



Assignment #2:

Path Following

Due Date: See Ninova.

Summary

Assignment: The task is to write a ROS Kinetic package containing node that:

- (1) Subscribes to `/waypoint_cmd` to find out the next waypoint in the route that the robot needs to travel. This route is provided by a separate referee package (`trajectory_referee_456`).
- (2) Uses the `tf` library to find the location of the robot according to its odometry.
- (3) Sends commands to `/cmd_vel_mux/input/navi` to go to each point in the path.

You do not need to make use of the robot's range sensors (if you don't want to). Odometry will be enough for this task.

Submission details: A single C++ or Python file containing the source code.

It should be possible to compile the package by placing the file into a catkin workspace referring to the file and running the catkin compilation.

Step-by-step

Basic knowledge

It is advised that you go over the TF tutorials at <http://wiki.ros.org/tf/Tutorials> (in addition the basic ROS tutorials you were advised to go over before the previous assignment). In particular, “Introduction to TF” and “Write a TF Listener”. Also you may like to see the C++ documentation for the `atan2` function or the Python documentation for the `atan2` function (not strictly necessary).

Set up your workspace

See the step-by-step instructions in the assignment 1 handout. You will need Ubuntu 16.04, ROS Kinetic, the Gazebo simulator with Turtlebot and your own workspace. Recall that if you created your catkin workspace in the directory `~/catkin_ws` then you need to run the following command to allow commands like `roslaunch` work with it:

```
source ~/catkin_ws/devel/setup.bash
```

You may simplify your life by appending this command to your `~/.bashrc` file so that you don't have to retype it frequently.

Set up the referee & skeleton

From the assignment description on Ninova, download the files `trajectory_referee_456.zip` and `trajectory_skeleton_456.zip` and unzip them into your catkin workspace source directory (e.g. `~/catkin_ws/src` if your workspace is `~/catkin_ws`). Then, run `catkin_make` in your catkin workspace. E.g. if your catkin workspace is `~/catkin_ws`:

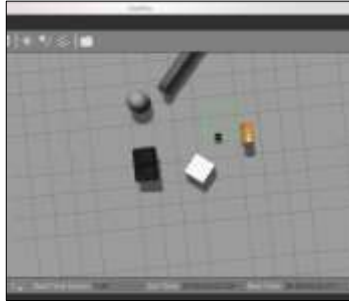
```
cd ~/catkin_ws && catkin_make
```

Run the simulator & referee & skeleton

You will want the simulator running.

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

You should see the Turtlebot robot in the Gazebo GUI amongst a scattering of objects:



In a separate terminal window, run the referee – use ONE OF the following:

```
roslaunch trajectory_referee_456 referee.py route1 dis
```

```
roslaunch trajectory_referee_456 referee.py route1dor
```

If you use **dor** the referee expects you to achieve both target orientation and position, whereas **dis** only expects you to achieve the target position. For full marks your robot needs to be successful with the **dor** setting but for most marks, **dis** is sufficient. Make sure your robot can solve the task with **dis** before you try **dor**.

The referee has several different routes available to it, called **route1**, **route2**, **route3**, and **route4**. The last argument on the command line determines which route is used. **route4** can be considered too difficult for this level, but exists for those who like to be challenged. **route3** may be challenging also.

The referee will start publishing a destination pose to the topic **/waypoint_cmd** – you can examine what it is sending using **rostopic echo** or **rqt_graph**. Carefully examine the output of these commands:

```
rostopic echo /waypoint_cmd
```

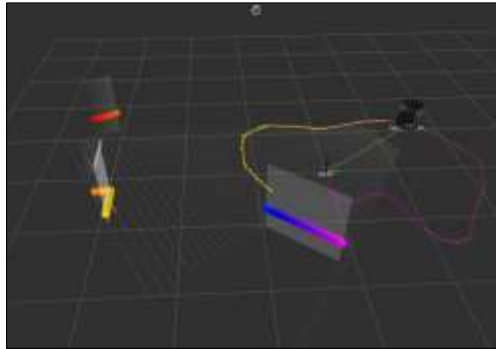
```
rostopic info /waypoint_cmd
```

```
rosmmsg show geometry_msgs/Transform
```

Generally you will want both the simulator and the referee running while you test your code.

In order to see what your robot sees, in a new terminal window run

```
roslaunch trajectory_skeleton_456 view_robot.launch
```



The visualisation will show the output of the robot's self-model, the range camera output (grey point cloud), laser scanner (colourful large circles), the trajectory if published (a colourful path of small arrows), and the robot's frame of reference (`/base_footprint`) relative to its odometry estimation of its original location (`/odom`) as maintained by ROS's TF transforms library.

In particular, rviz should be subscribed to a new topic provided by the referee called `/visualization_marker_array` (you can add this to any rviz session by clicking *Add... Marker Array*). This topic allows you to visualize the full route that your robot needs to follow (though `/waypoint_cmd` only receives the next point at any given time).

Recall: In order to drive the robot around in its simulated world, in a terminal:

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

It is recommended that you do this in order to investigate the problem that you will need to solve with your controller.

Once you are familiar with the referee, the visualisation, and the robot teleoperation, you can run the skeleton code:

```
roslaunch trajectory_skeleton_456 high_speed_controller
```

(if you have used `catkin_make` to compile the cpp file) or, for Python:

```
roslaunch trajectory_skeleton_456 high_speed_controller.py
```

The best way to investigate this setup is to have in different windows arrayed across your workspace¹:

- The simulator (to see what the robot is really doing).
- The rviz window (to see what the robot sees).
- The output of the referee (to report on your progress).
- The output of your controller (for debugging it).
- The keyboard teleoperator (for you to guide the robot yourself). Though you need to close this program (Ctrl-C) before the controller will be able to move the robot.

The following window will also be opened but you wouldn't usually look at it unless there is an error with the simulation:

¹ You will get plenty of advice from other students about how to array your terminals and windows, including programs such as *terminator*, *tmux*, or the tabbing capability of your favourite terminal program. Many Linux window managers allow you a lot of control over your workspace windows with keyboard shortcuts or mouse gestures.

- The terminal window in which you launched the simulator (`turtlebot_world.launch`).

Now you are in a position to edit the controller file `high_speed_controller.cpp` or `high_speed_controller.py`. If you are using the C++ version, use `catkin_make` to recompile (as with assignment 1), and rerun the controller (as above) to see how it performs. Your code would normally go at the bottom of the file `high_speed_controller.cpp` / `high_speed_controller.py` where it says “DRIVE THE ROBOT HERE”. The Python version does not need to be recompiled. You will need to make use of the variables calculated above that point in the code called:

C++ version: `waypoint.translation.x`, `waypoint.translation.y`, `waypoint_theta`, `robot_pose.getOrigin().x`, `robot_pose.getOrigin().y`, and `robot_theta`.

Python version: `waypoint.translation.x`, `waypoint.translation.y`, `waypoint_theta`, `translation[0]`, `translation[1]` (for the robot location), and `robot_theta`.

More information on state and transforms

TF:

In order to obtain the robot's current pose the skeleton code uses the *TF* package to obtain the transform between the robot's original pose (which also happens to be the map frame as long as the robot's odometry agrees with the localisation subsystem) and its currently known pose according to the odometry. These poses are known to the TF transform manager as frames `/odom` and `/base_footprint`.

In order to use TF well, you can examine carefully the output of the following commands to view what is going on:

```
roslaunch tf view_frames && evince frames.pdf
roslaunch tf tf_echo /base_link /odom
roslaunch tf tf_monitor /odom /base_link
```

Waypoints:

Run the following commands to find out more about the waypoint message that is being published:

```
rostopic info /waypoint_cmd
rostopic echo /waypoint_cmd
rosmmsg show geometry_msgs/Transform
```

The skeleton code obtains the target pose from the `/waypoint_cmd` topic.

Marking criteria

- Publishing movements that take into account the waypoint and current state.
- Responding quickly to changes in waypoint or state.
- Publishing movements that get the robot sometimes closer to the target position.
- Publishing movements that get the robot to the target position.
- Publishing movements that get the robot through the route1 trajectory.
- Publishing movements that get the robot through the route2 trajectory.
- Publishing movements that get the robot to the target orientation.
- Clear code.
- Lack of errors.

Bonuses available for:

- Use of Proportional (P) control or another advanced controller.
- Figuring out and making use of the `/route_cmd` topic also provided by the referee.
- Conquering route3 and route4.
- Doing other cool things.

Don't forget to add a README.txt or notes in your code to bring the marker's attention the cool stuff that you do.

Assignments not submitted according to requirements will not be evaluated.