# 2.12: Introduction to Robotics
# Lab 3: Introduction to ROS and Motion of Robot Arms*

Spring 2020

**Instructions:**

1. **When your group is done with the lab, call a TA to do your check-off.**
2. **No need to turn in your code or answers to the questions in the handout.**

## 1 Introduction

In this lab, we will go through the basics of ROS and motion of robot arms, including:

1. Set up for running ROS programs
2. The publisher-subscriber communication paradigm in ROS
3. Using inverse kinematics (IK) to find joint positions given some target endpoint position (Sometimes called the "tool center point" (TCP))
4. Scripting a complex trajectory.

## 2 Setting up the code

Open a terminal (Ctrl-Alt-T), and enter the following commands.

```
cd ~                        # make sure we are at home folder
git clone https://github.com/mit212/lab3odrive.git
cd lab3odrive/catkin_ws  # go into the catkin_ws
catkin_make                 # let ROS know and build our packages
source devel/setup.sh    # Source the catkin environment
```

*

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
4. Version 4 - 2020: Jerry Ng, Steven Yeung, Rachel Hoffman, Kamal Youcef-Toumi

### 2.1 Folders and files

- `catkin_ws` : ROS catkin workspace
  - `src` : source space, where ROS packages are located
    - `me212arm` : for the arm experiment in this lab.

  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/run_planning.py` : a ROS node containing scripted trajectory with multiple way points.
- `scripts/odrive_master.py` : a ROS node handling publishing of ODrive axis states and commanding motor set points.

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

### 2.2 Robot Operating System [ROS]

All information regarding ROS, is taken from the ROS website [1] or general knowledge.

In general, there are three main communication paradigms in ROS:

- **Topics:** Uses the publisher-subscriber framework. Meant for situations where there is a continuous stream of information.
- **Services:** Meant for situations where there is a request with some given input, a process which terminates quickly, and a specific reply.
- **Actions:** Meant for discrete behavior that may have longer run time, but provides constant feedback during the process.

In this lab, we use the topics paradigm for all the communication over ROS. The reality is that all of these communication patterns can be used for various tasks with some degree of creativity, but some are more suited for certain tasks than others. When the performance of a system needs to be robust or responsive in a certain fashion, it may become necessary to use different paradigms.

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

The important components for the arm system to run are:

- `roscore`: A collection of nodes and pre-requisites to start any program that utilizes ROS.
- `odrives`: Publishes the joint controller positions which are read from the ODrive, and subscribes to any commands for the two joint controllers.
- `run_planning`: Utilizes `planner.py` to send publish a trajectory command for the ODrives. Subscribes to the joint controller positions to select the specific trajectory to publish based on the inverse kinematics solution.

If this is your first time using ROS, it is worth noting that in not every python script is an actual ROS node. ROS nodes are actively running.

**Question 1** *Though in this lab there is no specific communication paradigm used for yielding the inverse kinematics of a given end effector position, which paradigm should be used?*

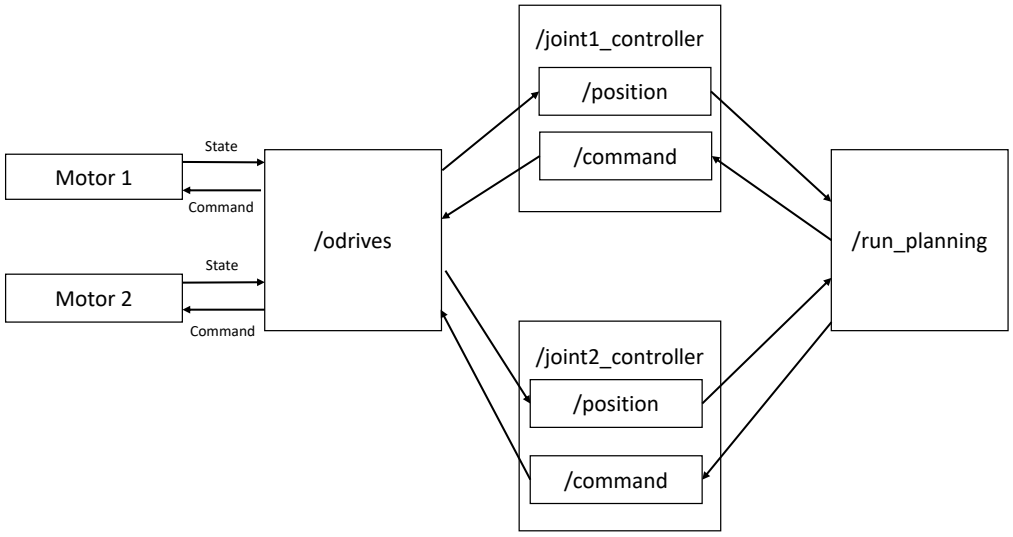


Figure 1: System architecture of the arm system.

### 2.3 Inverse Kinematics

In this section, you will be operating on `planning.py` and inputting the inverse kinematics into the function `ik`.

**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 $(x, z)$. **Though unintuitive, 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 [2]:

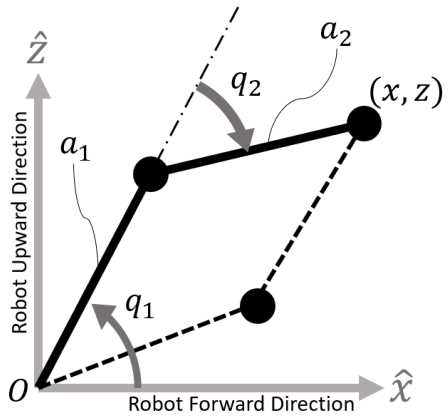


Figure 2: The kinematics of two-link manipulator. **Note that in the configuration shown, $q_1$ has a negative value, and $q_2$ has a positive value.**

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

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.

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.

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 $\pi$
- `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.

**Question 2** *Notice that the function for* `ik` *returns the output of another function,* `select_best_q`. *Why is that done? How does it select the best candidate? How would this change as the number of links increases/decreases?*

### 2.4 Testing

It would be foolish to implement the inverse kinematics and utilize the motors right away to send the robot arm to some given position without checking if the implementation is accurate. As such, test your inverse kinematics implementation by running:

```
cd ~/lab3odrive/catkin_ws/src/me212arm/scripts/
python ik_checker.py
```

The terminal should print a statement saying you passed if you did the implementation properly, and a statement saying you need to check your implementation if it has been done improperly.

### 2.5 Trajectory Planning

In this section, you will be editing `odrive_master.py` and `run_planning.py`.

To begin, edit `odrive_master.py` with the serial number of the ODrive at your table.

To see the robot move arm move to a default desired position, you must do the following:

First, open a terminal and run:

```
cd ~/lab3odrive/catkin_ws
source devel/setup.sh
roscore #this enables the ros environment
```

**WARNING:** When you run this next script, the ODrive will connect, calibrate its encoders, and then begin a homing procedure. This homing procedure involves velocity controlling one motor at a time until the arm hits a hard stop. After the homing is completed, it will rotate away from the hardstop to a safe position. **If the motors are rotating away from the hardstop or rotate quickly, be ready to press the ESTOP.**

In another terminal run:

```
cd ~/lab3odrive/catkin_ws
source devel/setup.sh
rosrun me212arm odrive_master.py
```

After the calibration process has completed and the motor has moved to a safe position, open another terminal and run the following command:

```
cd ~/lab3odrive/catkin_ws
source devel/setup.sh
rosrun me212arm run_planning.py
```

This should move the motor to a specific position. However, we wish to do more in the lab. Specifically, we wish to make the robot arm move in a circle with a given radius and center. To do so, you're going to edit the file `run_planning.py`. There will be comments in the code specifically indicating variables that need to be changed and assigned.

**Question 3** *What safety features are put in place to make sure that the robot can only be commanded to a position within the workspace? How many redundancies are there? Hint: Check* `odrive_master.py`

**Question 4** *In* `odrive_master.py` *there are a lot of commented out functions and functionality. What could this additional functionality be used for?*

**Question 5** *Does the robot arm actually yield the commanded trajectory? Why/why not? Hint: Check how the trajectory is commanded.*

## References


[1] Ros.org — about ros. [Online]. Available: https://www.ros.org/about-ros/

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