Robot Operating System

Lab 4: the navigation stack

1 Goals

This lab aims to discover the capabilities of the ROS 2 navigation stack. This set of packages are designed to have one or several mobile robots move in a known environment. Robots are assumed to carry laser scanners[1], that give a local sensing of their surroundings. These laser scans are used:

- To build the map, if the environment is not known yet (SLAM, Simultaneous Localization And Mapping)
- To localize on a known map (pure localization)
- To detect and avoid obstacles that are not on the map (obstacle avoidance)

1.1 The navigation2 packages

By using the navigation stack you will get familiar with:

- Standard 3D frames for mobile platforms (REP 105)
- Capabilities of the navigation2 stack:
  - SLAM
  - Map server: take a map and publish it to other nodes
  - Localization: estimate the pose of the robot by comparing laser readings to the map
  - Planning: compute the path to a given goal position on the map
  - Control: track the path, avoid unexpected obstacles if needed

In this lab, SLAM is not performed as it takes too much time. Indeed, taking 15 or 20 min of raw laser data and create a map out of it is quite computationally intensive. Some online tutorial explain how to do it from a bag file, either in ROS 1 or ROS 2.
Here we thus assume that a map is given.

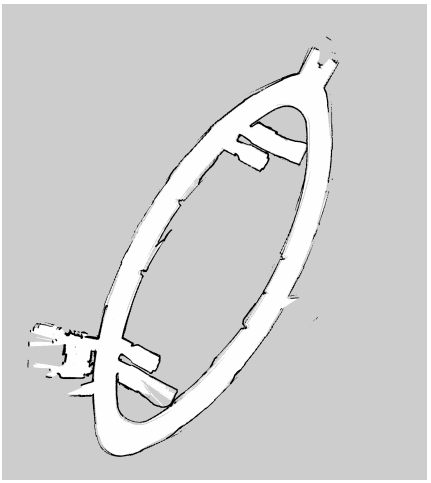
[1] other exteroceptive sensors (camera, depth-camera, ultrasonic) may be used but most tutorial assume laser scanners

1.2 Maps in ROS

In ROS 1 or 2, a map is a combination of two files:

- An image file describing occupied (black) / free (white) or unknown (grey) occupancy grid
- A YAML file describing the link between image space (in pixels) and Euclidean space (in meter)

```
# map image
image: batS.pgm
# size of one pixel in meters
resolution: 0.020000
#coordinates of the pixel (0,0)
origin: [-6.064576, -10.118860, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```



Such maps can be generated from a SLAM algorithm, and are the core data of any localization or navigation capabilities.
Here, the map was built by running a Turtlebot robot on the 4th floor of Building S on the Centrale Nantes campus. The room on the bottom-left is actually my office.

1.3 Compiling and running the simulation

The package should be compiled by `colcon` to have the files discoverable by ROS 2:

```
ros2ws
cd ~/ros2
colbuild --packages-select lab4_navigation
```

In this lab, the map is also used as a simulated environment. Simulated robots can move and measure their distance to the walls or other robots through a laser scanner. The environment is run with:

```
ros2 launch lab4_navigation simulation_launch.py
```

This will create two windows:

- `Simulator 2D` which is the actual simulation (without any robot for now)
- `Rviz2` which is the main visualization GUI in ROS.

The windows can be kept alive during the whole lab.

2 Pure localization

In the first step, pure localization will be considered with manual control. This can be done for one robot by running:

```
ros2 launch lab4_navigation bb8_amcl_launch.py
```

The BB8 robot is displayed in the simulation (true pose) and in RViz (estimated pose).

For this initial run, things should not go as planned. Identify which topics AMCL subscribes to, versus which topics should be used instead. These topics (and frames of interest) are defined in the file `urdf/amcl_param.yaml` that sets a number of ROS parameters for AMCL.
Change their names either here, or using remappings in `bb8_amcl_launch.py`.

2.1 AMCL

The set of all red arrows represents how AMCL (Adaptative Monte-Carlo Localization) works. Here localization is not done eg with a Kalman Filter but with a particle filter. Each particle is a state candidate (pose + velocity). The overall estimation is the center of gravity of all particles.
From laser readings, some particles are pruned as they are not consistent with the scans.

2.2 Interaction with RViz

You can reset the estimation by using the `2D Pose Estimate` button in RViz. Try giving a completely wrong estimate and check that:

- The simulated robot is still at the same place in the simulation window
- Its pose is not the same in RViz, and the laser scans are not aligned with the walls

2.3 Moving the robot

This launch file also runs a slider publisher to move BB8 as a 2-0 robot with a (v, ω) command. When the robot has a slight estimation error, try to have it rotate (pure ω command) to check if AMCL is able to realign the laser scans to the walls.
On the opposite, try to put the robot at a similar but different place (the floor being almost symmetric). AMCL will not be able to tell that the robot position is completely wrong, as the laser scans can be correctly aligned with the known walls.

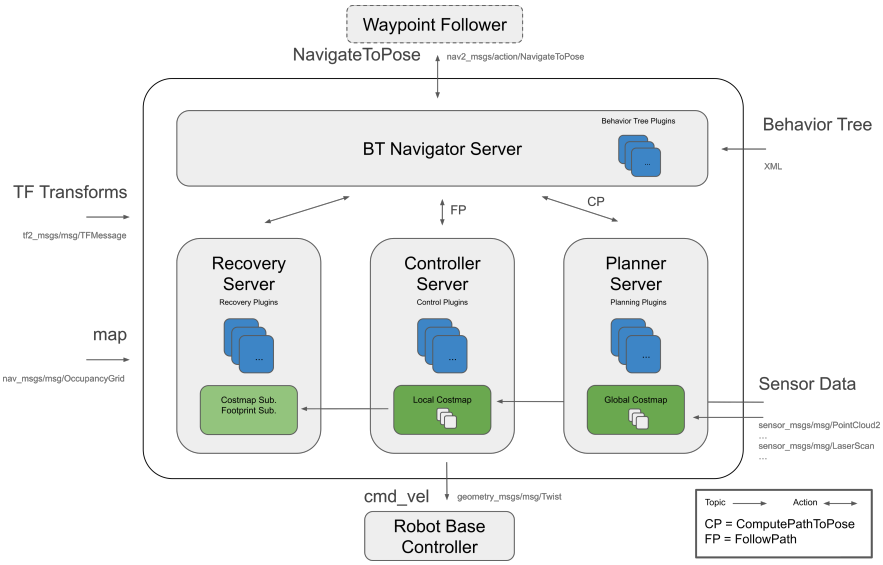
2.4 Nodes and frames

Use `rqt_graph` and TF tools (`tf2_tools view_frames.py`) to display the nodes and the frames:

- Which frames are linked to the robot, to the map, in-between?
- Which node publishes each frame transform?
- Especially, what do the following transforms correspond to and which nodes are responsible for them: `map` → `bb8/odom` → `bb8/base_link`.

3 Planning and control

In this part, we will consider a perfect localization. Instead of running AMCL (that can be CPU intensive), a perfect estimation will be published by running the `vel2joints` node with the parameter `static_tf` set to True.
Many nodes have to be run to have full capabilities of the navigation stack, as shown in the following figure:



3.1 Creating a robot-generic navigation launch file

Four robots are defined in the lab package. There are two BB-type (BB8 / BB9) and two D-type (D0 / D9).

Copy/paste the `bb8_amcl_launch.py` file to a new `spawn_launch.py` file. Then, update this file such that:

- It takes a new argument called `'robot'` with default value `'bb8'`.
- It runs the `vel2joints` node with a parameter `static_tf` set to True.

Install this file by compiling the package again (it is the last time!).

The type of a robot can be deduced at run-time with the following lines in the launch file:

```
robot = sl.arg('robot')
robot_type = sl.py_eval("''.join(c for c in '", robot, "' if not c.isdigit())")
```

`robot_type` will be either `'bb'` or `'d'`. In practice, it runs a bit of Python code to extract the non-digital characters of the robot name: `'bb8'` thus becomes `'bb'`, and so on.

The corresponding model can be loaded accordingly:

```
sl.robot_state_publisher('lab4_navigation',
                         sl.name_join(robot_type, '.xacro'),
                        'urdf', xacro_args={'name': robot})
```

this line will read either `bb.xacro` or `d.xacro` depending on the passed `robot` argument.

3.2 Allowing for manual control

Create a new argument called e.g. `use_nav` with default value False.

A first conditionnal block is to run the slider publisher only if this parameter is indeed False. This is done with the following syntax:

```
with sl.group(unless_arg='use_nav'):
    cmd_file = sl.find('lab4_navigation', 'cmd_sliders.yaml')
    sl.node('slider_publisher', 'slider_publisher', arguments=[cmd_file])
```

Test that the launch file works properly and that you can run BB8 in a terminal, and another robot in another terminal. You should have two slider publishers, one for each robot. In RViz, BB8 and D0 are already setup but you can easily add description / laser scanner display for the other robots.

3.3 Running navigation nodes

If the parameter `use_nav` is True, then the slider publisher should be replaced by the navigation nodes. This is done by creating a block under the condition:

```
with sl.group(if_arg='use_nav'):
    # many things to do in this case
```

In `bb8_amcl_launch.py`, the AMCL node in run through a life cycle manager. This should be the case for all navigation nodes. This can be done through a loop in the launch file, where each navigation node is identified by its package and its executable:

```
nav_nodes = [('nav2_controller','controller_server'),
       ('nav2_planner','planner_server'),
       ('nav2_bt_navigator','bt_navigator'),
       ]
for pkg,executable in nav_nodes:
    # run this executable from this package
    # add the executable name to some list of nodes to be managed by life cycle manager
```

Analyze the syntax used to run a node and add it to the life cycle manager, and run all the navigation nodes.

The parameter file that should be used is now `nav_param.yaml` instead of `amcl_param.yaml`. At this time, you can run:

```
# runs BB8 with nav capabilities
ros2 launch lab4_navigation spawn_launch.py use_nav:=True
# runs D0 with manual control (in another terminal)
ros2 launch lab4_navigation spawn_launch.py robot:=d0
```

Nothing will work as planned with the navigation however. Indeed, the nodes are not correctly configured.

3.4 Adapting the parameters

Compare the files `amcl_param.yaml` and `nav_param.yaml`. The first one explicitely writes that the node is run in the `bb8` namespace. Also, the frames of interest (`bb8/odom`, `bb8/base_link`)

are explicitely given in the parameter file.

It is not the case for `nav_param.yaml`. And it should not, as this file may be used for various robots that all have their own namespace and frame prefix.
We will use a launch capability to update the parameters during launch. Do to that, import the following function in the launch file:

```
from nav2_common.launch import RewrittenYaml
```

This function takes the following arguments:

- `source_file`: the initial parameter file
- `root_key`: any namespace to be used (in our case: the robot variable)
- `param_rewrites`: a dictionary of parameters names with their new value. Such required changes are defined below.
- `convert_types`: should be set to True

Then, the output of the function should be passed instead of the actual parameter file:

```
configured_params = RewrittenYaml(source_file = ...,
                                  root_key = ...,
                                  param_rewrites = ...,
                                  convert_types = True)
sl.node(..., parameters = [configured_params])
```

Parameters that need to be rewritten are:

- All frames, such as `odom` or `base_link`, that should be prefixed with the robot name to make them look like `bb8/odom` or `d0/base_link`. Note that the name of the parameter is to be used in `param_rewrites`, not its default value. For example, `base_link` usually appears as the `robot_base_frame` parameter in `nav_param.yaml`.
- The robot radius, that has default value 0.105 and is used for obstacle avoidance. This should be set to '.27' for a BB-type or '.16' for a D-type. This can be done dynamically:

```
# robot radius depends on the robot type
robot_rad = sl.py_eval("'", robot_type, "'=='bb' and .27 or .16")
```

- The Behavior Tree file, that is usually found in the `nav2_bt_navigator` package. Use this syntax to pass it to the rewritten params:

```
'default_bt_xml_filename': sl.find('nav2_bt_navigator','navigate_w_replanning_time.xml') # or another file
```

3.5 Playing with navigation

Once the launch file is updated, you can spawn BB8 with nav capabilities, and another one with manual control. The `2D Goal Pose` button in RViz allows giving a pose setpoint for BB8 to reach. A trajectory will be found (hopefully) and the control with begin.

Try to have the manually-controlled robot annoy BB8 by going on the planned path. It should be detected by the laser scanner, and BB8 should be able to modify its trajectory.

You can also inspect the `nav_param.yaml` file and try tuning the control / planning parameters. For instance, it may be allowed or not to move backwards, and various thresholds are defined to consider that the desired pose is reached.

4 Conclusion

In this lab, the navigation stack is detailed. You have seen that a single, potentially well-tuned, configuration file can be used for several robots by updating it on-the-fly during launch. The only frame common to all robots is the map, while any other frame is either published by the localization node (odom) or by the robot state publisher.

Many behaviors can be defined to tune how a robot navigates. Feel free to try various Behavior Tree files and see the impact on the robot.