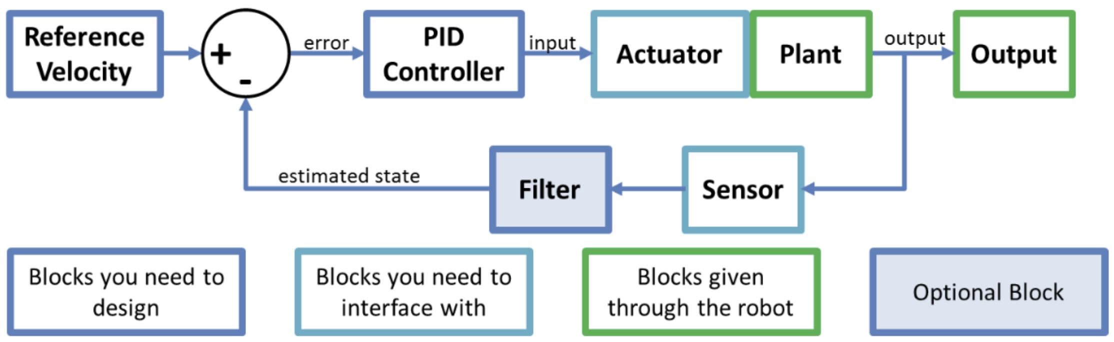

Figure 1: Block diagram of control scheme to be implemented.

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.

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

Impedance Control: Rather than controlling the system's position or force, an impedance controller regulates the impedance between force and position, velocity, and/or acceleration. Essentially it tries to make the system behave like a second-order mass-spring-damper with defined inertia, spring constant, and damping. First read part two of this paper for reference. This is the paper that originally introduced impedance control in 1984, and is worth a read. Other papers you could use for referece are: This comprehensive 1997 review, and this barebones 2017 review. Note that if you access this link on a non-campus wifi network, you'll need to use the Berkeley library proxy for access.

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 here. 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 check the trajectory when executing MoveIt! trajectories** You can view a trajectory in RViz. Sometimes MoveIt! plans extremely strange trajectories, and checking them before running them can prevent damage to the robot.

4. **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.

5. **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.

6. **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.

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

8. **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 controllers, and use MoveIt!'s controller to execute several trajectories you define. The tasks are formally listed below:

1. *Part A: Defining Trajectories*

   (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 table 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 circle radius, line length, and height above the squares, just make these parameters clear in your report.

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

2. *Part B: Comparing Control Schemes* Follow constant-velocity trajectories using your custom PD implementations for workspace velocity, joint-space velocity, and joint-space torque control. Grad students also need to implement workspace and joint-space impedance controllers. You'll also want to run these paths through MoveIt! to compare the performance of your controllers to that of Rethink's joint trajectory action server.

3. *Visual Servoing* Use each of your three (five) controllers 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. You don't have to use MoveIt! for visual servoing (and you probably shouldn't).

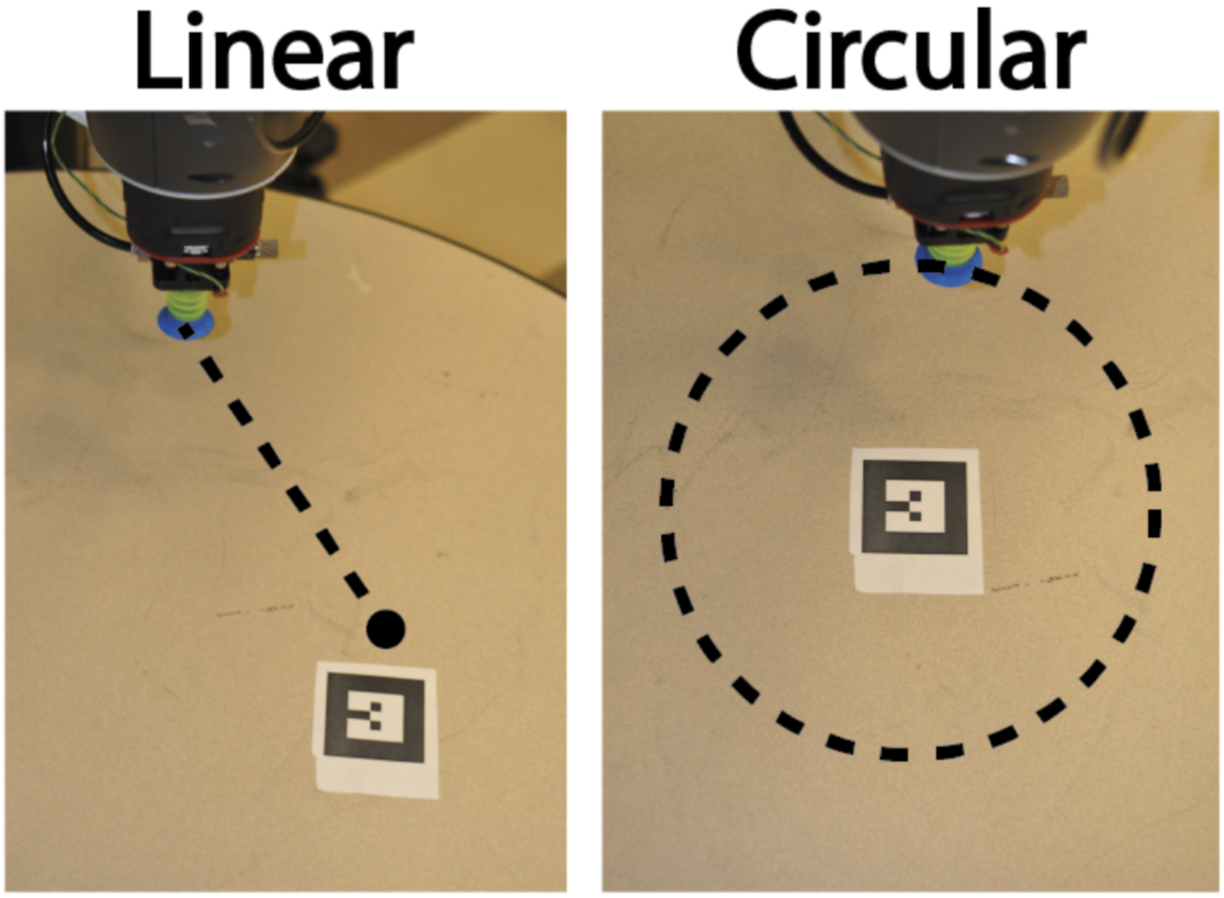

Figure 2: (Left) Example straight-line trajectory. (Right) Example circular trajectory

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. Submit a report with the following:
   (a) Provide a link to a private github repo with your code.
   (b) Describe your approach and methods in detail. Make sure to write the equations you derive for each controller, the reasoning behind your math, and a discussion of your tuning process.
   (c) 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.
      - 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:

- MoveIt! controller
- Workspace velocity control
- Joint Space velocity control
- Joint Space torque control
- Workspace impedance control (grad students)
- Joint Space impedance control (grad students)

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

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

3. **BONUS:** In addition to your report, write a couple paragraphs detailing how the lab documentation and starter code could be improved. This course is evolving quickly, and we're always looking for feedback. This should be a separate section in the report, and only well-considered, thoughtful *feedback* will merit full credit.

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
```

Re-run your bashrc with `source ~/.bashrc`, 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/valmik/baxter_pykdl.git
git clone https://github.com/chriscorrea14/lab1_pkg.git
cd lab1_pkg
python setup.py develop
cd ../..
catkin_make
source devel/setup.bash
```

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 lab1_pkg start_joint_trajectory_action_server.py
   ```

   (Remember that you can also run this directly with python)

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
```

We've modified the package to provide joint-space Coriolis and gravity vectors by exposing more of the underlying OROCOS KDL functionality. The Coriolis matrix returns a 7x1 `ndarray`, which we believe represents $C(q, \dot{q})\dot{q}$. However, we take no responsibility if this is incorrect. The gravity matrix returns a 7x1 `ndarray`.

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. First you will need the AR tracking package:

```
cd ~/ros_workspaces/lab1_ws/src/
git clone https://github.com/sniekum/ar_track_alvar.git
cd ar_track_alvar
git checkout kinetic-devel
cd ~/ros_workspaces/lab1_ws/
catkin_make
```

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
```

Then launch the AR tracking with

```
roslaunch lab1_pkg baxter_left_hand_track.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 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.

We have provided a bunch of files. The most important one's you should be looking at are: `main.py`, `paths.py`, `controllers.py` and `utils.py`. The others are somewhat auxiliary, or there only if you have problems.

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 workspace 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 variables, and make it choose when to switch from one path variable to the next.

You'll need to call the `to_robot_trajectory()` method to convert this into a RobotTrajectory. We have already included the code to pass this into MoveIt!

5.4.2 Part B: Controller Implementation

The `FeedForwardJointVelocityController`, `PDWorkspaceVelocityController`, `PDJointVelocityController`, `PDJointTorqueController`, `WorkspaceImpedanceController`, and `JointSpaceImpedanceController` 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 `FeedForwardJointVelocityController` has already been implemented for you as an example. It's the same controller you saw in Lab 7 in EECS 106A (though organized a little differently).

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. The `PDJointTorqueController` takes in the same inputs, but 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()`.

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'll probably want to fill out the `follow_ar_tag` method we've provided. When following an AR tag do you need to call `execute_path`?

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.

5.5 Common Problems

- **AR tag not recognized**
    1. Open the baxter camera in RViz. It's possible the glare causes the AR tag to be washed out. You can either turn off one of the lights in the lab or put some black paper behind the AR tag to reduce glare.
    2. Instead you can run the `tag_pub` script which constantly publishes the latest known position of the AR tag.
- **IK returning None**
    1. Make sure you're passing position and orientation (in quaternion)
    2. Try using the other arm
    3. Add a seed to the `baxter_kinematics` IK. You can use the current position as the seed.
    4. Use MoveIt!'s IK service. Check `moveit_ik.py` on how to use it.
    5. Download and use `trac_ik`. It presents several improvements over the OROCOS KDL inverse kinematics algorithm (which is used both in `baxter_pykdl` and MoveIt!), as detailed here.

6 Scoring

Table 1: Point Allocation for Lab 1

Section	**Points**
Video	3
Code	3
Methods	10
Results: Workspace	10
Results: Jointspace Velocity	10
Results: Jointspace Torque	10
Results: Workspace Impedance Control	5*
Results: Jointspace Impedance Control	5*
Comparison with MoveIt!	2
Application	2
Bonus Difficulties section	5*

Summing all this up, for undergrads this mini-project will be out of 50 points, with an additional 15 points possible. For grad students, the mini-project will be out of 60 points, with an additional 5 points possible.

7 Submission

You'll submit your writeup(s) on Gradescope and your code via git (the exact method is still being determined). You do not need to submit part A separately, just include the results from part A in the full report for Lab 1. 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.

8 Improvements

We've provided you with well over 1500 lines of starter code and comments, as well as an 8-page lab document. Two 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 here. Next years students will thank you.

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 (like we did with the `tag_pub` node from last year).

1. Verify the Coriolis matrix output in `baxter_pykdl`. The OROCOS codebase has very poor documentation, and we didn't have enough time to figure out how they were calculating the Coriolis matrix. We believe that their function returns $C(q, \dot{q})\dot{q}$, but we aren't entirely sure. Ideally, you'd also figure out if we can get the full nxn Coriolis matrix $C(q, \dot{q})$ rather than a vector. You can find the OROCOS KDL source code here.

2. Derive cartesian representations of the Coriolis and Gravity terms based on their joint-space counterparts. There's a couple hours of dynamics in here, and I don't think we'll have time to get to it.

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

4. Look at where we modified the joint trajectory action server. After executing a command, the action server keeps the robot in the same position until it recieves another command. This meant that if you tried using one of your custom controllers after running a path through MoveIt, the robot would fight with itself.

   We solved this problem by simply getting rid of the command maintaining the position, but this isn't the best solution. Ideally, there would be a subscriber listening to the joint command topic (I can't remember which one it was), and dropping the hold command if a new joint command is sent to the robot.