Lab 3: Software Integration using ROS: Visual Navigation as example
2.12: Introduction to Robotics
Fall 2016

Peter Yu, Ryan Fish, and Kamal Youcef-Toumi

Instructions:

1. **When your team 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

Today's lab is an introduction to robotic software engineering using ROS framework. Good software practice helps us debug sophisticated system, develop a large system as a team, and reuse/share code. We will learn

1. ROS "topics" and "services" for communication between software modules, often called inter-process communication;
2. ROS "tf" package to handle transforms in robotic system;

You will apply the framework to develop a navigation program with visual feedback. Remember that in Lab 2 we let robots follow a trajectory using odometry. Localization with odometry would accumulate error over time, and the estimated pose is relative to where the robot starts. Here, the visual feedback help you localize in a world frame.

1.1 Prerequisites

To enable you to explore ROS using Python we suggest you have basic knowledge of:

- Python syntax, variables, operators, decision making, lists, and modules. Please see the tutorials on those topics in [1].
- Python `numpy arrays`, and its associated operations. See *Numpy for Matlab users* [2] for a tutorial and also look it up when you want to speak some Matlab in Numpy.

Even if you have used Python for years, it is still easy to forget some particular functions or syntax. Use Google to help yourself first and then consult a TA if you need more help. This way you will be more used to debugging your code when TAs are not around.

2 What is ROS?

ROS stands for Robot Operating System. ROS is not a normal operating system (OS) such as Linux, Windows, or Mac OS, but a platform that runs on top of a normal OS for robotic software development. ROS has two main functions: inter-process communication (IPC) and software package management. We will focus on communication, and touch on ROS Packages as we proceed.

ROS is well-used and well-established in robotic industry. Today, almost every robot hardware component, e.g. robot arms, cameras, has a ROS package (a.k.a. ROS driver) for it. Popular software components, e.g. localization, motion interfaces, also often have ROS packages. They can usually be found on `GitHub.com`.

3 ROS communication

To run a ROS-based system, you must have a `roscore` running. It contains a server for ROS nodes communicate. It is launched using the `roscore` command. There are three types of communication between nodes: *topic*, *service*, and *action*. We will introduce topic and service here but not action. Task 1 is an exercise for publishing and subscribing to a topic. An exercise for creating a service is left in future directions.

3.1 ROS topics

A ROS topic is a communication channel between ROS nodes for certain messages. A ROS node corresponds to a process, which is a running instance of a main program written in C++, Python, or other programming languages. To communicate high-rate data such as odometry or camera images, we use ROS topics.

A topic has publishing nodes and subscribing nodes. A publisher creates the topic and publishes messages through it; a subscriber subscribes to the topic by name and then receives the messages. A conceptual diagram is shown in Figure 1. Here is an example for 1 to 1 communication, a node of moving platform may publish the current odometry, and the controller node will subscribe to it. ROS topic can also let multiple nodes send and receive the same type of messages. For example, a kinect node publishes images; the navigation node subscribes to it to get images for obstacle avoidance, and the computer vision node will use the image to detect objects in the environment.

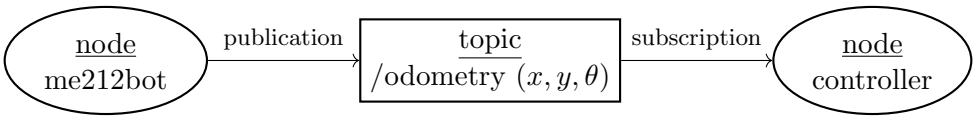

Figure 1: ROS topic concept with an example that node me212bot publishes topic /odometry, to which node controller subscribes.

To publish a message to a topic, we need to:

1. define a message type or reuse a predefined one;
2. initialize a publisher of a topic name;
3. fill values into a message object of the message type;
4. use the publisher to send the message.

Naming Note that the topic name `/odometry` has a leading slash. Much like files, you can organize messages into categories. On your filesystem, you would call these folders, on ROS, they are namespaces. They help you separate and categorize your topics. You may have many moving parts on your robot that you want to track the velocities of, and creating a bunch of global `/vel1`, `/vel2`... would get confusing quickly. Instead, you may sort and name your topics after the part of the robot, e.g. `/left_wheel/vel`, `/right_wheel/vel`.

Below is an example `*.msg` file to define structure for sending 2D odometry:

```
float32 x
float32 y
float32 theta
```

On the other hand, to subscribe to a topic we need to

1. write a handling function for incoming messages;
2. initialize a subscriber with the topic name and the function;

We will practice publishing and subscribing with examples in Task 1.

Using topics can save programming time in handling low-level IPC issues such as

1. finding the IP addresses and creating ports on both sides;
2. establishing network connections;
3. encoding and decoding a data structure into and from byte strings for transmission;
4. sending and receiving data.

ROS topic is convenient for one-direction transmission of high-frequency data such as odometry, control signals, and camera images. Some messages may be dropped due to limited transmission and processing bandwidth, but it does not matter much. However, sometimes a node may want to call a function in another node, which we need to get a response reliably, e.g., detect an object in an image, path planning. This is what ROS service is designed for.

3.2 ROS services

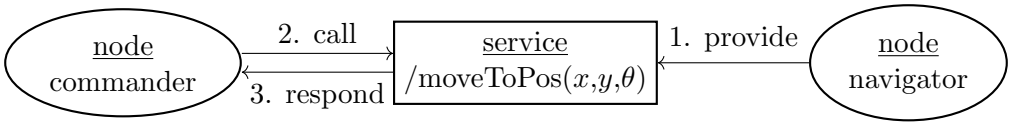

Figure 2: ROS service concept.

A ROS service is a function that can be called remotely. Programmer needs to define the service type in a `*.srv` file, which includes both the structures for *request* and *response*. An example of service type for commanding pose command is shown below:

```
# Request fields
float32 x
float32 y
float32 theta
---
# Response fields
string msg
```

We won't go into detail today. A step-by-step tutorial can be found in [3].

4 ROS tf package

Handling kinematics *correctly* is critical in robotics. ROS tf package helps programmer to manage coordinate frames based on ROS topics. In some ROS nodes, we broadcast the relative transforms between two named frames frequently. For example, node 1 broadcasts transform of frame A relative to frame B, and node 2 broadcasts transform of frame B relative to frame C. And in another node we can use functions in tf to transform a pose relative to A into pose relative to C. Under the hood, ROS tf helps handle the following low-level issues:

- coordinate frame transform using homogeneous transform as we learned in lecture;
- synchronize the timestamps of the time varying transforms in order to respond to a query for transform at a specific timestamp.

Frames, transforms and poses are interchangeable using the homogeneous transform representation. Note that we are dealing with them in Euclidean group SE(3), which have 3 DOFs (degrees of freedom) in translation and 3 DOFs in rotation, in total 6 DOFs. Also note that transforms are directional. It is important to communicate directions *correctly*. There are several ways to specify a direction of the transform shown in Figure 3, and the following expressions are equivalent:

- ${}^{A}_{B}T$ (Craig);
- frame B w.r.t. (with respect to) frame A (Craig);
- frame B relative to frame A (Craig);
- the transform from A to B (ROS) (sometimes ambiguous, not recommended);
- the transform where A is the reference/parent frame and B is the child frame (ROS).

Figure 3: Graphical representation of frame B relative to frame A.

5 AprilTags

AprilTag is a 2D fiducial marker that is designed for easy estimation of the 6D pose of the tag using a monocular camera in realtime [4]. Every tag has an ID which can be identified. AprilTags are wildly used in robotic experiments to simplify and bypass object detection, identification, and pose estimation problems because they are still very challenging in general. In this lab we will use it to estimate robot pose in world frame. See Figure 4 for an example.

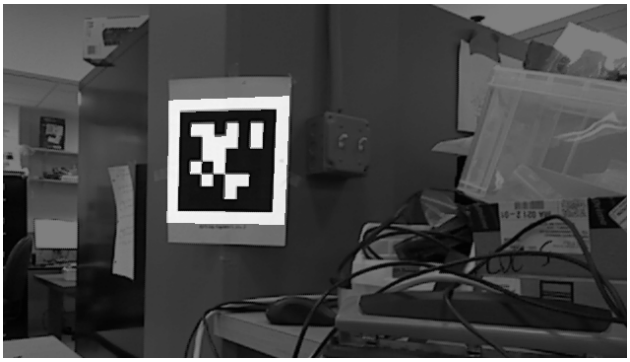

Figure 4: An AprilTag detection example. The detection program overlay highlighted tag image on the whole image after the tag is detected.

6 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/me212lab3.git
$ cd me212lab3/catkin_ws
$ catkin_make                       # compile our packages
$ rebash                            # reload bash resource file, ~/.bashrc
```

6.1 Folders and files

The `me212lab3` folder contains all the files required for this lab. The overall structure follows the ROS catkin build system and is shown below. Unimportant files and folders are ignored for brevity.

- `catkin_ws` : ROS catkin Workspace
  - `src` : source space, where ROS packages are located
    - `me212bot` : a folder containing a ROS package for the 2.12 moving platform.
    - `pr_apriltag` : a folder containing a ROS package for detecting AprilTags [5].

The folder hierarchy helps organize a large project that contains several *packages*. Package is a term for units of files related to one idea. The files can be code for libraries and programs, as well as config files. Now let's focus on the content inside package `me212bot`, which is for our robot platform. We shorten the name from `me212_robot` from Lab 2 for brevity. Important files in the package are listed below.

- `src/controller` :
  - `controller.ino` : a wheel velocity controller in Arduino. (Your old friend.)
  - `helper.h`, `helper.cpp` : helper classes for the controller.
- `scripts/me212bot_node.py` : a ROS node for the moving platform: read odometry from and send velocity commands to Arduino. (Your new friend.)
- `scripts/apriltag_navi.py` : a ROS node for implementing navigation policy using AprilTag. (Your new friend.)
- `scripts/helper.py`, `tf_helper_example.py` : helper functions for doing transform and their example usage.
- `launch/frames.launch` : a launch file for publishing two static coordinate transforms.

In this lab, you need to modify `me212bot_node.py` and `apriltag_navi.py`. We recommend you to use *Geany* text editor because Python uses indents to indicate blocks. It is important to be consistent of the indent characters. Either use tabs or spaces, and use the same amount to indicate one level. The convention in this lab is 4 spaces for one level. To help you be consistent, Geany is configured to show spaces and tabs explicitly.

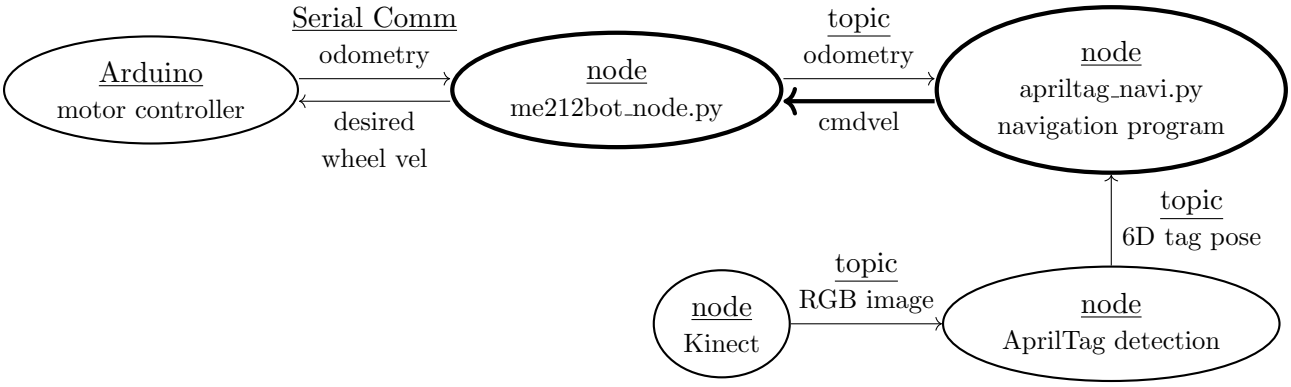

Figure 5: System architecture. The cmdvel topic, shown with a thick arrow, is the one that we will walk through. This involves subscribing in me212bot_node and publishing in apriltag_navi node.

7 Task 1: Publish/subscribe cmdvel through ROS topics

Create a ROS node for our moving platform. Here we name the package as `me212bot`. The main idea here is to package the Arduino controller as a component. See Figure 5 for the system architecture. The Arduino should focus on low level motor velocity control, which won't be changed frequently. Anything higher than that should be moved to a high-level program on the PC side with ROS interface. We remove the `PathPlanner` class from Arduino because this is a high level component and may be changed more frequently. The ROS node `me212bot_node` will act as a translator between ROS topic and serial communication. We outline the 4 major steps to complete Task 1:

Step 1 Upload the provided `controller.ino` to Arduino. Remember to change the motor type in the code according to your robot. The type is written on the robot: either 26 or 53.

Step 2 Define the message type for `cmdvel`.

Step 3 Create a subscriber in `me212bot_node.py` to subscribe to the `cmdvel` and send the commands to Arduino.

Step 4 Create a publisher in `apriltag_navi.py` to publish constant wheel velocities.
Step 1 is straightforward. After uploading the program, you will not see the wheels running because it waits for commands in serial port. The next 3 subsections will describe Step 2 to 4.

7.1 Message type definition

We first need to create our message type for sending desired wheel velocities. Create a text file named WheelCmdVel.msg under `me212bot/msg` folder with the following content. This file describes the message type structure.

```
float32 desiredWV_R
float32 desiredWV_L
```

Second, in file `me212bot/CMakeLists.txt`, uncomment the `add_message_files` section, and add the file name `WheelCmdVel.msg` as follows. This will inform message generator to generate this message type.

```
add_message_files(
  FILES
  WheelCmdVel.msg
)
```

Third, generate the message type by typing these in a terminal

```
$ catmake    # to compile the msg file into Python script to encode/decode
$ rebash     # reload environment variables
$ rosmsg show WheelCmdVel  # check if ROS knows our new message type
```

Command `catmake` is an alias for going into the `me212lab3/catkin_ws` folder and do `catkin_make`.

7.2 Subscription

Create a subscriber to `cmdvel` in main function in `me212bot_node.py`, and fill in `cmdvel_callback` function in `me212bot_node.py`. The function will be called when a message is received.

Hint 1: The message object is named as `msg`. You can access the desired velocities for right wheel by `msg.desiredWV_R`. It is similar for the left wheel.

Hint 2: Here is an example of writing a string of floats through serial channel. The string `strCmd` will be `'0.1,0.2'` with a newline character `\n` appended at the end. The newline is important here because consecutive commands are separated by a newline character.

```
strCmd = '%f,%f\n' % (0.1, 0.2)
serialComm.write( strCmd )
```

Hint 3: Here is a minimal example of subscriber code in Python. You should refer to it for usage and modify `me212bot_node.py`

```
#!/usr/bin/env python
import rospy
from me212bot.msg import WheelCmdVel

def main():
    # init a ROS node named listener
    rospy.init_node('listener', anonymous=True)

    # create a subscriber to /cmdvel topic of WheelCmdVel type
    # with handling function callback()
    rospy.Subscriber('/cmdvel', WheelCmdVel, cmdvel_callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

# will be executed when a msg come in
def cmdvel_callback(msg):
    print 'I received (%f,%f)' % (msg.desiredWV_R, msg.desiredWV_L)

if __name__ == '__main__':
    main()
```

Testing: To test this subscriber node, first run `roscore` in a terminal.

```
$ roscore
```

In another terminal run:

```
$ rosrun me212bot me212bot_node.py
```

In another terminal you can manually publish data by:

```
$ rostopic pub -- /cmdvel me212bot/WheelCmdVel 0.1 0.1
```

The `--` flag is to enable using negative numbers. If the code works properly, the robot wheels should move in the velocity you specified. Remember to turn on the motor. After testing, stop all commands by Ctrl+C, or Ctrl+Z if it does not work.

Note that the usage of rosrun is:

```
$ rosrun PACKAGE_NAME EXECUTABLE_NAME [ARGS]
```

We can do rosrun regardless of the current working directory as long as we do `catkin_make` and `rebash` to let ROS know where the packages are. If the executable is a Python script, the `EXECUTABLE_NAME` will be the Python script name without specifying where it is in the package folder.

7.3 Publication

In the last step we publish the message of command velocity velocity from command line. Now we are going to publish the message using Python. Fill in `constant_vel_loop` function in `apriltag_navi.py`.

Hint 1: Here is a minimal example of publisher code in Python. You should refer to it for usage and modify `apriltag_navi.py`.

```
#!/usr/bin/env python
import rospy
from me212bot.msg import WheelCmdVel

def main():
    rospy.init_node('publisher', anonymous=True)

    # create a publisher to topic /cmdvel, type WheelCmdVel
    pub = rospy.Publisher('/cmdvel', WheelCmdVel, queue_size=1)
    rate = rospy.Rate(100) # 100hz

    wcv = WheelCmdVel()         # create a msg object
    while not rospy.is_shutdown():
        wcv.desiredWV_R = 0.1   # set the values
        wcv.desiredWV_L = 0.2
        pub.publish(wcv)        # publish the msg object

        rate.sleep()            # sleep to keep the loop at 100hz

if __name__ == '__main__':
    main()
```

Testing: To test this publisher node, you'll first run it by:

```
$ rosrun me212bot apriltag_navi.py
```

Then you can use rostopic echo to see the published message.

```
$ rostopic echo /cmdvel
```

After making sure the message displayed is correct, you can run the `me212bot_node.py` to listen to the message and send the command to Arduino.

```
$ rosrun me212bot me212bot_node.py
```

You should see the wheels turning with the velocities you are sending in the code.

8 Task 2: Use ROS tf and an AprilTag to estimate robot pose

Similar to publishers and subscribers in ROS topic, there are publishers (a.k.a. broadcasters) and listeners in ROS tf. You can create both of them using command line tools or in Python code. Let's first use command line tools.

8.1 ROS tf tutorial using command lines

The first way to publish a transform is

```
$ rosrun tf static_transform_publisher x y z yaw pitch roll fid child_fid period
```

Arguments `x/y/z` are in meters, the `yaw/pitch/roll` are in radians, `fids` are frame ids in strings without spaces, `period` is in ms (100 is good). The second way is to use a quaternion for rotation:

```
$ rosrun tf static_transform_publisher x y z qx qy qz qw fid child_fid period
```

Quaternions are generally used because math on quaternions is more stable. There are combinations of raw/pitch/yaw that cause the rotation axes to align, which reduces the degrees of freedom to 5 instead of 6. Although the numbers in a quaternion are hard to understand, your robot will appreciate it. ROS provides utilities that will help us visualize the robot pose. Now let's publish an `apriltag` frame w.r.t. `map` frame as in Figure 6.

```
$ rosrun tf static_transform_publisher 0 0 0.44 0.5 0.5 0.5 0.5 map apriltag 100
```

Use the following command to display the transform in terminal.

```
$ rosrun tf tf_echo map apriltag
```

Use the following command to launch rviz to visualize the frames.

```
$ roslaunch me212bot viz.launch
```

There may be a robot model in display but it's not relavent in this situation.

Experiment 1: With the display and visualization running, change the publisher pose and observe.

Experiment 2: Run `static_transform_publisher`s 4 times in 4 terminals to chain 4 transforms to create a 4-link manipulator with known transforms between consecutive links.

```
$ rosrun tf static_transform_publisher 0 0 0 0.3 0 0   map   link0 100
$ rosrun tf static_transform_publisher 0 0 0.2 0 1.3 0 link0 link1 100
$ rosrun tf static_transform_publisher 0 0 0.2 0 1.3 0 link1 link2 100
$ rosrun tf static_transform_publisher 0 0 0.12 0 0 0  link2 link3 100
```

And find frame link3 in the map frame (forward kinematics).

```
$ rosrun tf tf_echo map link3
```

Or you can find it in the left hand side of rviz under TF/Frames/link3. See Figure 7.

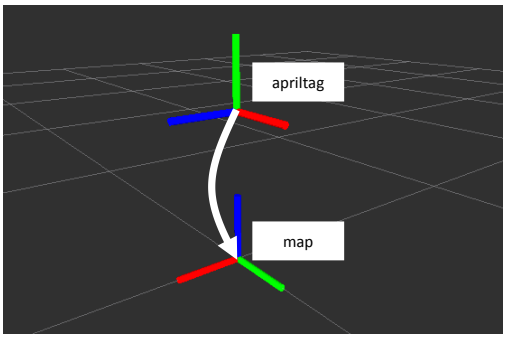

Figure 6: rviz view of the frames published. The white arrow and labels are annotated by hand.

8.2 ROS tf tutorial using Python

Representations In math writing, we usually assume the transforms are represented as 4-by-4 homogeneous transform matrices, ready for multiplication. In ROS convention, a transform is represented by a message type `Pose`, which has the following structure. It contains 7 numbers in total.

- Point position

Figure 7: rviz view of the 4-link manipulator.

  - float64 x, y, z
- Quaternion orientation
  - float64 x, y, z, w

TAs are *lazy* in coding and usually just use a native Python `list [x,y,z,qx,qy,qz,qw]` to represent transforms in Python code. We call it `poselist`. When we need to use ROS tf functions, we use conversion functions to convert `poselist` to/from ROS `Pose`; when we need to do multiplications by ourselves, we covert `poselist` to/from matrix form in `numpy.array`. Side note: be careful of the order of the quaternion numbers. For example, in Matlab, the order is `[qw,qx,qy,qz]`.

Here are the helper functions in `helper.py` written by TA to interact with ROS tf.

- `pose2poselist(pose)`: input ROS Pose and return `poselist`.
- `poselist2pose(poselist)`: input `poselist` and return ROS Pose.
- `invPoselist(poselist)`: input a `poselist` and return its inverse as `poselist`.
- `lookupTransform(listener, sourceFrame, targetFrame)`: return the transform from `sourceFrame` to `targetFrame`.
- `transformPose(listener, poselist, sourceFrame, targetFrame)`: transform a pose in `sourceFrame` into `targetFrame`.
- `pubFrame(broadcaster, poselist, frameId, parentFrameId)`: publish a frame in using the relative transform `poselist` where its parent frame is `parentFrameId` and child frame is `frameId`.

An example code for these functions is in `tf_helper_example.py`. You can run it by running the following 3 commands in 3 terminals.

```
$ rosrun tf static_transform_publisher 0 0 0.44 0.5 0.5 0.5 0.5 map apriltag 100
$ roslaunch me212bot viz.launch  # to see the result of pubFrame
$ rosrun me212bot tf_helper_example.py
```

8.3 The Frames in Visual Navigation

See Figure 8 for the transforms between coordinate frames. There are 4 frames, we name them in ROS as `map` (a.k.a. world), `apriltag`, `camera`, and `robot_base`. And there are 4 transforms of interest:

- ${}^{\text{map}}_{\text{tag}}T$: `apriltag` w.r.t. `map`, fixed and known;
- ${}^{\text{base}}_{\text{cam}}T$: `camera` w.r.t. `robot_base`, fixed and known;
- ${}^{\text{cam}}_{\text{tag}}T$: `apriltag` w.r.t. `camera`, changing, estimated by AprilTag node;
- ${}^{\text{map}}_{\text{base}}T$: `robot_base` w.r.t. `map`, changing, what we want to calculate.

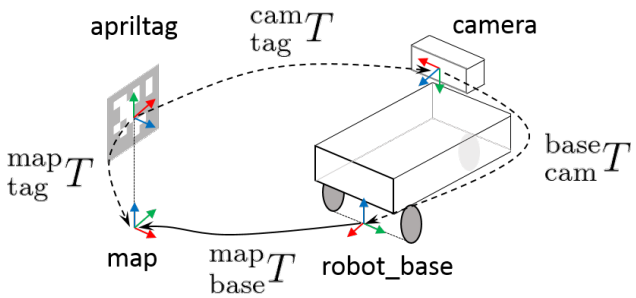

Figure 8: Coordinate frames. The red, green, and blue arrows of the frames correspond to their x, y, and z axes respectively. The black arrows are the transforms between 4 frames. If frame A points to frame B, that means the transform is A w.r.t. B. Dotted arrows are known transforms, and the solid one is what we want to calculate.

We suggest to calculate ${}^{\text{map}}_{\text{base}}T$ with the following operations:

$$ {}^{\text{base}}_{\text{tag}}T = {}^{\text{base}}_{\text{cam}}T \cdot {}^{\text{cam}}_{\text{tag}}T \tag{1} $$

$$ {}^{\text{tag}}_{\text{base}}T = {}^{\text{base}}_{\text{tag}}T^{-1} \tag{2} $$

$$ {}^{\text{map}}_{\text{base}}T = {}^{\text{map}}_{\text{tag}}T \cdot {}^{\text{tag}}_{\text{base}}T \tag{3} $$

8.4 Instructions

Fill in `apriltag_callback` function in `apriltag_navi.py`. You should use the helper functions in `helper.py` introduced in Section 8.2 to do operation (1), (2), and (3).

8.5 Testing (don't type the 6 commands, read the next page)

Now we have several nodes and launch files to run in order to test the AprilTag localization.

```
$ roscore
$ roslaunch me212bot frames.launch     # launch static transform publishers
$ roslaunch me212bot viz.launch              # launch rviz
$ roslaunch freenect_launch freenect.launch # launch Kinect node
$ rosrun me212bot me212bot_node.py           # run the robot node
$ rosrun me212bot apriltag_navi.py           # run the navigation node
```

Here we put the static transform publisher of `apriltag` w.r.t. `map` and `camera` w.r.t. `robot_base` into a launch file `frames.launch`. The basic function of a launch file is to run several nodes in one command. You can open `frames.launch` to understand how to do that. It is just reformatting `rosrun` commands into XML [6].

To run these 6 commands we need 6 terminals, which would be chaotic. Also, remembering and typing all the commands are error-prone. The tool we want to introduce to help us is `pman`. Instead of typing all the commands separately in different terminals, just type:

```
$ pman
```

Then you will see a GUI with the 6 commands already inside (Figure 9). You can right click on each one to start. Note that you should start `roscore` before others. Or you can go to the menu bar on the top of the screen, and click on `Scripts/run`. That is a script written by TAs to run the first 5 commands, and you need to start the `apriltag_navi.py` manually by right clicking because that program can be seen as a *main* program in this test and others are auxiliary.

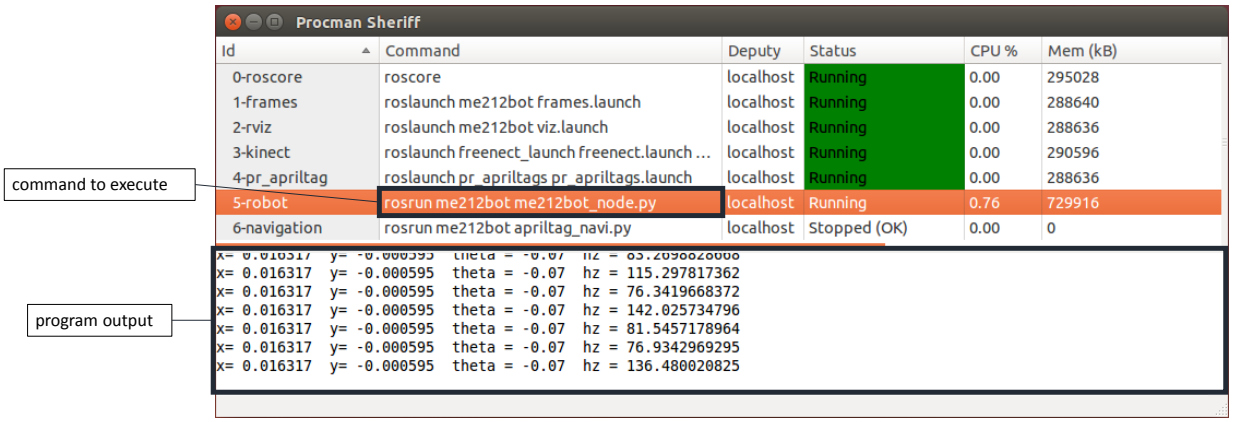

Figure 9: GUI of pman.

If the code are written and run correctly you should see the robot localization result in `rviz` when you hold the AprilTag in front of the Kinect. The result may be noisy depending on the distance to the tag, so the robot may *fly around*.

To provide more details, the command `pman` is an alias for

```
bot-procman-sheriff -l ~/me212lab3/software/config/procman.pmd
```

The program `bot-procman` is a process manager at `libbot` [7] developed in MIT and is used in many robotic projects, including the DARPA Robotics Challenge. You can use it to start/stop/restart a subset of processes. The configuration, including the commands and scripts, can be edited using the GUI and saved to a config file `*.pmd`, or be edited directly inside a config file. You don't need to edit it in this lab.

9 Task 3: Visual Navigation using AprilTag

In last section, we use AprilTag to estimate robot pose in world frame. Now we can run a navigation policy, which receives the robot pose, and outputs wheel velocities for each iteration of the controller loop. The target pose is set to $(x, y, \theta) = (0.25, 0, 3.14)$ in the world frame. Here since

the robot is constrained to have planar motion on the ground, so we use planar representation as in Lab 2. We can project the 6D pose estimated to 3D. TA already filled in `navi_loop` function in `ariltag_navi.py` with a simple navigation policy:

- `if` AprilTag is not in view:
  ⇒ stop moving
- `else if` robot position and orientation is in the target region:
  ⇒ stop moving, and keep stopped
- `else if` robot position is in the target region:
  ⇒ turn to the target orientation
- `else if` robot faces toward the target position:
  ⇒ go straight forward
- `else` :
  ⇒ turn to face the target position

In `apriltag_navi.py`, remember to change the boolean variable `constant_vel` to `False` to use the navigation policy. Then, you put the robot on the ground, in a 2m×2m square in front of the AprilTag, and facing the tag. Use `pman` to start all the processes including `apriltag_navi.py`. You should see the robot navigate to the target pose.

Question 1 *Given the tag size, tag location, camera pose on the robot, camera field of view, farthest distance in which tag can be detected, etc., that you can measure easily, what is the region of starting pose (x,y,θ) that the robot can successfully navigate to the target? You can reason the (x,y) space given fixed θ values, e.g. 0 deg, 15 deg, and 30 deg. Verify it with experiments.*

10 Future directions

Here we list some directions that you can pursue to practice ROS further and to complete your final project:

1. Let the Arduino report wheel velocities, and let `me212bot_node` publish the velocities using ROS topic.
2. Here we assume the robot can always see the AprilTag. You can come up with some simple control policy that can turn the robot to find a tag if no tag is in view.
3. Apply PID to control robot pose in apriltag navigation instead of fixed sets of wheel velocities.
4. Build a localization component that fuse information from odometry and AprilTag. Odometry is local but less jumpy, and AprilTag is global but usually noisy.
5. Create a ROS service in `apriltag_navi.py` to change target poses on the fly.

References

[1] Python tutorial. [Online]. Available: http://www.tutorialspoint.com/python

[2] Numpy for matlab users. [Online]. Available: https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

[3] Ros service tutorial. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29

[4] E. Olson, "Apriltag: A robust and flexible visual fiducial system," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[5] Ros wrapper for the apriltags visual fiducial tracker. [Online]. Available: https://github.com/personalrobotics/apriltags

[6] roslaunch format. [Online]. Available: http://wiki.ros.org/roslaunch/XML

[7] libbot. [Online]. Available: https://github.com/RobotLocomotion/libbot