Project 2: Characterizing odometry

OVERVIEW
In this lab, you will learn about odometry and characterize the accuracy of your turtlebot.

TASKS
1. Turn in and demo Project 1. Set up an SSH key for the Swarthmore Enterprise GitHub server on your Turtlebot computer account (team1 or team2) by following the directions at https://www.cs.swarthmore.edu/~adanner/help/git/git-setup.php. It does not matter which team member is logged into the GitHub website when creating the SSH key.

When you have done this, flag me down and I will work with you to get your local project1 directory submitted to the Swarthmore GitHub. You can also demo your Project 1 to me at this time, too.

2. Background. Please read the “Introduction to tf” at the ROS tf tutorials page: http://wiki.ros.org/tf/Tutorials. Then, read through (and optionally complete) tutorials 1 and 2 (Python versions). You may also want to watch some of the video tutorials as well.

Since we are using git to submit project solutions, it would be good to familiarize yourself with it. I suggest you start out with Andrew Danner’s git introduction at https://www.cs.swarthmore.edu/~adanner/help/git/. For a good reference that links concepts to commands, also see the “Git cheatsheet” at http://ndpsoftware.com/git-cheatsheet.html.

3. Setup. Once you are ready to check out the Project 2 starter code, run the following commands. IMPORTANT: When doing so, please make sure to replace TEAM with your team name (e.g. jdoe1-jsmith1) – you should be able to find yours by listing the repositories at https://github.swarthmore.edu/e28-fall2016).

```
cd ~/catkin_ws/src/e28_labs
git clone https://github.com/swatbotics/transform2d.git
git clone git@github.swarthmore.edu:e28-fall2016/project2-TEAM.git project2
cd ~/catkin_ws
catkin_make
```

Note that to run starter.py for this lab, you will need to make sure you power up the robot and start the robot software using

```
roslaunch turtlebot_bringup minimal.launch
```
4. Drive in a square. Modify the starter.py program in the project2 directory to create a new file, drive_square.py, which replaces the “turn left” behavior with a “drive straight” behavior. This behavior should instruct the robot to drive 1.2192 m forward (this is equal to four 12” floor tiles’ distance) before turning right.

In order to do this, it would be a good idea to understand the full functionality of the starter code, including the transform2d package that you added to your Catkin workspace. Before you try creating your own drive straight behavior, please answer the following questions:

1. How does the turning behavior know when it is done? What computations or variables are involved?

2. What are the purposes of the ANGULAR_RAMP_TIME and ANGULAR_RAMP_SPEED variables?

3. What happens if you change ANGULAR_MIN_SPEED to 0.0? Why?

4. What happens if you lower ANGULAR_GAIN to 0.5? What if you raise it to 20?

5. Why do you think it might be important to make sure that the commanded angular velocity changes continuously? Answer in terms of both wear and tear on the robot, as well as accuracy of odometry.

Note that you will probably want to create (and choose appropriate values for) analogous variables for straight driving. Your drive_square program should exit (rospy.signal_shutdown() in Python) after two full square “laps”. I suggest you implement the quitting behavior by adding a counter member variable to the controller object that increments after each turn. After the program ends, carefully measure the robot’s position and orientation, and compare it to the odometry estimate of the overall change in position and orientation since the program was started.

Assume the robot starts at \((x, y, \theta) = (0, 0, 0)\). Let \((x_e, y_e, \theta_e)\) be the expected final pose of the robot – in this case, also \((0, 0, 0)\). Next, let \((x_m, y_m, \theta_m)\) be the measured final pose when the robot halts (yes, it’s hard to measure the robot’s heading in the lab – do your best). Finally, let \((x_o, y_o, \theta_o)\) be the final pose of the robot reported by odometry – again, assuming you started at the zero pose (in practice, you can compute this by obtaining the relative pose between self.very_first_pose and self.get_current_pose()).

We can now compute two different types of error: execution error, which compares where the robot actually ended up with where we expected it to end up, and dead reckoning error, which compares where the robot actually ended up to where it thinks it ended up. Mathematically,

\[
\begin{align*}
\varepsilon_{e,xy} & = \sqrt{(x_m - x_e)^2 + (y_m - y_e)^2} \\
\varepsilon_{e,\theta} & = |\theta_m - \theta_e| \\
\varepsilon_{d,xy} & = \sqrt{(x_m - x_o)^2 + (y_m - y_o)^2} \\
\varepsilon_{d,\theta} & = |\theta_m - \theta_o|
\end{align*}
\]
Run three trials of the square driving procedure, and report all four of these errors for each trial, along with averages. Also, record a video showing your robot driving a square, along with whatever ad-hoc procedures you devised for measuring the robot’s final position and orientation.

5. **Follow a specific sequence of actions.** Create a new node, `odom_test.py`, which causes the robot to perform the following sequence of actions:

1. Drive four tiles forward.
2. Turn 90° right.
3. Drive two tiles forward.
4. Turn 90° left.
5. Drive six tiles forward.

The robot should pause briefly between each action. Once again, measure all four errors for each of three trials and report them along with their averages.

Also, record video of your robot executing this sequence of actions once.

**WHAT TO TURN IN**

Please use git to add, commit, and turn in your project2 code (both nodes you created).

Finally, write a short informal report describing how you developed your program, including:

- who did what
- answers to the questions in task 4
- what problems you ran into, and how you solved them
- measurements and averages of errors mentioned above
- link to a YouTube video (it can be unlisted, as long as I can access it), showing your robot running the programs associated with both nodes once, along with video documentation of your calibration and measurement procedures

Submit your report (in PDF format) using Git, too.