EECS C106A: Remote Lab 5 - Inverse Kinematics and Path Planning *

Fall 2020

Goals

By the end of this lab you should be able to:

- Use MoveIt to compute inverse kinematics solutions and plan paths with the ROS action server for Baxter
- Create a visualization of Baxter's kinematic structure, as defined in the URDF file
- Use MoveIt to move Baxter's gripper(s) to a specified pose in the world frame and perform a rudimentary pick and place task.
- Plan and execute paths with obstacles and orientation constraints on Baxter.
- Understand the difference between open-loop and closed-loop control.
- Implement a trajectory-tracking controller on Baxter.

If you get stuck at any point in the lab you may submit a help request during your lab section at https://tinyurl.com/106alabs20.

Note: For all labs this semester you may collaborate with a lab partner but we expect everyone to do every part of the labs themselves. You should work closely with your partner to overcome obstacles in the labs but each member of the team must do the lab themselves.

Contents

*Developed by Amay Saxena and Tiffany Cappellari, Fall 2020 (the year of the plague).
Based on previous labs developed by Aaron Bestick and Austin Buchan, Fall 2014. Modified by Laura Hallock and David Fridovich-Keil, Fall 2017. Modified by Valmik Prabhu, Nandita Iyer, Ravi Pandya, and Phillip Wu, Fall 2018.

Introduction

In Lab 3, you investigated the *forward kinematics* problem, in which the joint angles of a manipulator are specified and the coordinate transformations between frames attached to different links of the manipulator are computed. Often, we're interested in the *inverse* of this problem: Find the combination of joint angles that will position a link in the manipulator at a desired location in $SE(3)$.

It's easy to see situations in which the solution to this problem would be useful. Consider a pick-and-place task in which we'd like to pick up an object. We know the position of the object in the stationary world frame, but we need the joint angles that will move the gripper at the end of the manipulator arm to this position. The *inverse kinematics* problem answers this question.

The MoveIt package also includes a variety of powerful path planning functionality — in fact, to move between your specified end effector positions, MoveIt was using this functionality behind the scenes. In this lab, you'll get acquainted with path planning on a Baxter robot.

This lab has three parts. In Part 1, you'll learn how to use ROS's built in functionality to compute inverse kinematics solutions for a robot as well as what a URDF is. In Part 2, you'll use inverse kinematics to program Baxter to perform a simple manipulation task with constraints. Finally, in Part 3 you will learn about open loop and closed loop controllers.

1 Getting started with Git

Our starter code is on Git for you to clone and so that you can easily access any updates we make to the starter code. It can be found at https://github.com/ucb-ee106/lab5_starter.git. First, create a new ROS workspace called `lab5`.

```
mkdir -p ~/ros_workspaces/lab5/src
cd ~/ros_workspaces/lab5/src
catkin_init_workspace

cd ~/ros_workspaces/lab5
catkin_make
```

Next clone our starter code by running

```
git clone https://github.com/ucb-ee106/lab5_starter.git
```

wherever you would like and then you are free to move the files into your lab's workspace into the appropriate subdirectories. We highly recommend you make a **private** GitHub repository for each of your labs just in case.

2 Installing Baxter MoveIt Configuration

In this lab, we will use a package called MoveIt, which provides an inverse kinematics solver and other motion planning functionality. You should have already installed the bulk of MoveIt in Lab 0, however, we now also need to install another package called `baxter_moveit_config` that will allow us to use MoveIt to solve IK problems as well. First open up a terminal and run

```
git clone https://github.com/ros-planning/moveit_robots.git
```

Now, we only care about the Baxter configuration files so we can go ahead and delete the other robot packages.

```
rm -rf moveit_robots/atlas_moveit_config
rm -rf moveit_robots/atlas_v3_moveit_config
rm -rf moveit_robots/iri_wam_moveit_config
rm -rf moveit_robots/r2_moveit_generated
```

Note: You can actually do the above commands all in one line by just separating the different directories you want to remove with spaces. We split it up here for you since it would be too long to render in a single line in the PDF and, depending on your PDF viewer, make copying and pasting weird.

Finally, move the remaining files that we want into your `rethink_ws` and build and source your workspace

```
mv moveit_robots ~/rethink_ws/src
cd ~/rethink_ws
catkin_make
source devel/setup.bash
```

3 Inverse Kinematics

An inverse kinematics solver for a given manipulator takes the desired end effector configuration as input and returns a set of joint angles that will place the arm at this position. In this section, you'll learn how to use ROS's built-in inverse kinematics functionality.

3.1 Specify a robot with a URDF

While the `tf2` package is the de facto standard for computing coordinate transforms for forward kinematics computations in ROS, we have several options to choose from for inverse kinematics.

Both MoveIt and `tf2` are generic software packages that can work with almost any robot. This means we need some method by which to specify a kinematic model of a given robot. The standard ROS file type for kinematic descriptions of robots is the Universal Robot Description Format (URDF). The URDF file for Baxter is contained in the `baxter_description` package. (Note: The entire Baxter SDK, including the `baxter_description` package, should be in your `~/rethink_ws`. To find the URDF, run `roscd baxter_description`). Also, note that `baxter.urdf` is actually deprecated: modern versions of ROS use the xacro package to describe the same information more cleanly. We will not examine these xacro files in this lab, but we encourage you to look through them if you plan to write your own URDF files for your final project. Because `baxter.urdf` is deprecated, it no longer exists in the Baxter SDK packages you installed so we have provided it in `lab5_starter`.

Task 1: It's hard to visualize the actual robot by staring at an XML file, so ROS provides a tool that creates a more informative kinematic diagram. Navigate to the folder containing your copy of the Baxter URDF (that we provided with the starter code) and run

```
check_urdf baxter.urdf
urdf_to_graphiz baxter.urdf
```

Open the PDF file that this command creates. Some of the nodes in the diagram should look familiar from Lab 3, but some are new. What do you think the blue and black nodes represent? In addition to Baxter's limbs, what are some of the other nodes included in the URDF, and how might this information be useful if you want to use Baxter's sensing capabilities?

3.2 Compute inverse kinematics solutions

To use MoveIt, you must first start the `move_group` node, which offers a service that computes IK solutions. The file `move_group.launch`, in the provided starter code, loads the URDF description of Baxter onto the ROS parameter server, then starts the `move_group` node.

To set up your environment, make a shortcut (symbolic link) to the Baxter environment script `~/rethink_ws/baxter.sh` using the command

```
ln -s ~/rethink_ws/baxter.sh ~/ros_workspaces/lab5/baxter.sh
```

from the root of your Lab 5 workspace.

Connect to the Baxter simulation by running `./baxter.sh sim` as you did in Lab 3. and start up the world simulation by running

```
roslaunch baxter_gazebo baxter_world.launch
```

and then enable the robot by running

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

Next, echo the `tf` transform between the robot's base and end effector frames by running

```
rosrun tf tf_echo base [gripper]
```

where `[gripper]` is `left_gripper` or `right_gripper` since we are working with Baxter.

Now create a new package called `ik` with dependencies on `rospy`, `moveit_msgs`, and `geometry_msgs`. Move `move_group.launch` into the new package's `/launch` subdirectory (you may have to create this subdirectory) and save `baxter.urdf` in the package's `src` subdirectory. Then run the launch file with `roslaunch` to start the MoveIt node. MoveIt is now ready to compute IK solutions.

Task 2: Edit `service_query.py` to be a node that prompts the user to input (x, y, z) coordinates for an end effector configuration, then constructs a `GetPositionIK` request, submits the request to the `compute_ik` service offered by the `move_group` node, and prints the vector of joint angles returned to the console. A couple of tips:

1. The `GetPositionIK` service takes as input a message of type `PositionIKRequest`, which is complicated. The most important part is the `pose_stamped` field, which specifies the desired configuration of the end effector (see our `service_query.py` for an example on how to use it).
2. The orientation of the end effector is specified as a quaternion. Since we're not worried about rotation, you can set the four orientation parameters to any values such that their norm is equal to 1. For reference, a value of $(0.0, \pm 1.0, 0.0, 0.0)$ will have Baxter's grippers pointing straight down.
3. Baxter robots have both a left and right arm, and the gripper frame is called either `left_gripper` or `right_gripper` depending on which arm you use.

Task 3: Now lets test your IK solutions from MoveIt using your FK node from Lab 3. Copy over the necessary files you need to run your `forward_kinematics.py` node from that lab (including the node's file itself) and edit the node so that it now takes in your own inputs rather than subscribing to the robot. Run it with the input being the solution of your IK node for some chosen end effector position and verify that the numbers match.

Does the IK solver always give the same output when you specify the same end effector position? If not, why not? Any ideas why the solver sometimes fails to find a solution? Again, keep in mind which arm you are using on the Baxter. Your Lab 3 code was made specifically for the left arm.

Checkpoint 1

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain what a URDF is
- Validate the output of the MoveIt IK service by solving IK for different poses, plugging these returned joint angles back into your Lab 3 forward kinematics function, and verifying that you get the original pose back
- Explain whether IK solutions are unique or not and why

---

4 Planning with Baxter

In this section, you'll use inverse kinematics to program Baxter to perform a simple manipulation task of picking up a small object and placing it elsewhere.

MoveIt's path planning functions are accessible via ROS topics and messages, and a convenient RViz GUI is provided as well. In this section, you'll learn how use MoveIt's planning features via both of these interfaces.

4.1 Using the MoveIt GUI

In your workspace create a package named `planning` inside the `src` directory of your `lab5` workspace. It should depend on `rospy`, `roscpp`, `std_msgs`, `moveit_msgs`, `geometry_msgs`, `tf2_ros`, `baxter_tools`, and `intera_interface`.

Create a folder in the new package called `launch` and copy the `baxter_moveit_gui_noexec.launch` file from `lab5_starter` into the new folder. Now move the file `spawn_table_and_cube.py.py` into `planning/src`.

Also move the `models` folder into your `planning` directory as well (just leave it in the root of the package). Don't forget to build and source your workspace and make sure python files are executables!

4.1.1 Basic planning

Now ssh into the Baxter robot and use `roslaunch` to run `baxter_moveit_gui_noexec.launch`. The MoveIt GUI should appear with a model of the Baxter robot. In the Displays menu, look under "MotionPlanning" $\Longrightarrow$ "Planning Request." Under "Planning Request" can check the "Query Start State" and "Query Goal State" boxes to show the specified start and end states. You can now set the start and goal states for the robot's motion by dragging the handles attached to each end effector. When you've specified the desired states, switch to the "Planning" tab, and click "Plan." The planner will compute a motion plan, then display the plan as an animation in the window on the right. In the Displays menu, under "Motion Planning" $\Longrightarrow$ "Planned Path." If you select the "Show Trail" option, the complete path of the arm will be displayed. The "Loop Animation" option might also be useful for visualizing the robot's motion. Note that execution will not work (for now), because execution has been disabled in the launch file. Later in the lab, when we're working on the robot, you'll be able to execute paths through the GUI as well.

4.2 Using the action server interface

The MoveIt GUI provides a nice visualization of the solutions computed by the planner, but in a real world system, the start and end states would likely be generated by another ROS node, rather than by dragging the robot's arms in a graphical environment. The GUI is just a front end for MoveIt's actual planning functionality, which is accessible via ROS topics and messages. MoveIt's ROS interface allows you to define environments, plan trajectories, and execute those trajectories on a real robot, all by using the same `move_group` node you used in the previous section.

A complicated task like planning and executing a trajectory would be difficult to coordinate using simple topics and services. Therefore, the interface to the `move_group` node's planning functionality uses a third type of communication within ROS known as an *action server*. The ROS website provides the following description:

> In any large ROS-based system, there are cases when someone would like to send a request to a node to perform some task and also receive a reply to the request. This can currently be achieved via ROS services.
>
> In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The `actionlib` package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

As the description states, the `move_group` action server interface allows us to plan and execute trajectories, as well as monitor the progress of a trajectory's execution and even stop the trajectory before it completes.

An action server and client exchange three types of messages as a request progresses:

- **Goal:** Specifies the goal of the action. In our case, the goal includes the start and end states for a motion plan, as well as any constraints on the plan (like obstacles to avoid).
- **Result:** The final outcome of the action. For a motion plan, this is the trajectory returned by the path planner, as well as the actual trajectory measured from the robot's motion.

- **Feedback:** Data on the progress of the action so far. For us, this is the current position and velocity of the arm as it moves between and start and end states.

These messages are exchanged over several topics, as shown in Figure 1. You can use `rostopic echo` to view the topics as you would with normal ROS topics.

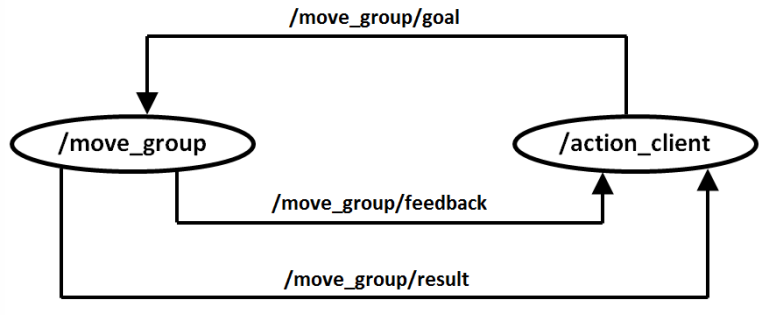

Figure 1: Action server topics

The data contained in these three message types is specified by a `.action` file. The file for the `move_group` action server is located in the `/action` subdirectory of the `moveit_msgs` package. Open this file and examine it.

4.2.1 Test a simple action client

From `lab5_starter`, move `action_client.py` into the appropriate location in your `planning` package. Run `baxter_moveit_gui_noexec.launch` using `roslaunch` to start the `move_group` node, followed by `action_client.py`. The second file should request a motion plan from the action server, then print the result to the terminal. You should also be able to see an animation of the plan in RViz.

Task 4: Now let's try this using the action server. Modify `action_client.py` so that you can input the start and goal states for the manipulator at the terminal, rather than having them hard coded into the program. Since there are a lot of angles, just make it so you can input **the first four joint angles of both the start and goal states**. Test this with several combinations of joint angles and comment on the results. Does the planner always return the same result given identical requests? If not, do you have any ideas why?

4.3 Path Planning

Complete the following steps to run MoveIt on Baxter. (Remember to ssh into the robot in each terminal window that will be interacting with the robot.)

1. ssh into the robot by running

```
./baxter.sh sim
```

2. Begin the Gazebo simulation by running

```
roslaunch baxter_gazebo baxter_world.launch
```

3. Enable the robot by running

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

4. Start the trajectory controller by running

```
rosrun baxter_interface joint_trajectory_action_server.py
```

5. Start MoveIt by running

```
roslaunch baxter_moveit_config demo_baxter.launch right_electric_gripper:=true
        left_electric_gripper:=true
```

Note: Because of how your PDF viewer may render this file, if you copy and paste the above line you may be missing a space between

`right_electric_gripper:=true`

and

`left_electric_gripper:=true`

in which case MoveIt will still launch but it will not work properly so be sure to check that your command is correct.

MoveIt is now ready to compute and execute trajectories on the robot. Please note that for the rest of this section you will need the `joint_trajectory_action_server` and the `demo_baxter.launch` files running.

Note: If things look like they are getting "stuck" in Gazebo, try closing everything and rerunning all the commands again.

4.3.1 Planning with obstacles and orientation constrains

Copy `path_test.py` and `path_planner.py` from `lab5_starter` to the `src` folder inside your `planning` package, and examine the two files. Rather than putting all the base MoveIt code into the `path_test.py` script, we have encapsulated it into the `PathPlanner` class inside `path_planner.py`. Make sure to understand both pieces of code before continuing.

Make sure that `path_test.py` is an executable. Run `path_test` using:

```
rosrun planning path_test.py
```

The robot should loop through three poses in series. As the arm moves, pay attention to the orientation of the right gripper. Does it remain at the same orientation throughout the motion?

While end effector orientation during a manipulator's motion is sometimes unimportant, in other cases it can be critical. Examples include moving liquid-filled containers and performing "peg-in-hole" insertion tasks. MoveIt allows you to plan paths with orientation constraints using the same interface as before.

As mentioned in the introduction, path planning algorithms can also solve the problem of planning with obstacles present in the environment around the robot. The `PathPlanner` class contains the `add_box_obstacle` function, which adds a box-shaped obstacle to the planning scene. While it's possible to add more complex shapes, including shapes from STL or OBJ files, it's rarely worth doing so, as more complex geometries will simply increase computation time.

Task 5: Edit `path_test.py` to add a box representing the table to your scene. For example, try making an obstacle at pose

```
X = 0.5, Y = 0.00, Z = 0.00
X = 0.00, Y = 0.00, Z = 0.00, W = 1.00
```

and for the size you might want to try

```
X = 0.40, Y = 1.20, Z = 0.10
```

. You should see a green box appear in the RViz window, at the position you create the object. Now try creating an artificial "wall" in the air near Baxter's arm.

4.4 Grippers

It's easy to operate Baxter's grippers programatically as part of your motion sequence. Move the file `gripper_test.py` into `planning/src` and (un)comment the appropriate `import` statement so that you are only importing the gripper class from either the `intera_interface` package or the `baxter_interface` package (here let's just use Baxter for now). Make sure the file is in the `src` directory of your `planning` package and try this by running

```
rosrun planning gripper_test.py
```

with the `baxter_world` simulation running which calibrates and then opens and closes Baxter's right gripper in the simulation.

4.5 Pick and Place

Now let's try and do a simple pick and place ourselves. Run this command with the `baxter_world` simulation up

```
rosrun planning spawn_table_and_cube.py
```

Note: This may take about a minute or so to load after the node finishes running. Once it completes, you should see a table and a cube in your Gazebo environment.

Now let's try seeing where the objects are in relation to the world frame. All the objects' transformations are published to the topic `gazebo/model_states`. Try viewing them by running

```
rostopic echo gazebo/model_states
```

Note: You may encounter an error that says "`WARNING: no messages received and simulated time is active. Is /clock being published?`". If this happens, try running "`rosparam set use_sim_time false`" and then try again.

This topic contains 3 arrays: a name array, a pose array, and a twist array. You can use the name array to find what index your desired object is located in and then use that index to find the object's pose and twist (you don't necessarily need to use this in your code for this lab but it is good information to have for a final project).

Now, based on the location of the block and the robot's arm and end effector, what positions do you think Baxter's arm gripper needs to go to in order to pick up the block, move it over, and then put the block back down on the table? Write down the poses. Please note though that the positions of the block and table are given with respect to the Gazebo coordinates which are different from coordinates with respect to the fixed frame base that you see in RViz. From some trial and error we believe the base frame in RViz's z-axis is shifted about 92 cm above the Gazebo world frame. The x and y-axes appear to be about the same.
Note: We recommend first running the command

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

to move the robot's arms to a more neutral position above the table to make the planning a little easier. The easiest way may be to move along just one or two axes at a time. You can view the end effector's position in the Rviz window that pops up when you launch MoveIt.

Task 6: Make a copy of the `path_test.py` file in the `planning/src/` directory. Modify your file so that it moves the arm through the series of poses that you recorded earlier and attempt to perform a pick and place. Try adding a table obstacle constraint as well (for reference the dimensions of the table are 0.913, 0.913, 0.04). Do you need an orientation constraint? You should use code from `gripper_test.py` to open and close the grippers at an appropriate time. Can you make the arm return to the same position, repeatedly, with good enough accuracy to pick up an item?

We don't expect your pick and place task to work perfectly (especially this semester with Gazebo lag playing a part) so please just be able to show us that you can move the Baxter's arm to some series of poses whether or not it can actually pick and place the object.

After completing this task, you might have noticed that open loop manipulation is, in general, difficult. In particular, it's hard to estimate the position of the object accurately enough that the arm can position the gripper

around it reliably. Do you have any ideas about how we might use data from the additional sensors on Baxter to perform manipulation tasks more reliably?

Checkpoint 2

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point you should be able to:

- Explain what an action server is and why it is useful
- Move Baxter using an action server
- Be able to plan a path with orientation and obstacle constraints
- Perform a pick and place task. This doesn't have to work perfectly (or at all), but at least make an attempt. If pick and place doesn't work, what could you do to improve it?

---

5 Controlling Baxter

Now you'll be replacing MoveIt's default execution with a controller of your own. Copy `controller.py` from `lab5_starter` to the `src` folder inside your `planning` package, and examine the file. `controller.py` implements an open-loop, or feed-forward, velocity controller to follow a provided path (a `moveit_msgs/RobotTrajectory` message, which is returned by MoveIt's plan method). You'll be modifying this file to implement a PD (proportional derivative) and a PID (proportional integral derivative) controller on the robot.

The robot trajectory contains not only the desired position at each point, but also the desired velocity (as the number of path waypoints goes to infinity, this becomes the derivative of the desired position). An open loop velocity controller simply sets the manipulator's velocity as the desired velocity at the current point in the path.

If the distance between each waypoint in the path is small, and robot executes each command perfectly, we would expect to see the robot's actual trajectory exactly equal the desired trajectory. However, the world is rarely perfect, and the robot must deal with environmental disturbances, time delays, inaccurate sensors, and imperfect actuators. This is why engineers generally implement closed-loop, or feedback, control. Let's first analyze the performance of the feed-forward controller.

5.1 Feed-Forward (Open Loop) Control

Task 7: Edit the original `path_test.py` to use the controller instead of MoveIt's default execution, then test out the controller. Be sure to check the path in RViz before hitting Enter. You may want to disable your orientation constraints and remove the tables as well.

After each execution, the controller displays a plot of each joint value over time, with the target in green and the measured value in blue. How does the performance look in the plot? Use `tf` (either `tf_echo` or do it programmatically) to check the final end effector pose against the goal. How does the open-loop controller compare to the built-in controller?

5.2 Feedback (Closed Loop) Control

Now you'll be implementing closed loop control on these robots. PID (proportional integral derivative) control is ubiquitous in industry because it's both intuitive and broadly applicable. You'll be learning about PID control in class these next few lectures. The control law is:

$$u = u_{ff} + K_p e + K_i \int_0^t e\,dt + K_d \dot{e} \tag{1}$$

where e is the state error $q_d - q$ (where q_d is the desired position/joint angles and q the current position/joint angles) and u_{ff} is the feedforward term (desired velocity). The controller consists of a proportional term which pulls the error towards zero, a derivative term which dampens the controller and reduces oscillation, and an integral term which compensates for constant error sources (like gravity).

Task 8: First, edit `controller.py` to implement a *PD* controller (ignore the integral term) and execute it on the robot. You should be using the gains specified in `path_test.py` and should not need to change them. How do the plots change from when you used the open-loop controller? What could still be improved?

Finally, add the integral term to implement a PID controller and execute it on the robot. Once again, you should not need to change any of the gains. How do the plots change?

Checkpoint 3

Submit a checkoff request at https://tinyurl.com/106alabs20. At this point, you should be able to:

- Explain the whole of `controller.py`.
- Execute paths using the PID controller.
- Discuss the performance of the open loop, PD, and PID controllers against the default controller.
- What do you think K_w is for? Why is it needed?