Autonomous Robotic Systems Engineering (AURO)

Week 8 practical - Odometry and autonomous navigation

Example code

As usual, the example code can be found here:
https://github.com/alanmillard/auro-practicals/tree/main

Specifically, there is a new ROS package for Week 8:
https://github.com/alanmillard/auro-practicals/tree/main/week_8

Downloading the code

If you have previously cloned the GitHub repository into a workspace, you should be able to run "`git pull`" from the `src` directory to pull the changes.

If you are starting from scratch, create a workspace and download the packages as follows:

```
mkdir -p ~/auro_ws/src
cd ~/auro_ws/src
git clone https://github.com/alanmillard/auro-practicals.git .

git clone https://github.com/DLu/tf_transformations.git
pip3 install transforms3d
```

Building the new packages

To build the packages, run these commands:

```
cd ~/auro_ws
colcon build --symlink-install && source install/local_setup.bash
```

Remember that you will either need to run the following command every time you start a new terminal, unless you add it to your `~/.bashrc` file:

```
source ~/auro_ws/install/local_setup.bash
```

GAZEBO_MODEL_PATH

IMPORTANT: Before you can run the example code, you will need to set an environment variable to tell Gazebo where to find the TurtleBot3 models. You will need to run this command every time you start a new terminal, or you can add it to the end of your `~/.bashrc` file.

```
export
GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:/opt/ros/humble/york/install/turtlebot3_gazeb
o/share/turtlebot3_gazebo/models/
```

GAZEBO_PLUGIN_PATH

You may also need to set the `GAZEBO_PLUGIN_PATH` environment variable to use the custom differential drive plugin, for example:

```
export GAZEBO_PLUGIN_PATH=$GAZEBO_PLUGIN_PATH:~/auro_ws/build
```

Odometry

The robot's odometry readings are generated by the `gazebo_ros_diff_drive` plugin:

https://github.com/ros-simulation/gazebo_ros_pkgs/blob/ros2/gazebo_plugins/include/gazebo_plugins/gazebo_ros_diff_drive.hpp

https://github.com/ros-simulation/gazebo_ros_pkgs/blob/ros2/gazebo_plugins/src/gazebo_ros_diff_drive.cpp

The settings for this plugin are configured using XML in the TurtleBot3 SDF:
https://github.com/ROBOTIS-GIT/turtlebot3_simulations/blob/humble-devel/turtlebot3_gazebo/models/turtlebot3_waffle_pi/model.sdf#L476-L507

Gazebo odometry source

By default, the SDF file for the TurtleBot3 does not specify an `odometry_source`. This causes the `gazebo_ros_diff_drive` plugin to default to the `WORLD` odometry source, which simulates GPS-like "ground truth" data - i.e. the precise pose of the robot in the world relative to Gazebo's origin:

https://github.com/ros-simulation/gazebo_ros_pkgs/blob/ros2/gazebo_plugins/src/gazebo_ros_diff_drive.cpp#L604-L618

A physical robot would not normally have access to this kind of information - even GPS is only accurate to a few metres (unless used in conjunction with RTK for centimetre-level accuracy). Precise "global" localisation can be achieved indoors with a motion capture system (e.g. OptiTrack), but this is expensive.

Without this external infrastructure, a robot can estimate its current pose based on how far its wheels have turned since its initial pose. The `gazebo_ros_diff_drive` plugin can generate these odometry estimates, but only if the odometry source is set to `ENCODER`:

https://github.com/ros-simulation/gazebo_ros_pkgs/blob/ros2/gazebo_plugins/src/gazebo_ros_diff_drive.cpp#L557-L602

This week's practical examples use a modified local copy of the TurtleBot3 SDF, to set the odometry source to `ENCODER`:

```
<odometry_source>0</odometry_source>
```

This is encoded as an integer, which corresponds to the enum values in the `gazebo_ros_diff_drive` plugin source code.

Drift

Even with the `gazebo_ros_diff_drive` plugin configured to use wheel "encoders" as the odometry source, it still produces very accurate estimates of the robot's pose. This is because it obtains the wheel joint velocities directly from the simulation, so it perfectly senses how fast the robot's wheels are turning.

Physical robots often use rotary encoders (e.g. optical or magnetic) to determine how far their wheels have turned. These are imperfect, so they can only provide an **estimate** of how far the wheels have turned.

When trying to estimate a robot's pose using odometry / dead reckoning (e.g. with differential drive kinematics), the error (difference between the sensed wheel rotation and the true wheel rotation) at each time step will compound over time. The amount of error at each step determines how long it will take before the robot's estimated pose drifts so far from its true pose that it becomes unusable.

This week's practical example code uses a modified version of the `gazebo_ros_diff_drive` plugin to add bias to the wheel encoder readings, to illustrate the effect of odometry drift.

Launch the example as follows:

```
ros2 launch week_8 week_8_launch.py
```

Now teleoperate the robot, and observe the difference in trajectories in RViz:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

Try editing the encoder bias values in `week_8/models/turtlebot3_waffle_pi_odom/model.sdf`:

```
<encoder_bias_left>0.9</encoder_bias_left>
<encoder_bias_right>1.0</encoder_bias_right>
```

These are multipliers, so 1.0 represents no bias. They are currently set to make the left wheel encoder underestimate the true left wheel speed.

Note that you will need to rerun `colcon` to "install" the changes to the SDF each time you make changes.

Returning home

Run the example robot controller as follows (stop teleoperating the robot first):

```
ros2 run week_8 robot_controller
```

This uses the `tf2` system to determine how far the robot needs to turn to return to the `odom` reference frame. This tutorial from the official ROS documentation explains the concepts:

https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Writing-A-Tf2-Listener-Py.html

Try changing the odometry source in `week_8/models/turtlebot3_waffle_pi_odom/model.sdf` from `ENCODER` to `WORLD`, as follows:

```
<odometry_source>1</odometry_source>
```

This will make the `odom` frame relative to the origin of the Gazebo simulation.

Try running the robot controller again, and see how it tries to return here, even though this wasn't its initial pose.

The consequences of drift

Try running the robot controller with increasing levels of odometry drift, and observe the effect on the robot's behaviour.

With the encoder biases both set to 1.0, the robot should safely roam around the `odom` frame, without crashing into the walls.

With encoder biases set to anything other than 1.0, the robot's estimate of its pose will drift further from its true pose over time. The robot controller does not check for obstacles with the LiDAR when it is returning "home", so it will eventually crash into the walls when the odometry estimate becomes too inaccurate.

Autonomous navigation

Last week, we used the navigation stack (Nav2) to semi-autonomously operate a robot via RViz. This week, we are going to use the Nav2 Simple Commander API to set initial pose estimates and navigation goals directly from a ROS node.

You do not need to create your own map this week - one has already been created for you using Cartographer SLAM, and can be found under: `week_8/maps`

Map vs odom frames

Launch the example world and RViz as follows:

```
ros2 launch week_8 week_8_nav2_launch.py
```

Use RViz to set a "2D Pose Estimate" based on the true location of the robot in Gazebo.

Change the RViz visualisation settings so that you can see the TF names:

- TF > Check "Show Names"
- Uncheck the Map visualisation

The `map` frame is set to the centre of the occupancy grid. This is determined by the metadata in `auro_map.yaml`, which sets the origin relative to the "lower-left pixel" of the map, relative to its coordinate system.

The `odom` frame won't be visible initially, as it is based on the robot's initial pose (under the robot's other TF frames).

Try setting a "Nav2 Goal" in RViz - notice that the `odom` frame moves around relative to the `map` frame. AMCL attempts to localise the robot in the `map` frame, using the odometry estimates and LiDAR data.

Simple Commander API

Re-launch the example world and RViz as follows:

```
ros2 launch week_8 week_8_nav2_launch.py
```

This time, don't set an initial pose estimate via RViz. Instead, run this ROS node in a separate terminal:

```
ros2 run week_8 simple_commander
```

This uses the Simple Commander API to automatically set a navigation goal, and then tells the navigation stack to drive to that goal.

Read through the associated code, and make sure you understand how it works. The documentation for the Simple Commander API can be found here: https://navigation.ros.org/commander_api/index.html

Things to try

This is just a basic example that shows how the Simple Commander API can be integrated into a ROS node.

There are various things that you could try to change the behaviour of the robot:

- Edit the navigation stack parameters (`params/nav2_params.yaml`)
  - https://navigation.ros.org/tuning/index.html
  - https://navigation.ros.org/configuration/index.html

- Cancel the navigation from Python (not via RViz) based on some condition
  - For example, after a certain amount of time has passed

- Use the TF system to get the robot's current pose relative to the `map` frame
  - You could use this to record "waypoints" to return to later via navigation goals

- "Fight" the navigation stack for control of the `cmd_vel` topic
  - Rather than cancelling navigation, you can simultaneously publish `Twist` messages to the `cmd_vel` topic to steer/stop the robot