

EECS C106B: Project 1 Part B - Trajectory Tracking *

Due Date: Feb 14th, 2023

Goal

Implement closed-loop PD control on 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 Sawyer. This project will be split into two parts:

- In Part A, you defined a few trajectories for the robot to execute. You also implemented 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. **Graduate students have the additional task of implementing a visual servoing task.** Undergrads may complete this task for extra credit.

Contents

1	Part B Tasks	2
1.1	Trajectories	2
1.2	Controllers	2
1.2.1	Jointspace PD Velocity Control	2
1.2.2	Jointspace PD Torque Control	3
1.2.3	Workspace Velocity Control	3
1.2.4	Plotting controller performance	4
1.3	Visual Servoing (Grad Students)	5
2	Deliverables	5
3	Getting Started	6
3.1	Project Teams	6
3.2	Robot Reservations and Safety	6
3.3	Pulling starter code	6
3.4	Sawyer Setup	6
3.5	Kinematics and Dynamics Library	7
3.6	Object Tracking with AR Markers	7
3.7	Starter Code	7
3.7.1	Controller Implementation	7
3.7.2	Visual Servoing	8
3.7.3	Notes	8
3.8	Common Problems	8
4	Scoring	8
5	Submission	8
6	Improvements	9
6.1	Larger improvements	9

*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 2023 by Han Nguyen

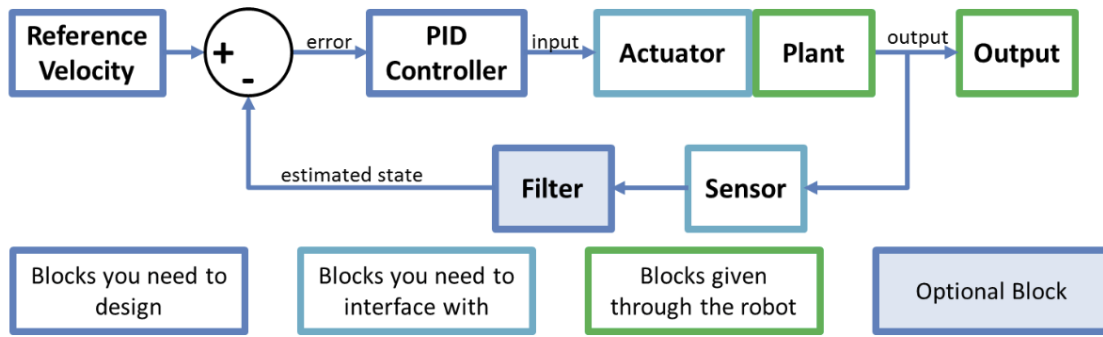


Figure 1: Block diagram of control scheme to be implemented.

1 Part B Tasks

For Part B of this project you must implement closed-loop joint velocity, workspace velocity, and joint torque controllers. You will compare their performance with MoveIt!'s controller while executing the trajectories you defined in Part A on a robot. Note that a large part of this course is developing your ability to translate course theory into working code, so we won't be spelling out all of the theory here.

The project 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.

1.1 Trajectories

In Part A, you specified 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.

In Part B, you will augment your Part A code to make the robot execute the following trajectories:

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

You can choose your own parameters such as the height to hover above the AR tags (we recommend somewhere around 0.2 m - 0.3 m), start point, circle radius, and trajectory duration, but make sure to clearly specify the trajectory parameters that you use for Part B in your report.

Make sure you can execute the trajectories with MoveIt! before attempting to run them with the feedback controllers you will implement.

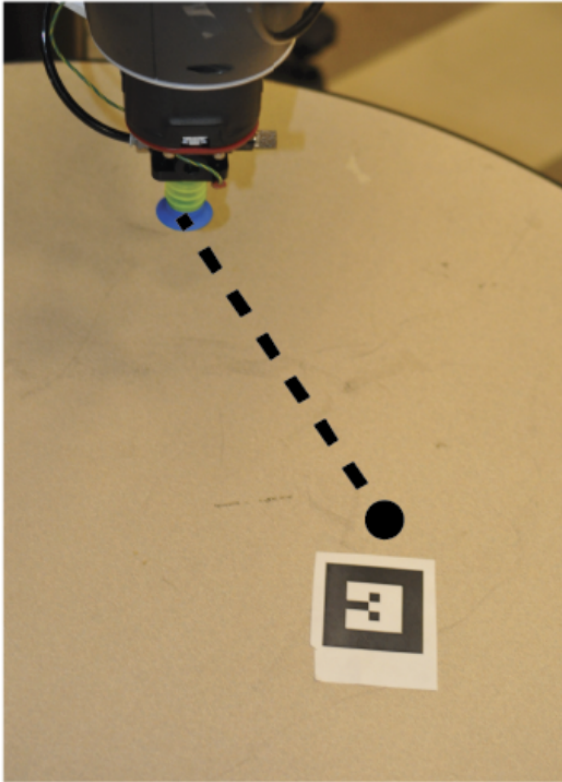
1.2 Controllers

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.

1.2.1 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 $\ddot{\theta}_d(t)$. You will also be looking directly at the current vector of joint angles of Sawyer's seven joints $\theta(t) \in \mathbb{R}^7$ to decide on a control input. Recall that in driving Sawyer, 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 Sawyer's seven joints.

Linear



Circular

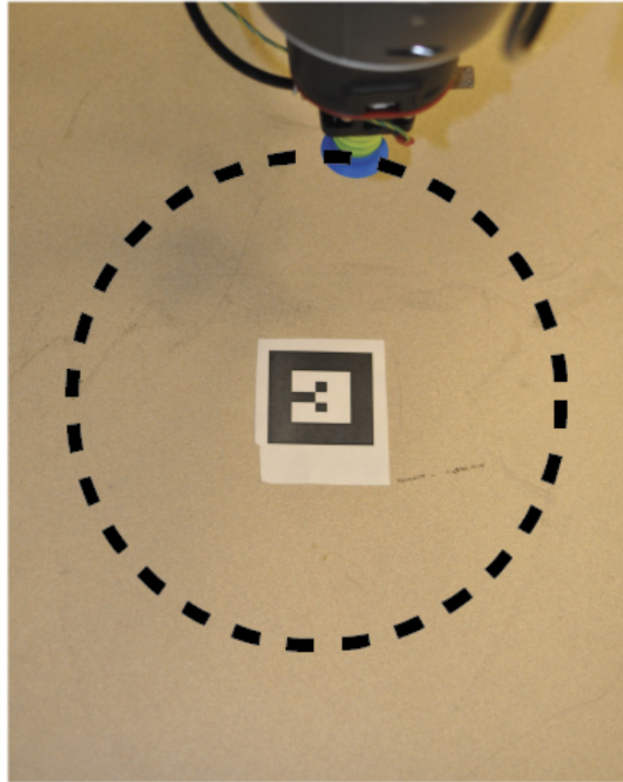


Figure 2: (Left) Example straight-line trajectory. (Right) Example circular trajectory.

1.2.2 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 that 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.

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 `baxter_pykdl` package (which contains support for Baxter and Sawyer), and you should use this to access the robot's inertia, gravity, and coriolis terms. See the *getting started* section to correctly install this package, and the comments in `controllers.py` for exact usage.

1.2.3 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. 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*

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

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 $U^s = \text{Ad}_{g_t} U^b$, 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.

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. 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.

1.2.4 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. Note that the plotting capabilities are not available for running trajectories with MoveIt! and will only work for your feedback controllers.

1.3 Visual Servoing (Grad Students)

Visual servoing, also known as vision-based robot control, is when you use feedback from a vision sensor (camera) to control motion. Use one of your three feedback controllers to make Sawyer follow an AR marker with its end-effector. For example, you can make the end-effector always stay a constant offset away from the marker.

Undergraduate teams may complete this task for extra credit.

2 Deliverables

You will be expected to 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 and state your final gain values, optional: include 1 figure per controller to supplement explanation (~1 column)

2. Experimental Results

For each control method:

- (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 with self-explanatory captions for each experiment. Your plots should compare the desired and true end-effector position and velocity as a function of time for each of the trajectories (including visual servoing). Describe and explain any differences that you see.
 - (c) Compare and contrast the controller with MoveIt!. Consider the speed and accuracy of the controller. Consider what kinds of tasks each controller performs well on, supporting your analysis with plots where applicable. (~1-2 columns). Which controllers work the best/worst, why?
 - (d) Provide any other plots, tables, numbers, or videos you think would be valuable.
3. (Grad Students) Show plots with captions comparing the desired and true end effector position and velocity as a function of time for visual servoing with the controller of your choice. Why did you choose to use this controller? How well did it perform? Describe and explain the behavior that is occurring in the plots.
4. Applications: For each control method, give one application where you think the method best applies. (~1 column)
5. 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>).
- #### 6. Appendix
- (a) GitHub Link: Provide the link to your GitHub repo. We will be able to see any changes you push to your assignment repository.
 - (b) Videos of your implementation working on the robot for each trajectory-controller combination (including MoveIt!). You should have at least 12 videos from the 3 trajectories and 4 controllers. Grad students should also include a video of visual servoing. 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 link to your video in the report.
 - (c) Bonus: What do you think the learning goals of this assignment? How effective was this assignment at fostering those learning goals for you? How can the lab documentation and starter code be improved for the future? We are especially looking for any comments you have regarding the pedagogical success of the assignment. 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.

3 Getting Started

The lab machines you're using already have ROS and the Sawyer 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 Project 1.

3.1 Project Teams

Create a private project repository for your team and come up with a name for your team. Once you have formed your team, fill out [this form](#) so we can assign your team to a robot.

3.2 Robot Reservations and Safety

Make sure that you have received full point on the Robot Usage Quiz. Anything found using hardware without a full score is subject to an immediate fail for this project. We take the safety of you and your partners very seriously and want to ensure that everyone knows how to properly operate the robots.

Robots should be reserved on the robot calendars. Make sure that you do not have more than two hours reserved at a time: [Sawyer Robot Calendar](#)

3.3 Pulling starter code

Remember that groups must be composed of 2-3 students, at least one of whom must have completed EECS106A. In EECS106B, projects are going to take a while and will benefit more from collaboration compared to labs in EECS106A. Thus, we will expect you to set up a **private** GitHub repo for each project. As mentioned in project 0, you can find the starter code for this project in the [starter repository](#).

```
git clone https://github.com/ucb-ee106/106b-sp23-project-starter
*Drag out files that you need*
```

3.4 Sawyer Setup

To set up your environment for interacting with Sawyer, make a shortcut (symbolic link) in the root of your catkin workspace to the Sawyer environment script `/opt/ros/eecsbot_ws/intera.sh` using the command

```
ln -s /opt/ros/eecsbot_ws/intera.sh [path-to-workspace]
```

From the root of the catkin workspace, use the following line to connect to one of the Sawyer robots:

```
./intera.sh [name-of-robot].local
```

where `[name-of-robot]` is either `azula`, `alice`, `amir`, `ada`, or `alan`.

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

1. Enable the robot

```
roslaunch intera_interface enable_robot.py -e
```

2. Test motion

```
roslaunch intera_examples sawyer_tuck.launch
```

3. Start the trajectory controller (Required for MoveIt!)

```
roslaunch intera_interface joint_trajectory_action_server.py
```

4. Check that MoveIt! works

```
roslaunch sawyer_moveit_config sawyer_moveit.launch electric_gripper:=[true or false]
```

3.5 Kinematics and Dynamics Library

For velocity and torque control, you'll likely need the Jacobian and Manipulator Inertia Matrix. There is an existing [Baxter Kinematics Dynamics Library](#), but we've modified the package to provide joint-space Coriolis and gravity vectors by exposing more of the underlying OROCOS KDL functionality as well as support for Sawyer. These matrices returned by the kdl are slightly incorrect and may need to be tuned, for which instructions are in `controllers.py`.

This package is already included in your starter code.

3.6 Object Tracking with AR Markers

Create a folder called `camera_info` and put the file `head_camera.yaml` into this newly created folder. Now make sure you're in the directory that contains `camera_info` and run the following command to move this folder into a ROS subdirectory so that the camera can be set up correctly.

```
mv camera_info ~/.ros
```

First you will need the AR tracking package:

```
git clone https://github.com/machinekoder/ar_track_alvar.git -b noetic-devel
```

The Sawyers conveniently have Logitech cameras mounted to them. Launch the AR tracking with

```
roslaunch proj1_pkg sawyer_webcam_track.launch
```

The AR tracking launch files publish 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 AR marker 0. If the script is working and the AR marker is in view of the camera you should see some output from

```
roslaunch tf_echo base ar_marker_0
```

3.7 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 two 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.

We have provided a bunch of files. You can copy in your completed `trajectories.py` file from Part A into the `trajectories.py` file here, and the most important files 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.

3.7.1 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 and output the control input required to move to the desired position. Your job is to implement the `step_control()` functions, which take 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 `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.

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 `intera_interface.Limb()`.

Run the program using `main.py`. If you set the `--log` flag, it will plot the end effector positions and velocities, along with their target values.

3.7.2 Visual Servoing

For visual servoing, you'll probably want to fill out the `follow_ar_tag` method we've provided in `controller.py`. When following an AR tag do you need to call `execute_path`?

Run the program using `follow_ar.py`.

3.7.3 Notes

A couple notes:

- You may have to edit the AR tracking launch files if you use a different-sized AR tag.
- Make sure to always `source 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.

3.8 Common Problems

- **AR tag not recognized**

1. Open the 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.py` 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. Add a seed to the `sawyer_kinematics` IK. You can use the current position as the seed.
3. 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](#).

4 Scoring

Table 1: Point Allocation for Project 1

Section	Points
Trajectory Videos (Part A)	10
Video (Part B)	4
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
Results: Visual Servoing (Part B)	10*
Application (Part B)	3
Bonus Difficulties section (Part B)	5*

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

5 Submission

You will submit your report on Gradescope. 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

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 (like we did with the `tag_pub.py` file from a previous 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. Adapt our plotting code so that it works on paths executed through MoveIt!