Neato Horde
Computational Robotics Fall 2014: Mobile Robotics Project

Authors: cfong, srli
Date: 10/14/14
Time spent: ~26 hours

TABLE OF CONTENTS:

I. Project overview

We chose to explore multi-agent robotics for this project. Specifically, we sought to coordinate multiple Neatos to move as a horde or pack -- in this case, demonstrated by autonomously moving and rotating in synch while following a teleoperated 'leader' Neato. This was a non-trivial problem due to the differing lag rates, physical mechanical drift, etc. of individual robots.

Our main goal for this project was to create this synchronized Neato horde, that could later be used as a platform for more interesting behaviours (eg: multi-agent localization and mapping, object/person-tracking, choreographed dances) in future expansions.

Our learning goals included:

- gain experience with odometry for robot localization and control
- gain comfort with coordinate transforms
- explore methods of multi-agent communication
- learn about common localization techniques

II. Implementation of multi-agent system

Our approach to this multi-agent system relied heavily on coordinate frames and the transformations between them. We used the tf ROS package in order to maintain the relationships between these coordinate frames over time.

Coordinate frames

The Neato robots have three built-in coordinate frames:

- Robot odometry coordinate frame (/odom)
- Robot base coordinate frame (/base_link)
- Laser scanner coordinate frame (/base_laser_link)

We added an additional map coordinate frame (/world) as parent to the /odom frames. This allowed us to locate the robots within a constant, global map. Although we considered approaches without a map frame (eg: locating all followers with respect only to the leader robot), we found these to be less intuitive and more difficult to debug. Using the map frame also makes the system more easily scalable.

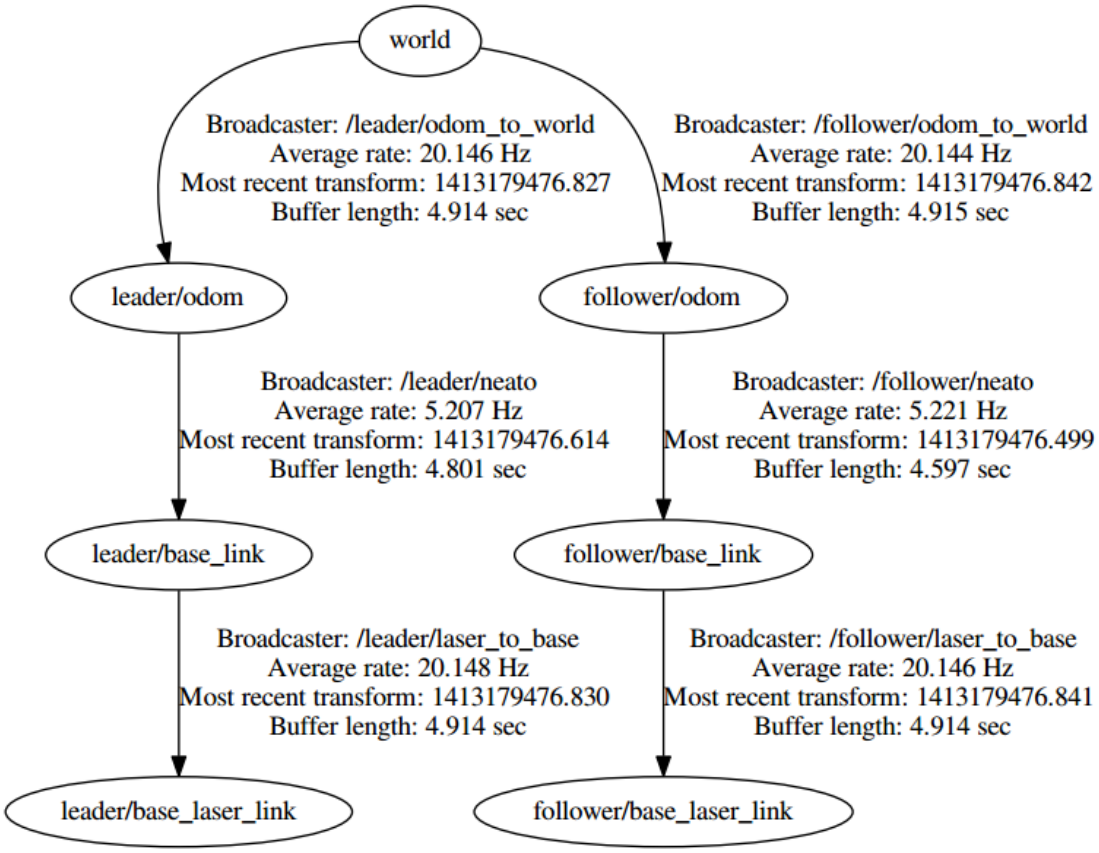

Figure 1: tf tree representation of coordinate frames for a two-robot system

Although we are not currently utilizing the /base_laser_link frame, it would be necessary for dealing with the laser range scan data.

Frame transformations

The relationship between two frames is represented as a 6 degree of freedom relative pose, consisting of a translation (XYZ vector) and rotation (quaternion representing rotation matrix).

For each robot in our system, the frames can be visualized as:

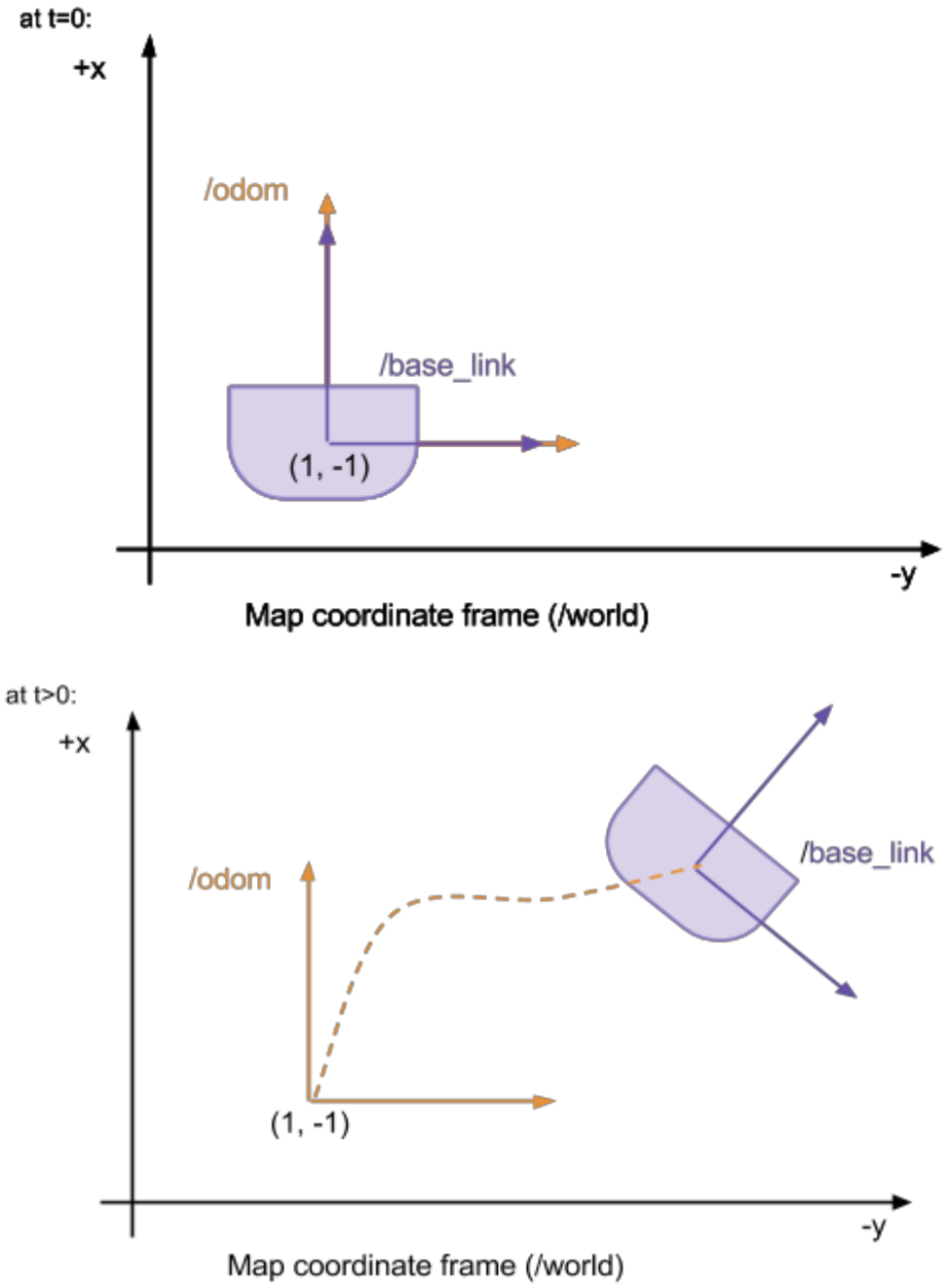

Figure 2:Diagram of coordinate frames upon starting (top) and after running for some time (below)

As indicated in these diagrams, the odometry frame has its origin at the robot's initial starting location (1, -1 in map coordinates). Even after the robot moves, the odometry frame maintains this relationship.

Before the robot moves, its odometry and /base_link frames overlap. As it moves, the /base_link frame also moves. The /base_link frame can be considered the local coordinate frame of the

robot, with normal and transverse axes maintained even during rotation. (Although not depicted in Figure 2, the /base_laser_link frame is offset from the /base_link frame but travels with it.)

Broadcasters and listeners

To get the robot position and movement in terms of the map frame, we created a static transform broadcaster between /world and /odom in our launch file:

```
<!-- Creates static transform publishers between /odom and /world frames -->
<node pkg="tf" type="static_transform_publisher" ns="leader" name="odom_to_world"
  args="0 0 0 0 0 0 world leader/odom 50" />

<node pkg="tf" type="static_transform_publisher" ns="follower" name="odom_to_world"
  args="-0.9 0 0 0 0 0 world follower/odom 50" />

<!-- ALL STATIC TRANSFORMS BASED ON INITIAL POSITIONS GO ABOVE THIS LINE -->
```

We include the known initial location of the robot in this broadcaster in the arguments. In the above example, the follower robot is located 0.9m behind the leader robot in the x direction.

We chose to use a static transform broadcaster implemented in the launch file because the odometry and map frames maintain the same offset relationship throughout the robot's run. If the relationship changed with time, we would instead have implemented the broadcaster as a function in a running ROS node, that activated when the /odom topic updated (according to a Subscriber).

```
    offset = 0
    if neato_name == 'bigbird':
        offset = 3
    br = tf.TransformBroadcaster()
    br.sendTransform((msg.pose.pose.position.x, msg.pose.pose.position.y, msg.pose.pose.position.z),
                     tf.transformations.quaternion_from_euler(0, 0, msg.pose.pose.orientation.z),
                     rospy.Time.now(),
                     '{}/odom'.format(neato_name),
                     "world")



if __name__ == '__main__':
    neatos = ['oscar', 'bigbird']
    rospy.init_node('turtle_tf_broadcaster')
    print('TEST_BROADCASTER RUNNING')
    #turtlename = rospy.get_param('~turtle')
    for neato in neatos:
        #print '/%s/odom' % neato
        rospy.Subscriber('/%s/odom' % neato,
                         Odometry,
                         handle_turtle_pose, neato)
    rospy.spin()
```

As seen in the code above, we attempted this method previously, but realized it was the wrong implementation for our problem.

Handily, the neato_node package already implements the transformations between the /odom, /base_link, and /base_laser_link coordinate frames. However, the transformation between

/base_link and /base_laser_link is implemented through a similar static transform broadcaster, just with different known offsets.

III. Implementation of horde behavior

In this project, our main goal was to implement a leader-follower relationship between two Neatos. We realized that this would require transforms between the two robots, and did research into the TF transforms package that ROS offers. After figuring out how to properly pull transform data from TF, we could simply then send Twist commands to the follower neato depending how the leader was moving.

We created a static world frame, then took transforms of each of the neato base_links to the world. By finding the difference between the follower transform to the world and the leader transform to the world, we were able to determine how much and in what direction the follower neato had to move.

As a reach goal, we decided to implement multiple follower neatos. By refactoring our code to have a single “Robot” class and have a follower method that inherits certain follower only variables, we are able to indefinitely increase the number of followers. However, as uncertainty increases with distance and with number, we decided to stick with 3 Neatos for now, though the code could be easily changed to allow for more.

IV. Challenges faced

Hardware

We had some difficulty interfacing with the Neatos for this project. For some reason, some of the robots would only work with one laptop or the other -- we worked around this problem fine, but never actually figured out what was causing it. Sometimes we experienced a lot of lag (+200ms ping) with the physical robots as well, which made testing difficult.

Sometimes, especially as the project progressed, we ran out of robots. We tried meet at odder hours when we expected more robots to be free, but we were never able to comprehensively test or develop on more than two robots without inconveniencing another team.

Additionally, since we rely solely on the odom data given to us from the Neatos, drifting becomes an issue when the leader robot does erratic motions. Since solutions would likely require additional code to account for object detection or likewise operations, we didn’t really account for this issue. Drift is not too big an issue for instances when the Neato is moving in a straight line or is turning slowly, we have decided to not formally address this problem for now, but acknowledge that this is an issue.

Software

We faced a lot of challenges with implementing the coordinate frames and the transform broadcasters/listeners.

We began by following the ROS tf tutorials, which were useful on a conceptual level but also confusing because of how the turtlesim turtles operate differently than the neatos (eg: not the same breakdown of /odom, /base coordinate frames). The tf tutorials also did not address static transform broadcasters, which is what we ended up using -- and most of the documentation we could find was for C++ which couldn't help with our syntax issues.

We also ran into some challenges with getting the listeners to consistently and continuously listen to the published transforms. We had particular difficulty with getting the rotational component of the frame transformations right. Unfortunately, it turned out that we were listening to the Y-rotation instead of the Z-rotation in our code, so it was human error. After resolving the problem with wrong rotations in our code, our neato ran fine.

In the first half of the project, we struggled a lot with modifying the launch files to deal with multiple robots, and then with implementing the static transforms. This wasn't actually a difficult task, but neither of us was familiar with the XML and we didn't find any beginner-level tutorials or documentation on ROS launch files.

We had a few minor issues with getting catkin to recognize packages that were created elsewhere and then sym-linked into the catkin workspace, but we resolved those fairly quickly.

Additionally, on my side specifically, I'm way more out of practice with OOP than I anticipated. -CF

V. Future improvements

There are many things that we would like to modify and/or add in the future. Although we moved to a class-based structure for this project, it is not as neat or intuitive as it potentially could be.

We would also like to implement the following behaviors.

- Obstacle avoidance by each individual robot (eg: follower robots would not run into obstacles between them and leader robots)
- Ability to transfer leadership between robots
- Ability to have a variety of "formations" that can be switched between at will
- Ability to induct new robots into the pack on the fly (currently impossible given how robots and transform broadcasters are hard-coded into the launch file)
- Ability to localize robots at initialization (ie: not rely on hard-coded offsets from map frame)
- Using sensor data to help correct for odometry drift

We would also really, really like to test with more robots in the horde. As mentioned, we rarely had the chance to test on more than a couple without bothering other teams, and two robots does not quite make a horde.

VI. Applications to future robotics projects

We think that this project would serve reasonably well as a platform on which to execute other multi-agent projects by providing an easy and hopefully intuitive way to access the locations of multiple robots in a global map frame.

Aside from that, the things that we learned in this project -- coordinate frames, using the tf package, etc.. -- are applicable to most robotics projects that involve sensors or multiple components. For example, imagine a mobile robot with a sensor suite including a laser rangefinder and IR sensors. You would probably want coordinate frames for the robot base, odometry, base of the laser rangefinder, and base of each IR sensor to more easily deal with the data.

Additionally, learning how launch files actually work was useful in this project and would probably be useful in almost any other robotics project. That said, there is still *much* more regarding launch files for us to learn.

On a less technical side, we learned that having specifically defined goals (and internal deadlines) are very important to open-ended projects -- especially team projects. Otherwise, it is difficult to evaluate your progress (and, we found, harder to work consistently) over time. Similarly, locking down your goals early on is valuable -- we changed project ideas partway through the project and took a while to solidly decide what behaviors we wanted to focus on, which cost us valuable time.