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

Due Date: Feb 17, 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.

Contents

*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. Further further expanded and edited by Amay Saxena and Jay Monga, Spring 2021.

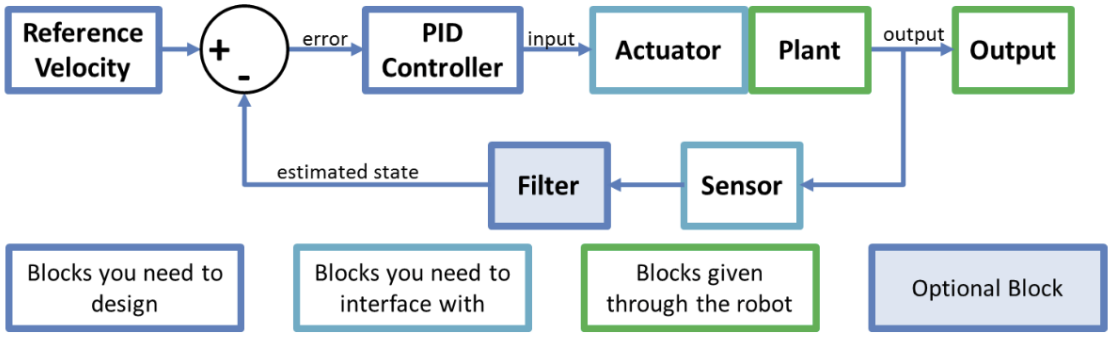

Figure 1: Block diagram of control scheme to be implemented.

1 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.

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. In this project, you will be comparing the performance of 3 types of controllers in tracking each of the trajectories you defined in part A.

For each controller, you will be given a desired position, velocity, and acceleration at each timestep, and your job is to design a control input in order to make the robot perform that particular desired behaviour. The rest of this section will briefly describe the pieces of theory you need to know to successfully complete each task of this project.

You will implement your controllers in the `controllers.py` file from the starter code. This will involve completing the implementation of the `step_control` function for each defined class. The `step_control` function will take as input a desired position, velocity, and acceleration (given in jointspace or workspace coordinates, depending on the kind of controller), and will compute a control input and send the required command to the robot.

1.1 Jointspace PD Velocity Control

This will be the simplest controller you will have to implement. Here, the desired position, velocity, and acceleration, will all be given in joint space coordinates, as $\theta_d(t)$, $\dot{\theta}_d(t)$ and $\ddot{\theta}_d(t)$. i.e. you will be looking directly at the robot's joint angles to decide on a control input. 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. We will construct this in a simple proportional-derivative fashion by considering the error $e(t) = \theta_d(t) - \theta(t)$ and its derivative. Here, $\theta(t) \in \mathbb{R}^7$ is the current vector of joint angles of Baxter's seven joints.

1.2 Jointspace PD 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. Once again, we incorporate error feedback in a PD fashion. A full treatment of jointspace torque control is given in chapter 4 of the textbook. Note the you may use either the computed torque control law from section 5.2 of chapter 4 or the PD torque control law from section 5.3.

1.2.1 Implementation notes

1. You will get the jointspace inertia matrix for the robot from the KDL library. See the respective comments in `controller.py` for more.

2. We have found that gravity vector returned by KDL is quite wrong. So, we have implemented our own package to compute the gravity vector. This is the `ee106b_baxter_kdl` package, and you should use this to access the robot's gravity terms. See the *getting started* section to correctly install this package, and the comments in `controllers.py` for exact usage.

3. We could not find a good way to access the coriolis matrix through KDL or an efficient custom implementation, so we will be ignoring the coriolis terms altogether, which is a valid assumption when $\dot{\theta}$ and $\dot{\theta}_d$ are small. In such a case, the feedback terms will be able to correct for any model mismatch introduced by the lack of the small coriolis terms. You should think about what this means for the kinds of trajectories your torque controller can successfully track (in particular, you may need to slow some of the paths down to have the torque controller successfully track them).

1.3 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-frame workspace velocity that we would like our end effector to perform, $U^b(t)$. This should be the velocity that will reduce the error between the current configuration of the end-effector $g(t)$ and the desired configuration $g_d(t)$. Once we have such a velocity, we can use the Jacobian to come up with a joint-space velocity $u \in \mathbb{R}^7$ which we will then send to the robot as a control input.

We will compose U^b by adding up two terms, a feedforward term (which is the nominal body-frame velocity of the desired trajectory) and a feedback term (which will be that body velocity which reduces the error between the robot's current configuration and the desired configuration at the current instant).

Recall that in implementing your trajectories, you returned the desired velocity in the form of the body-frame velocity of g_d. This is exactly the feedforward term, except we need to transform it into the robot's current body frame. Call this velocity V_d^b. By definition, we have $\hat{V}_d^b = g_d^{-1}\dot{g}_d$. Note that this velocity is written in the current *desired* frame. In order to convert it into a body velocity that the robot can perform, we need to transform it into the current *actual* frame of the robot.

If the current frame of the robot is g_t, then the *error configuration* is $g_{td} = g_t^{-1}g_d$. We can use the adjoint of g_{td} to convert V_d^b from the d frame to the t frame.

Next, the feedback term needs to be that velocity which reduces the error between the t and the d frames. In homework 1, you argued that the velocity that does this is exactly ξ_{td}, where

$$\hat{\xi}_{td} = \log(g_{td})$$

Finally, we will put all of this together to come up with our *Cartesian control law*

$$U^b = K_p \xi_{td} + \mathrm{Ad}_{g_{td}} V_d^b$$

where K_p is a 6x6 diagonal matrix of (positive) controller gains which you can tune.

Finally, to convert this workspace control input into a set of joint-velocities that we can send to the robot, we start by translating it into spatial coordinates as $U^s = \mathrm{Ad}_{g_t} U^b$, and then using the pseudo-inverse of the spatial Jacobian to get

$$\dot{\theta}(t) = J^\dagger(\theta) U^s(t)$$

Alternatively, you could use the body Jacobian directly, without needing to change reference frames first.

1.3.1 Jointspace to Workspace path conversion

In order to get the workspace controller to work with arbitrary moveit generated paths, we need to come up with a way to convert jointspace trajectories to workspace trajectories. You must do this by completing the implementation of the `convert_jointspace_to_workspace_trajectory` function in `utils.py`.

In this function, code has been written for you to come up with a discrete trajectory of joint-space waypoints with small time-intervals between them. You need to fill in two main lines of code: one to compute the target workspace position at each time-step (given the target jointspace position), and one to compute the target workspace velocity (as a body-frame $se(3)$ velocity, given the current and next target joint-space position). Problem 1b from homework 1 will be helpful here.

1.3.2 Implementation notes

1. K_p should be a diagonal matrix consisting of 6 tuneable controller gains. Call these $(K_x, K_y, K_z, K_{\omega_1}, K_{\omega_2}, K_{\omega_3})$. The first three of these correspond to translational error in the body-frame x, y, z directions. You can tune these differently to get different responses to errors in those three directions more-or-less independently, in a neighbourhood of the target trajectory. The next three gains however, together multiply the angular-velocity vector of the feed-back body velocity. Note that these do *not* independently control rotations about three axes. Rather, they scale the different entries of the angular velocity vector. So most likely, you want these to all be the same scalar. However, you *can* make these angular gains higher or lower than the translational gains to prioritize maintaining the correct end-effector orientation vs the correct end-effector position.

2. Note that this controller operates only on the desired configuration and velocity.

3. We once again found that KDL was returning a non-standard Jacobian (for the purposes of this class) and so we have provided a function to compute the spatial jacobian of the arm in the `ee106b_baxter_kdl` package. Usage instructions are in the comments in `controllers.py`.

4. Since we will be using the Jacobian to come up with our control input, this controller is highly sensitive to the starting joint configuration of the robot. In particular, if we start off close to certain joint-limits, the pseudo-inverse solution may try to push us toward those joint limits, preventing us from tracking the trajectory well. So, you may need to experiment with picking different starting points to give yourself ample space in the reachable joint-space to work with. We have found the tuck arms position is *not* particularly good due to the angle of the elbow, and so we have provided a script called `relax_arms.py` which puts the elbow in a more favourable positions, and allows you to pick a low-stretch or high-stretch configuration. You can examine this file and read the joint-angles we are using. If you're still having trouble, you can try new configurations by starting off in one of the relaxed positions and then using the `joint_position_keyboard.py` script from `baxter_examples` to move the arm around. You can print out the current joint angles of the robot in real time in a copy-pasteable format using the script `print_joint_positions.py` also provided by us.

5. You will also need to complete the implementation of `g_matrix_log` in `utils.py` yourself.

1.4 Plotting controller performance

You can run the `main.py` script with the option `--log` to display plots of the performance of your controller after the trajectory is done executing. You may use these plots to tune your controller, and as part of your report. If you wish to create other types of plots from the data, feel free to change the plotting code to additionally save the data being plotted to disk for you to use later. When running with one of your custom controllers, plotting is done by the `plot` function in the `Controller` class. When running with the moveit controller, the plotting is done by the provided `moveit_plot.py` script. We recommend using numpy's `savez` function to save datasets of numpy arrays to disk.

2 Deliverables

You will be expected to the deliver the following parts in your report. Our goal with this report is to prepare you to write your final project research paper and to demonstrate that your implementation works. Please format your report using the IEEE two-column conference template. Column suggestions do not include the figures.

1. Methods

   (a) Explain the theory for each of the 3 controllers (workspace velocity, jointspace velocity, jointspace torque). Make sure to write the equations you derive for each controller and the reasoning behind your math. (~1 column)

   (b) Explain the tuning procedure for each controller, optional: include 1 figure per controller to supplement explanation (~1 column)

2. Experimental Results

   (a) Explain your experimental design. What experiments did you run? (In this case, running the line, circle, random trajectories) Did you have a control group? If yes, what was the control group? (~1/2 - 1 column)

   (b) Plot the experimental results. Include plots for each experiment.

i. Your plots should come in the form of figures. Each figure will have the desired trajectory and true trajectories of the arm, and a self-explanatory caption.
ii. Each plot should all have 4 controllers (the 3 you implemented and the built-in MoveIt controller) so you can compare the performance of the controllers on these various tasks.
iii. Show the workspace performance. Include 3 figures, 1 for workspace position error, 1 for workspace velocity error, and 1 for workspace orientation error.
iv. Show the jointspace performance. Include 7 figures, 1 for each joint. (Ignore velocity error)

(c) Compare and contrast the 3 controllers and MoveIt, quantitatively and qualitatively. Consider the speed and accuracy of your controllers. Consider what kinds of task each controller performs well on, supporting your analysis with plots where applicable. (~1-2 columns). Which controllers work the best/worst, why?

3. Applications: For each control method, give one application where you think the method best applies. (~1 column)

4. Bibliography: A bibliography section citing any resources you used. This should include any resources you used from outside of this class to help you better understand the concepts needed for this project. Please use the IEEE citation format (https://pitt.libguides.com/citationhelp/ieee).

5. Appendix

(a) Future Improvements: How can the lab documentation and starter code could be improved? We are especially looking for any comments you have regarding the pedagogical success of the assignment.

(b) Github Link: Share your private github repo with us at the email ID berkeley.ee106b@gmail.com and provide a link to it in your report.

(c) Videos of your implementation working. Please edit all your videos into a single video no longer than 2 minutes in runtime (you may speed your videos as long as you report your speedup factor on screen). Provide the linkto your video in the report.

(d) Bonus: What do you think the learning goals of this assignment? How effective was this assignment at fostering those learning goals for you?

3 Getting Started

In addition to the set-up you already did for part A, here is what you need to do to get up-to-date to begin part B.

3.1 Configuration and Workspace Setup

First, pull the changes to the starter code (really sorry if you end up needing to merge some conflicts manually!! You'll only have to do this once!!) from `https://github.com/ucb-ee106/proj1_pkg.git`. If you have already added the starter code repo as a remote source for your repository, you will just be able to pull from this remote. See the part A document for instructions on this if you have not done so already.

3.1.1 Install dependencies

We need to install a couple new dependencies.

```
sudo apt-get install libeigen3-dev
pip3 install pybind11
pip3 install scipy
```

Next, we will install a package called `ee106b_baxter_kdl`, which is a custom package we wrote to compute Baxter's gravity vector.

```
cd ~/ros_workspaces/proj1_ws/src
git clone https://github.com/ucb-ee106/ee106b_kdl.git
cd ee106b_kdl
pip3 install .
```

Finally, build you workspace, activate it, and proceed.

```
cd ~/ros_workspaces/proj1_ws
catkin_make
source devel/setup.bash
```

3.2 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.

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

Additionally, as described above, we have provided you with another script that can be used to put the arms in a more relaxed position with the elbow straighter, which may prove more conducive especially for the workspace controller. It offers two poses, a "low stretch" pose and a "high stretch" pose.

```
rosrun proj1_pkg relax_arms.py low_stretch
rosrun proj1_pkg relax_arms.py high_stretch
```

Another script, `print_joint_positions` is provided to you to print the Baxter's joint angles in real time in a copy-pasteable format. You can try new configurations by starting off in one of the relaxed positions and then using the `joint_position_keyboard.py` script from `baxter_examples` to move the arm around, and record the joint angles.

3.3 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 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.

3.3.1 Changes from Part A

Most of the code is the same but the following changes have been made:

1. Some restructuring to `main.py` to allow for new code to plot the performance of the MoveIt! controller.
2. Some additional plotting and verification code was added to `paths.py`. You can use the new code to verify that your target velocity implementation is correct.
3. Some additional functions have been added to `utils.py` and some of the other utils functions have been altered slightly.
4. Some comments in `controllers.py` were changed and some new code was added to do correct quaternion interpolation.
5. New scripts were added: `moveit_plot`, `relax_arms`, and `print_joint_positions`.

3.3.2 Controller Implementation

The `FeedForwardJointVelocityController`, `WorkspaceVelocityController`, `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_control` function. 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 `WorkspaceVelocityController` 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.

If you run the `main` script with the `--log` option, it will plot the performance of the controller, plotting desired and true joint angles and end effector configurations.

3.3.3 Notes

A couple notes:

- 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.

4 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)	15
Workspace Plots (Part B)	10
Jointspace Plots (Part B)	10
Analysis of Results (Part B)	10
Applications (Part B)	4
Bonus: Learning Goals (Part B)	5*

Summing all this up, this mini-project will be out of 65 points, with an additional 5 points possible.

5 Submission

You'll submit your report on Gradescope and your code via git. Please put your code in a *private* repository, and share it with the Github account `berkeley-ee106b`. You may also share it via email with the address `berkeley.ee106b@gmail.com`. 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 on Gradescope.

6 Improvements

We've provided you with well over 1500 lines of starter code and comments, as well as an 8-page lab document. 3 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 we still 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.

6.1 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. Take a look at the source code for the `ee106b_baxter_kdl` package. The way this was generated was by computing the kinematic/dynamics matrices symbolically in a MATLAB script by parsing the Baxter URDF, and then generating C code for these symbolic expressions. This C code was then (after some small syntax changes executed by the script `codegen.py`) compiled into a C++ library, and then python bindings for that library were generated using `pybind11`. We were unable to get the Coriolis matrix to run efficiently, and hence we are asking you to try your hand at computing the Coriolis matrix for the Baxter robot by parsing the URDF. The naive approach we took of trying to compute the matrix symbolically in MATLAB may not be the best approach, and it may be the case that some other numerical algorithms need to be used.

2. 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.