Autonomous Robotic Systems Engineering (AURO)

Week 6 practical - Safety

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 6:
https://github.com/alanmillard/auro-practicals/tree/main/week_6

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/
```

Multi-robot example world

You can launch the example world as follows:

```
ros2 launch week_6 week_6_launch.py
```

This will launch Gazebo, as well as an RViz window for each robot.

Namespaces

Run `rqt_graph` and make sure you understand how the ROS topics are separated for each robot by namespace.

You can filter the output of rqt_graph to remove information that you're not interested in. For example, try adding this filter via the GUI:

```
-/gazebo, -/gazebo/gazebo_ros_state
```

Remapping topics

To teleoperate the robots, you will need to remap the namespaces as follows:

```
ros2 run turtlebot3_teleop teleop_keyboard --ros-args --remap __ns:=/robot1
```

```
ros2 run turtlebot3_teleop teleop_keyboard --ros-args --remap __ns:=/robot2
```

Drive one robot up towards the other, and look at the LaserScan data in RViz. Can the robots detect each other reliably with their LiDAR sensor?

Collision avoidance

You can run a simple obstacle avoidance controller for each robot as follows:

```
ros2 run week_6 robot_controller --ros-args --remap __ns:=/robot1
```

```
ros2 run week_6 robot_controller --ros-args --remap __ns:=/robot2
```

Teleoperate one of the robots and drive it in front of the other robot. Does the autonomous robot always detect the teleoperated robot and avoid it in time?

LiDAR faults

To introduce faults into the LiDAR of `robot1`, launch the `lidar_fault` node, and restart the `robot_controller` node with the `scan` topic remapped:

```
ros2 run week_6 lidar_fault

ros2 run week_6 robot_controller --ros-args --remap __ns:=/robot1 --remap
scan:=scan_faulty
```

To visualise the faulty sensor readings in the RViz window for `robot1`, change the `LaserScan` topic from `/robot1/scan` to `/robot1/scan_faulty`.

Intermittent faults

By default, the `lidar_fault` node will make the `LaserScan` readings intermittent, such that they only reach the robot controller with a 50% chance.

What effect does this have on the robot's behaviour?

Try adding some obstacles to the world via the Gazebo GUI using the primitive 3D shapes in the toolbar (e.g. cube, sphere, cylinder).

How does the faulty robot cope with static obstacles vs other robots?

Other fault types

The `lidar_fault` node contains some example code for other types of fault. Try changing the fault type, as well as the fault probability, and observe the effect on the `LaserScan` readings in RViz.

How do these different types of faults affect the robot's behaviour?

What is the maximum fault probability that it can cope with, for each type of fault?

Try implementing your own custom fault types, and observe their effect.

Mitigating faults

What could you do to mitigate these faults?

Can you modify the robot controller to detect any of these faults?

If so, can you use this information to change the robot's behaviour such that it mitigates the effect of the fault?

If not, can you change the robot's behaviour such that it mitigates the effect of the fault anyway?