2.12: Introduction to Robotics
Lab 4: Motion Planning of Robot Arms*

Spring 2019

Instructions:

1. **When your group is done with each task, call a TA to do your check-off.**
2. **After the lab, zip your files and make a backup using online storage/flash drive.**
3. **No need to turn in your code or answers to the questions in the handout.**

1 Introduction

In this lab we will experiment the following planning techniques:

1. using inverse kinematics (IK) to find joint positions given some target endpoint position (Sometimes called the "tool center point" (TCP)),
2. scripting a complex trajectory,
3. collision checking,
4. using Rapidly-Exploring Random Trees (RRTs) for searching obstacle-free trajectory

2 Setting up the code

Open a terminal (Ctrl-Alt-T), and enter the following commands without the leading $.

```
$ cd ~                      # make sure we are at home folder
$ git clone https://github.com/mit212/me212lab4.git
$ cd me212lab5/catkin_ws    # go into the catkin_ws
$ catkin_make               # let ROS know our packages
$ source devel/setup.sh     # Source the catkin environment
$ source ../software/config/environment.sh   # Source the pman software
```

*

1. Version 1 - 2016: Peter Yu, Ryan Fish and Kamal Youcef-Toumi
2. Version 2 - 2017: Yingnan Cui, Kamal Youcef-Toumi, Steven Yeung and Abbas Shikari
3. Version 3 - 2019: Daniel J. Gonzalez

2.1 Folders and files

- `catkin_ws` : ROS catkin workspace
  - `src` : source space, where ROS packages are located
    - `me212base` : for the 2.12 moving base. We moved this out of the previous `me212bot`.
    - `me212arm` : for the arm experiment in this lab.
    - `me212bot` : for integrating the arm and the moving base.

We will use some contents from `me212bot` developed previously in the new package `me212arm`. Now let's focus on the content inside package `me212arm`, which is for arm motion planning.

- `scripts/planner.py` : a Python library for inverse/forward kinematics of the me212arm.
- `scripts/interactive_ik.py` : a ROS node for interactive IK.
- `scripts/run_planning.py` : a ROS node containing scripted trajectory with multiple way points.
- `scripts/collision.py` : a Python library for testing collisions.
- `scripts/rrt.py` : a ROS node for planning with RRT.
- `model/me212arm.xacro` : the URDF file of the arm represented in xacro format.

In this lab, you need to modify `planner.py`, `run_planning.py`, and `collision.py`.

3 Task 1: Interactive Inverse Kinematics (IK)

3.1 Introduction

System architecture of the arm planning system is shown in Figure 1.

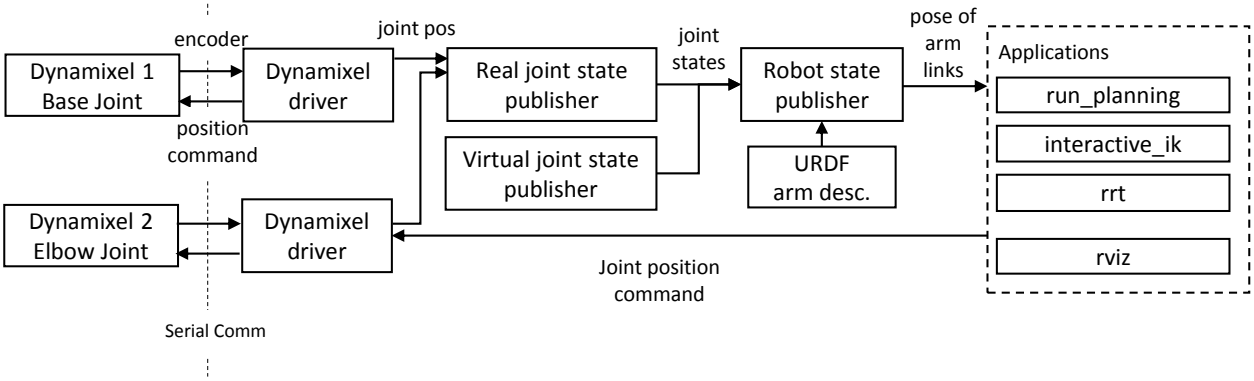

Figure 1: System architecture of the arm system.

Dynamixel There are two Robotis Dynamixel smart servo motors on the arm. Each Dynamixel contains a motor and a controller. This allows us to directly send high level motor commands such as target joint position or velocity. The encoder on board is an absolute encoder. The two

Dynamixels on the robot arm are the MX-64 model. They are very powerful motors so maintain safety distance with the arm when the motor drivers are connected. We have limited the velocity for safety.

URDF The Unified Robot Description Format (URDF) is an XML specification to describe a robot. We will use it with ROS to help us with forward kinematics and visualization. There are ROS packages that subscribe to current joint positions and publish the transform of every link and visualize the arm with mesh models.

To write a URDF file, we can write a `xacro` file to define macros to simplify the file and then generate the URDF later. For example, you can define constants such as `pi`; use some arithmetic operators instead of hard coding numbers; define and use macros to describe similar structures such as left and right arms. The URDF file will be loaded to ROS parameter server with name `robot_description`. During the loading time, we can invoke a `xacro` translator which will expand the xacro file into a URDF file.

Joint state publisher this is a ROS node that reads in `robot_description` in URDF and publish `joint_states`. It is useful for simulating a virtual robot.

Robot state publisher this is a ROS node that reads in `robot_description` in URDF and subscribe to `joint_states`. It publishes transforms of each links. In the URDF we defined 3 frames: `arm_base`, `link_1`, `link_2`, and `TCP` (Tool Center Point, or endpoint). Then you can use `tf` package to do pose transformations, e.g. find the position of the object w.r.t. the `TCP` frame.

Two-link manipulator The two-link manipulator is shown in Figure 2, in which we define the length of two links (a_1, a_2) in meters, and joint positions (q_1, q_2) in rad, and target endpoint position (x, z). **Note that due to the positive direction of Dynamixel motor motion, both q_1 and q_2 are defined as positive about the $\hat{y}$ axis (into the page).** The inverse kinematic solution follows that of [1]:

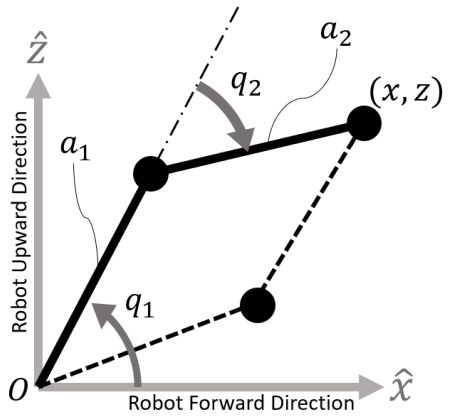

Figure 2: The kinematics of two-link manipulator. **Note that in the configuration shown, q_1 has a negative value, and q_2 has a positive value.**

$$q_2 = \pm 2 \tan^{-1} \sqrt{\frac{(a_1 + a_2)^2 - (x^2 + z^2)}{(x^2 + z^2) - (a_1 - a_2)^2}}, \tag{1}$$

and

$$q_1 = -\text{atan2}(z, x) - \text{atan2}(a_2 \sin q_2, a_1 + a_2 \cos q_2), \tag{2}$$

where atan2(y,x) is a function that computes the arc tangent two variables y and x. It uses the signs of both arguments to determine the correct quadrant of the result. Note that there are two solutions for (q_1, q_2) that corresponds to elbow-up and elbow-down configurations.

3.2 Virtual robot in joint space

We have set up a virtual arm using `joint_state_publisher`. Controlling robot with joint position is the most basic strategy. However, we want the endpoint of the robot arm go to a designated position. That is the inverse kinematics that we will complete.

3.3 Instruction

Fill in function `ik(target_TCP_xz, q0)` in `planner.py` using Equation (1), and (2). Note that the function has an input argument `q0` describing the current joint positions as a Python list: `[q_1, q_2]`. The purpose of feeding the correct position is that when there are multiple solutions, we will choose the solution that is closest to the current position.

You may want to use the following functions in `numpy` module and the exponent Python operator.

- `np.arctan()`, `np.arctan2()`, `np.cos()`, `np.sin()`, `np.sqrt()`
- `np.pi`: constant π
- `x**2`: x^2

To use them, we have imported `numpy` as `np` at the beginning of `planner.py`:

```
import numpy as np
```

Note there are alternatives that you can avoid writing the "`np.`" prefix every time you call the functions. One way is to import the function as follows:

```
from numpy import arctan, arctan2, cos, sqrt
```

Or simply:

```
from numpy import *
```

Although the last one is convenient for quick hacks but not recommended for usual programming because it could create conflicts of function names, and may introduce difficult bugs.

Changing arm description To change parameters of the arm such as link length, joint limits, etc to better match your robot arm, you can modify them in **planner.py** and **me212arm.xacro**.

3.4 Testing

Virtual Testing In practical robot programming, we often use a simulator to make sure the program is correct before running on the real robot for safety and avoiding breaking things. From the **pman** top menu bar, run **Scripts→run-virtual**. And then right click and start command

4.1-planning-interactive. You should see the interface shown in Figure 3. Specify the IK input with the arrows in `rviz`, and see if the solution visualized with the two links is correct. The arrows are known as *interactive marker*, which is a convenient tool for specifying position and orientation visually in `rviz`.

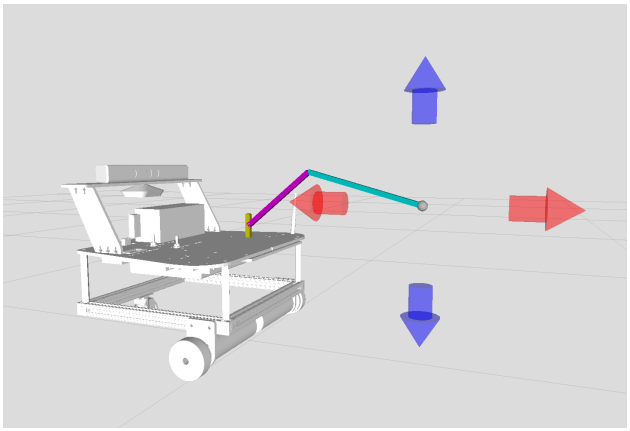

Figure 3: Interactive IK. You can drag the grey sphere in a 2D plane to specify the input (x, z) to the IK program, and then see the IK solution.

Real Testing After you are satisfied with the result in virtual world, you can start testing with real robot arm. First edit `joint1.yaml` under `me212arm/config` folder so that `motor_id` is the number on the motor of joint 1, the one close to the base. Also edit `joint2.yaml` similarly for the motor of elbow joint. **The ID number are labeled on the Dynamixel motors.** Second, in `pman`, stop all the processes by **CTRL-A** (select all) and **CTRL-T** (stop). Then run script `run-real`. Third, start command `4.1-planning-interactive` to control the robot arm using the interactive marker in `rviz`. Please move the marker gently to prevent any damage. Note that in `joint.yaml` is constrained within $(-\pi/4, \pi/2)$ to prevent hitting the robot. Also the speed limit is set to 0.3.

Question 1 *How does the software in* `planner.py` *deal with there being multiple solutions to the inverse kinematics?*

4 Task 2: Scripting trajectories by way points

In this task, you will specify an endpoint trajectory, e.g. a circle, and then use `ik` to find joint positions of each way points.

4.1 Instruction

Fill in **run_planning.py** to plan a circular trajectory. You will use `planner.ik(target_TCP_xz, q0)` that you filled in before. Remember that argument `q0` of the function takes in the current joint position. And a variable `q0` is set by querying the `/joint_states` topic. Thus, you can feed variable `q0` to function `ik`.

4.2 Testing

Virtual Testing Run the code on the virtual arm: stop all commands and run script `run-virtual`. And then run the command `4.2-planning-scripted`. See if the result is correct in `rviz`.

Real Testing run the code in real: stop all commands and run script `run-real`. And then run the command `4.2-planning-scripted`.

Question 2 *You may notice the jerky motion when executing on the real arm. Why? Is there a way to fix it?*

5 Task 3: collision checking

Collision checking is important for generating an obstacle-free path.

5.1 Instruction

Fill in function **in_collision()** in **collision.py** to test whether there are robot links intersecting with the obstacles. Both of them are approximated as a list of line segments. You may use the `intersect(A,B,C,D)` function which test whether line AB is intersecting line CD, where each input is a Cartesian coordinate (x, z). The `intersect` function is based on using cross product to check whether the vectors formed by the 4 points is arranged counterclockwise or clockwise. You can use the examples in Figure 4 to do a sanity check of the provided `intersect` function.

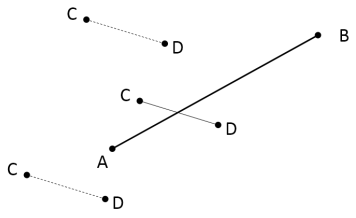

Figure 4: Possible relationships of 2 line segments.

Hint1 - Nested for loop: Here is an example of Python.

```
vlist = [1, 3, 5]
ulist = [2, 4]
for v in vlist:
    for u in ulist:
        print v, u
```

Result:

```
1 2
1 4
3 2
3 4
5 2
5 4
```

Hint2 - Nested lists: The line segments are represented as a nested list. Note that list index starts at 0.

```
segs = [ [ [1,2], [3,4] ], [ [5,6], [7,8] ] ]
print segs[0]
print segs[0][1]
print segs[0][1][0]
```

Result:

```
[[1,2], [3,4]]
[3,4]
3
```

5.2 Testing

In `pman`, while running the virtual arm environment, run the command `4.3-planning-collision-test` and see if the test example matches the answer. It should read:

```
False
True
False
```

6 Task 4: searching obstacle-free path using RRT

RRT (Randomly-Exploring Random Tree) is a sampling-based planning method that searches for a valid (collision-free) trajectory from a start configuration to an end configuration in a continuous configuration space [2]. Pseudocode for general RRT is shown below. We will utilize the collision checking from the previous task to check whether a new joint configuration is in collision and therefore not valid.

```
Algorithm: RRT():
  #Input: Initial configuration: qinit,
  #  target configuration:  qtarget,
  #  function to check if a configuration is valid: VALID(),
  #  number of vertices in RRT: K,
  #  incremental distance: eps
  #Output: A path from initial to neighbors of target configuration.

  G.init(qinit)
  for k = 1 to K
    qrand = RAND_CONF() #Generate a random configuration
    qnear = NEAREST_VERTEX(qrand, G) #Find nearest vertex to random point
    qnew = NEW_CONF(qnear, qrand, eps) #Extend vertex towards random point
    if VALID(qnew) #If this new configuration is collision-free...
      G.add_vertex(qnew) #...add it to our list of valid points.
      G.add_edge(qnear, qnew)
      if CLOSE(qnew, qtarget) #If this new point is close to the goal...
        return BACKTRACE(G, qnew) #...return the full path.
  return NO_SOLUTION #return nothing if no path found in K iterations
```

6.1 Instruction

The RRT code is provided. You should experiment modifying obstacles in `obstacle_segs`, and target endpoint in `target_x`, and other parameters in the top section of `rrt.py` to see how the RRT planner behaves.

6.2 Testing

Virtual Testing Stop all processes. Run script `run-virtual` in `pman`. Run the command `4.4-planning-rrt` and see if the searching works as expected. You should see the search tree growing in `rviz` as in Figure 5.

Real Testing Stop all processes. Run script `run-real` in `pman`. Run the command `4.4-planning-rrt` again.

Question 3 *Do you see some path penetrate through the obstacles? Why?*

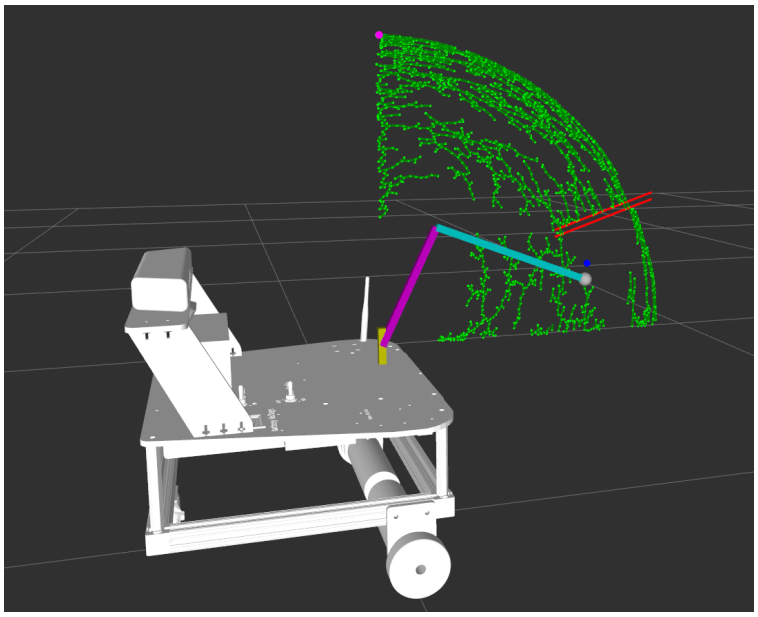

Figure 5: RRT path planning. Red lines: obstacles. Magenta sphere: starting point. Blue sphere: goal, Green lines: search tree.

References

[1] D. Gordon. Robotics: Forward and inverse kinematics. [Online]. Available: http://www.slideshare.net/DamianGordon1/forward-kinematics

[2] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.