INTELLIGENT ROBOTICS NOTES

October 5, 2022

School of Computer Science
University of Birmingham

Saif Sidhik*, Laura Ferrante*, Yanrong Wang*, and Mohan Sridharan (Module Lead)

Contents

Introduction

0.1 Installing ROS

Ideally, you should have a laptop running Ubuntu 20.04. (We cannot help you solve platform-specific issues if you use any other OS.) (See A.6 for some helpful notes if you don't have Ubuntu)

Follow instructions given in: `http://wiki.ros.org/noetic/Installation/Ubuntu`

0.2 Basic Tutorials

Before beginning the lab tasks, it is important that you understand at least the core concepts of ROS. Following through Section 1.1 in the tutorials given in `http://wiki.ros.org/ROS/Tutorials` is a good place to start.

Note: Although you are free to use any of the available ROS-supported programming languages (C++, python, java, etc.) to write your own nodes, it is recommended that you use Python (2). Nodes written in python are easier to code, debug, and are generally more readable to others. In the link above, you can choose the programming language you prefer to get tutorials in that language.

0.3 Minimum requirements before reading the rest of this document

Read through '1.1 Beginner Level' section of the ROS tutorials (`http://wiki.ros.org/ROS/Tutorials`). Note that some tutorials in the section are repeated for C++ and Python versions. In such cases, you may choose to ignore one.

Before continuing any further, make sure you know how to build ROS packages, and that you understand the concepts of (at least) ROS nodes, topics, messages. You should know how to write basic publisher and subscriber nodes.

Chapter 1

Let's get started!

1.1 Setup socspioneer package

Create a catkin workspace[1] and setup the 'socspioneer' package by following the instructions in `https://github.com/IRUOB/socspioneer`.

1.2 Build and view a map in simulation using Stage and Rviz

Ros Stage is a 2D simulator which creates a simulated robot in a pre-defined world, where it is possible to add different objects for the robot to sense and manipulate. Stage plugs into the ROS publish/subscribe mechanism, so you can control it and receive its simulated laser scan data just as if it was a real robot connected to your laptop. Stage provides various actuator and sensor models (e.g. infrared rangers, scanning laser rangefinder, sonar, etc...).

1.2.1 Constructing a map using Stage and Rviz

To simulate the robot's world and have it build a map of it, do the following.

1. Install the slam-gmapping package:

```
sudo apt install ros-noetic-slam-gmapping ros-noetic-map-server
```

2. Make sure you have `roscore` running in a separate terminal.

3. In the current version of ROS, there are problems aligning the frames of maps created using previous versions. So, we will instead use the robot's sensors in the stage simulator to create a properly aligned map. To do so, open stage with a specific 'world' file in a new terminal.

```
rosrun stage_ros stageros <catkin_ws>/src/socspioneer/data/meeting.world
```

   This will load a line map of the lower ground floor in the stage simulator.

[1] `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`

4. Run the gmapping software in a new terminal. :

```
rosrun gmapping slam_gmapping scan:=base_scan
```

(You can safely ignore the massive warnings once starting this node.)
This starts the 'gmapping' node which is part of the native SLAM (Simultaneous Localisation and Mapping) library that is shipped with ROS. It uses laser sensor data and wheel odometry data to build a map of its world. You will learn more about SLAM in the lectures later in the module. The 'scan' argument specifies the laser scan topic to listen to for building the map.

5. Then open rviz in a terminal (yes, another new one!) visualize the map being built.

```
rosrun rviz rviz
```

6. To display the laser data in rviz, click 'Add' (lower left of rviz screen); on the window that pops up, choose 'LaserScan' under 'By Topic' tab. In the 'Displays' pane on the left, expand the 'Laser Scan' section and check that the 'Topic' variable is set to '/base_scan' (sensor_msgs/LaserScan). Also click and 'Add' a 'Map' and check that the 'Topic' variable is set to '/map'. Finally, under 'Global Options' in 'Displays' pane, for the 'Fixed Frame' variable, choose '/base_link' from the drop-down list.

7. Alternatively, you can write a single launch file that equivalent to step 2-6. This will save you a lot of time in the future. An example is provided at 'socspioneer/launch/stage.launch'. You can run it as after you kill all previous processes:

```
roslaunch socspioneer stage.launch
```

You need to redo `catkin_make` in root of `<catkin_ws>` after you add launch files to run them.

8. Next run the 'teleop' topic in the socspioneer package to allow tele-operation of the robot in the map of LG loaded in stage. In a new terminal:

```
roslaunch socspioneer keyboard_teleop.launch
```

9. **If you want to run through these topics later, you will need to save them in a bag**[2]. To do so, in a new terminal, run:

```
rosbag record -a
```

This will store a <file name>.bag file in the directory in which you run this command. The <filename> is usually based on the date and time at which you executed this command. You can also create a separate "bagfiles" directory and run the `rosbag` command in it. It is also possible to save specific topics into a bag file.

[2]Refer to `http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data`

10. Now, go back to the terminal running 'teleop' and move the robot around using keyboard keys. You will see the robot move in stage and you should see a map forming in rviz. After you have moved the robot around and created a clean map in rviz, run the map saver in a separate terminal to save the map.

```
rosrun map_server map_saver
```

11. There should be a file map.pgm in your directory which you can check visually. *Note: if you were using a real robot, noise in sensing and actuation may result in grey "unknown" areas in the map and walls may not be straight, and you may need a few trials to get a good map.*

12. To use your own map image (for eg. the one you just created), you need a .yaml file corresponding to it; this would have been created automatically. You can also create a useful .world file if you want to load your custom map in stage (optional)[3]. Have a look at one of the existing .world and .yaml files to see what details you may want to change, if any.

13. Close all programs; you can keep `roscore` running if you want to continue playing with ROS (e.g., move to the next task below).

1.2.2 Viewing a constructed map and bag file

If you want to view the bag file you saved above and step through the process of creating the map, please use the following steps.

1. First make sure `roscore` is running in a separate terminal, and that no other processes are still running. Then, run the gmapping software in a new terminal:

```
rosrun gmapping slam_gmapping scan:=base_scan
```

   This starts the 'gmapping' node which is part of the native SLAM (Simultaneous Localisation and Mapping) library that is shipped with ROS.

2. Run rviz in a separate terminal:

```
rosrun rviz rviz
```

   In the 'Display' pane, remember to 'Add' the 'Map' topic and set 'Fixed Frame' variable to have the value '/map'.

3. Run the stored bag file in a new terminal:

```
rosbag play <path to bag file>/<bag file>
```

   You should now be able to replay the construction of the map that you created while recording the bag file. When you are ready to stop, terminate all programs; you can keep `roscore` running if you want to continue working on other tasks.

[3] Refer `https://medium.com/@ivangavran/ros-creating-world-file-from-existing-yaml-5b553d31cc53` for help.

1.3 Writing a basic controller node

For letting the robot explore the floor you will need to send commands to the actuators. The example below show you how to write a simple node which published velocity commands and cause the (simulated) robot to move forward.

1.3.1 Publisher

- Create a node to send motion commands to the robot. An example of the scrips can be the following:

```
#!/usr/bin/env python3
import rospy
from geometry_msgs.msg import Twist

def talker():
    pub = rospy.Publisher('cmd_vel', Twist, queue_size=100)
    rospy.init_node('Mover', anonymous=True)
    # rate = rospy.Rate(10) # 10hz

    while not rospy.is_shutdown():
        base_data = Twist()
            base_data.linear.x = 0.1
        pub.publish( base_data )
        # rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

- If you run the node and look at the stage window, you should see your robot travelling slowly forward. The laser data shown in Rviz should also be changing as your simulated robot moves.

- You can type in another terminal the following command to observe the commands your program is sending to the simulated robot

```
rostopic echo /cmd_vel
```

Try to get used to using

```
rostopic list
```

and

```
rostopic echo
```

- as they will be very useful when debugging your robot's software. As always in Linux, you can press Ctrl+C to kill your program in the terminal.

1.3.2 Laser data

Now you have a Publisher which can command the robot to move in a certain direction, it would be helpful to make use of the available laser sensor data so that the robot can make informed decisions on its own about where and how it should move. First, it would help to know what format the laser data comes in:

- After ensuring stage is running, open a terminal and type

```
rostopic list
```

- These are all the topics (or channels) on which a node can listen.
- The laser data is being published on the topic base_scan. You can 'listen' to it by typing

```
rostopic echo /base_scan
```

- Press Ctrl+C to stop listening. If you examine one of the messages, you will notice that it is simply a header and a long array of numbers, each of which is the range reported by the laser in a particular direction during its spin (`http://docs.ros.org/noetic/api/sensor_msgs/html/msg/LaserScan.html`).

1.3.3 Subscribing to laser messages and publishing to robot controller

We can now use those ranges when we come to write a program to control the robot. For instance we could make the robot moving around the floor while avoiding obstacles.

- Use the ranges[] field in the LaserScan messages to obtain the laser data. You can find here `http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html` the API.
- Read the tutorial (`http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)`) on writing a node in Python that subscribes to a topic and publishes on another topic. Write a node that subscribes to LaserScan messages on /base_scan topic, and publishes (to avoid obstacle and navigate) movement command messages to the robot on the /cmd_vel topic.
- Use rostopic echo /base_scan to examine the laser messages coming in from the robot. What sort of ranges do you get when the robot is in front of an obstacle?
- The field in the Twist object to move the robot forward is twist.linear.x (positive values to drive forward; negative values to drive in reverse). Click here (`http://wiki.ros.org/mini_max/Tutorials/Moving%20the%20Base` for a tutorial on how to move your base in Python.

- The program rqt_graph (`http://wiki.ros.org/rqt_graph`) provides an helpful interactive display of the current ROS nodes, including which topics they are publishing on and subscribing to.

1.4 Practice Exercise: The Free Spirit!

Write a code that makes the robot move in the (entire) map **on its own** while **avoiding obstacles**. This will involve writing a node that has a subscriber to listen to the laser data, and a publisher to publish commands to the robot's wheels. You will have to come up with a strategy that makes it move freely when there is no obstacle, and to avoid colliding with a sensed obstacle. How much area of the map does your robot manage to cover? How long does it take for it do it?

Hints!

1. It may help to group the laser ranges into a number of blocks (e.g. 'left', 'right', 'straight on'), then find the average range in each block, and simply turn the robot left, right, or carry on straight ahead, depending on which laser block has the longest average range.

2. The fields for sending commands to the robot are twist.linear.x (forward motion; negative to reverse) and twist.angular.z (z-axis anti-clockwise rotation; negative to go clockwise).

3. Movement commands only last for a short duration (less than a second) to prevent the robot driving into danger if one of the nodes crashes. For continued motion, you should publish the movement commands repeatedly in a for-loop, using sleep command (or a similar length of time), or alternatively calculate and publish appropriate movement commands every time you receive a LaserScan message.

Chapter 2

Assignment 1: Where am I?

Localisation using Particle Filter

In this assignment you will implement and visualise particle filter localisation of the robot within the map that you built. This chapter will get you started on writing your own localisation node.

2.1 Localisation with AMCL

Before you write your own particle filter, it is good to test the AMCL library that comes built-in with ROS. To see how localisation within ROS works, first get the AMCL localisation software up and running:

1. Make sure `roscore` is running in a terminal. In a new terminal, run the robot simulation in Stage:

   ```
   rosrun stage_ros stageros <catkin_ws>/src/socspioneer/data/meeting.world
   ```

2. Run a map server in another terminal:

   ```
   rosrun map_server map_server <path_to_your_map_yaml_file>
   ```

3. Run the AMCL localisation node **(If you get an error saying 'amcl' package is missing when you run the following command, install it by running the command 'sudo apt install ros-$ROS_DISTRO-amcl'.)**

   ```
   rosrun amcl amcl scan:=base_scan
   ```

4. To visualise the localisation:

   - Run rviz:

     ```
     rosrun rviz rviz
     ```

   - Set the Fixed Frame to */map*:
   - Add a Pose Array view listening on the */particlecloud* topic

- Add a Map view listening on the */map* topic
- Finally, click the *2D Pose Estimate* button, then draw an arrow on the map showing the approximate location and direction in which your robot is facing in Stage (e.g., Figure 2.1).

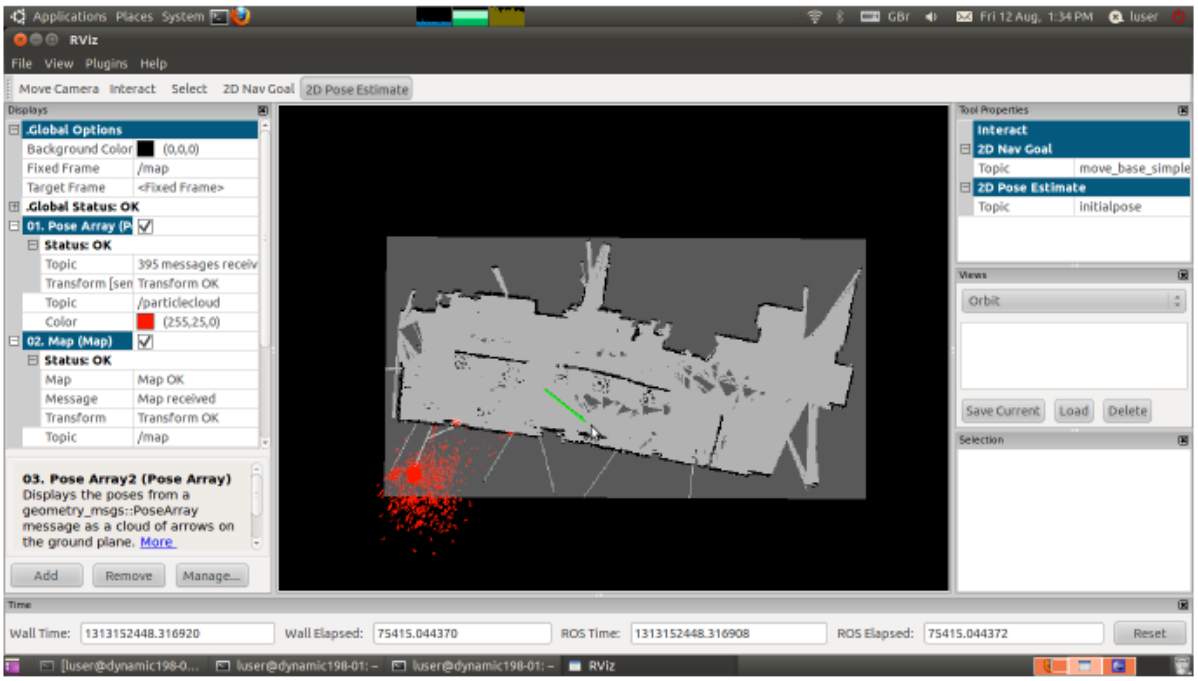


Figure 2.1

5. Move the robot around using the 'teleop' option described earlier:

```
roslaunch socspioneer keyboard_teleop.launch
```

AMCL will read in the laser data (topic */base_scan*) and odometry data (topic */odom*) and continually update the particle cloud with the predicted location of the Pioneer.

2.2 PF Localisation: Writing your own localisation node

Link to `pf_localisation` package: `https://canvas.bham.ac.uk/courses/65665/files/13694431/download?download_frd=1` (zip file is also available on Canvas)

For this exercise, you will write a replacement ROS node for AMCL in python which runs a basic particle filter and localises the robot within the supplied map. Unpack the *pf_localisation* python package (provided) into your '*<catkin_workspace>/src*' directory. The instructions to build the package and run the node can be found in 'README.md' file in the package.

This package contains a ROS node '*scripts/node.py*' which handles all the ROS backend stuff, and an abstract class PFLocaliserBase (*'src/pf_localisation/pf_base.py'*) which provides an easy way to interface with ROS and incorporate a SensorModel (*'src/pf_localisation/sensor_model.py'*) object (*'self.sensor_model'*) that provides particle weight calculation. Your task for this exercise is to write a

class called PFLocaliser which will extend PFLocaliserBase and provide localisation given a map, laser readings, and an initial pose estimate. A skeleton of this class is provided in *'src/pf_localisation/pf.py'*. **You can successfully implement the particle filter localisation by editing JUST this file**. The remainder of this section will describe the main parts of this class.

2.2.1 Constructor

The first thing your constructor should do, as when implementing any subclass, is to call the superclass constructor. Your constructor will also need to assign values for the odometry motion model:

```
# Set motion model parameters
self.ODOM_ROTATION_NOISE = ???? # Odometry model rotation noise
self.ODOM_TRANSLATION_NOISE = ???? # Odometry model x axis (forward) noise
self.ODOM_DRIFT_NOISE = ???? # Odometry model y axis (side-to-side) noise
```

These values will be used in the odometry update, in the method *'PFLocaliser:predict_from_odometry()'* inherited from the superclass. You will then need to implement the following three abstract methods.

2.2.2 initialise_particle_cloud(self, initialpose)

- Called whenever a new initial pose is set in rviz.
- This should instantiate and return a PoseArray object, which contains a list of Pose objects. Each of these Poses (corresponding to a particle in the particle cloud) should be set to a random position and orientation around the initial pose, e.g. by adding a Gaussian random number multiplied by a noise parameter to the initial pose.
- Orientation in ROS is represented as Quaternions, which are 4-dimensional representations of a 3-D rotation describing pitch, roll, and yaw ("heading"). This is more complex as the Pioneer only rotates around the yaw axis. To make it easier for you, you have been provided with some utility methods for handling rotations in *'util.py'* imported in code as 'import pf_localisation.util':

  ```
  rotateQuaternion(q_orig, yaw)
  ```

  Takes an existing Quaternion q_orig, and an angle in radians (positive or negative), and returns the Quaternion rotated around the heading axis by that angle. So, for example, you can take the Quaternion from the Pose object, rotate it by Math.PI/20 radians, and insert the resulting Quaternion back into the Pose.

  ```
  getHeading(q)
  ```

  This function performs the reverse conversion and gives you the heading (in radians) described by a given Quaternion q.

2.2.3 update_particle_cloud(self, scan)

- Called whenever a new LaserScan message is received. This method does the particle filtering.
- The PFLocaliserBase will have moved each of the particles in the map (approximately) according to the odometry readings from the robot. But odometry measurements are unreliable and noisy, so the particle filter makes use of laser readings to confirm the estimated location.
- Your method should get the likelihood weighting of each Pose in self.particlecloud using the *self.sensor_model.get_weight()* method. This weighting refers to the likelihood that a particle in a certain location and orientation matches the observations from the laser, and is therefore a good estimate of the robot's pose.
- You should then resample the particlecloud by creating particles with a new location and orientation depending on the probabilites of the particles in the old particle cloud, for example by doing roulette-wheel selection to choose high-weight particles more often than low-weight particles. Each new particle should have resampling noise added to it, to keep the particle cloud spread out enough to keep up with any changes in the robot's position.
- The new particle cloud should be assigned to self.particlecloud to replace the existing one.

2.2.4 estimate_pose()

- *self.particlecloud* should return the estimated position and orientation of the robot based on the current particle cloud. You could do this by finding the densest cluster of particles and taking the average location and orientation, or by using just the selected 'best' particle, or any other method you prefer – but make sure you test your method and justify it! Taking the average position of the entire particle cloud is probably not a good solution... can you think why not?
- To find the average orientation of a set of particles, you will encounter a problem because angles increase from 0 to pi radians then continue from -pi back to 0. For example, if you have two particles, one facing at -179 degrees and one facing at 179 degrees (only 2 degrees apart in reality), the mean orientation will be -179 + 179 / 2 = 0 degrees, not 180 degrees.
- This situation is exactly what quaternions were created for. Instead of calculating the heading of each particle and finding the mean, you can simply take the mean of each of the x, y, z and w values directly from the Quaternions (using getW() etc.) before creating a new Quaternion with these mean values. This new quaternion represents the average heading of all the particles in your set. Instead of using the whole set of points in your cloud, you might want to do some filtering to choose the ones you want to rely on for orientation.

2.3 Hints!

- As well as seeing your code running as your Pioneer explores your map, we will look for evidence that you have experimentally (i.e. in the form of records in some log book) investigated how the particle filter behaves, including (but not limited to): adding noise to the particle cloud, visualising

the sensor model function, etc. *self.sensor_model.predict(obs_range, map_range)* by plotting a graph of probabilities for various observation and prediction ranges, and measuring the overall location error over time.

- Remember to go through the 'README.md' file in the *'pf_localisation'* package for more information about the package and its usage.
- Remember to use *'rostopic echo <topic>'* and/or rviz to listen to the messages which are being passed between all nodes, including yours for debugging, testing etc.
- Your node, unlike AMCL, also publishes the estimated pose as a message which can be displayed by rviz. Add an rviz display for messages of type 'PoseStamped' on topic *'estimatedpose'* to see the exact position estimate for your robot.
- The node does not call your PFLocaliser update methods until odometry update messages have been recieved (i.e. it does not update automatically whenever there is a laser scan, only when the robot has moved). So you will need to run *'teleop_joy'*, *'keyboard_teleop'* or the exploration node to make the robot move before your particle cloud will appear on the map in rviz.
- Rotations around the compass are in radians. For example, 0 degrees is North and -3/4 pi degrees is South-West. You may find it easier to keep everything in radians as *math.cos()* and *math.sin()* take their arguments in radians, but if you want to work in degrees you can make use of the *math.radians()* method. For example, you could use *rotateQuaternion(heading, math.radians(90))* to rotate by 90 degrees, or *rotateQuaternion(heading, math.pi/2)* to do the same rotation in radians.

Appendix A

Additional Notes

A.1 Rviz

Rviz (`http://wiki.ros.org/rviz`) is a visualisation tool which is used to view the robot's position in the map and can also be used for drawing your own shapes onto the map. Below are the various different things that can be added to an rviz 'scene' (by clicking the 'add' button in the sidebar) The following is a bare-bones node which adds a spherical Marker at the origin of the */odom* frame. Functions can be created to add arbitrary shapes or text for debugging or display purposes. Note that lots of identical markers are better processed as a MarkerArray.

```
#!/usr/bin/env python
import rospy
# documentation: http://wiki.ros.org/rviz/DisplayTypes/Marker
from visualization_msgs.msg import Marker, MarkerArray
from geometry_msgs.msg import Point

rospy.init_node(name='marker_demo')
mark_pub = rospy.Publisher('visualization_marker', Marker, queue_size=10)
id_counter = 0

# place a point
m = Marker()

# specify reference frame (options in RViz Global Options > Fixed Frame)
# markers relative to 0,0 in odometry
m.header.frame_id = '/odom'
m.header.stamp = rospy.Time.now()

# marker with same namespace and id overrides existing
m.ns = 'my_markers'
m.id = id_counter
```

```
id_counter += 1

m.type = Marker.SPHERE
m.action = m.ADD

m.pose.position.x = 0
m.pose.position.y = 0
m.pose.position.z = 0
m.pose.orientation.x = 0
m.pose.orientation.y = 0
m.pose.orientation.z = 0
m.pose.orientation.w = 1
m.scale.x = 1
m.scale.y = 1
m.scale.z = 1
m.color.r = 1
m.color.g = 0
m.color.b = 0
m.color.a = 1

m.lifetime = rospy.Duration() # forever
#m.lifetime = rospy.Duration(...)

mark_pub.publish(m)
rospy.spin()
```

A.2 Stage Simulator

Stage is the name of the robot simulator. It is fed a description of the world and the robot in a .world text file, and an occupancy map in the form of a .pgm image. Below is a script to start the stage simulator, map server and rviz (using the world description given in the socspioneer directory which should be present on the provided laptops):

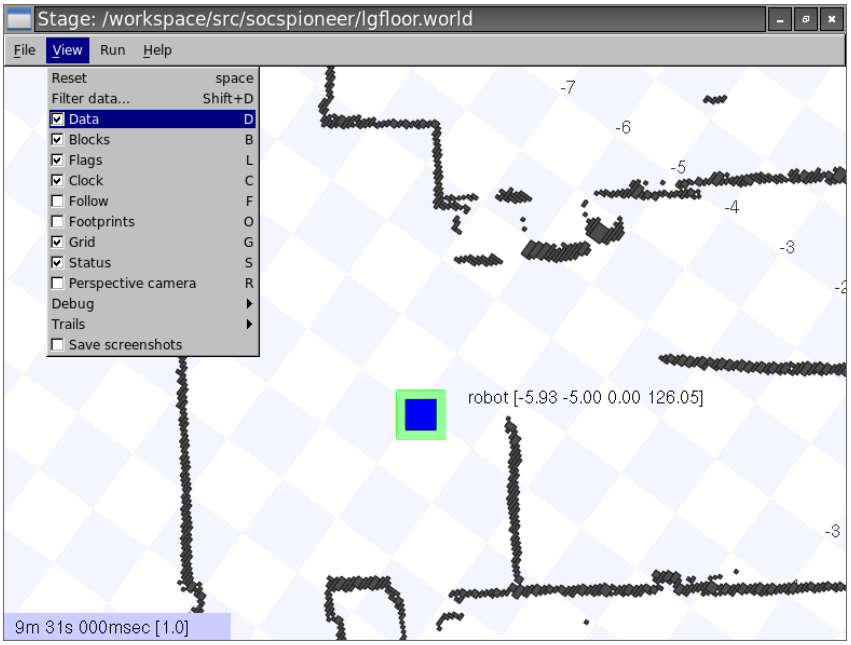
```
rosrun map_server map_server "socspioneer/lgfloor.yaml" &>/dev/null &
rosrun stage_ros stageros "socspioneer/lgfloor.world" &>/dev/null &
rosrun rviz rviz -d "/workspace/src/tools/docker/stage.rviz"
```

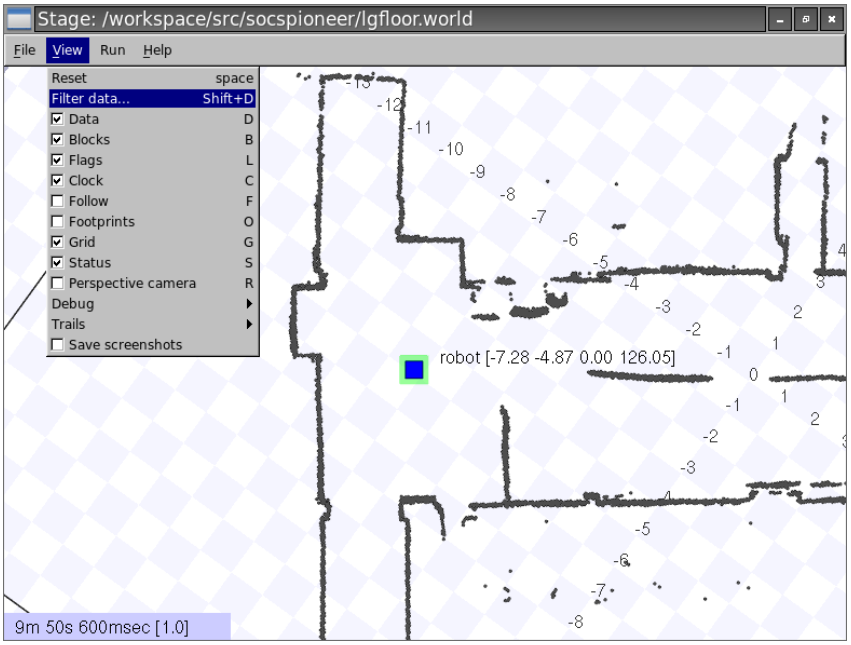
The map server is configured with a .yaml file and in this case uses the same occupancy map as the simulator. The map server allows you to view the map in rviz. The following node can be used to drive the simulated robot using the keyboard:

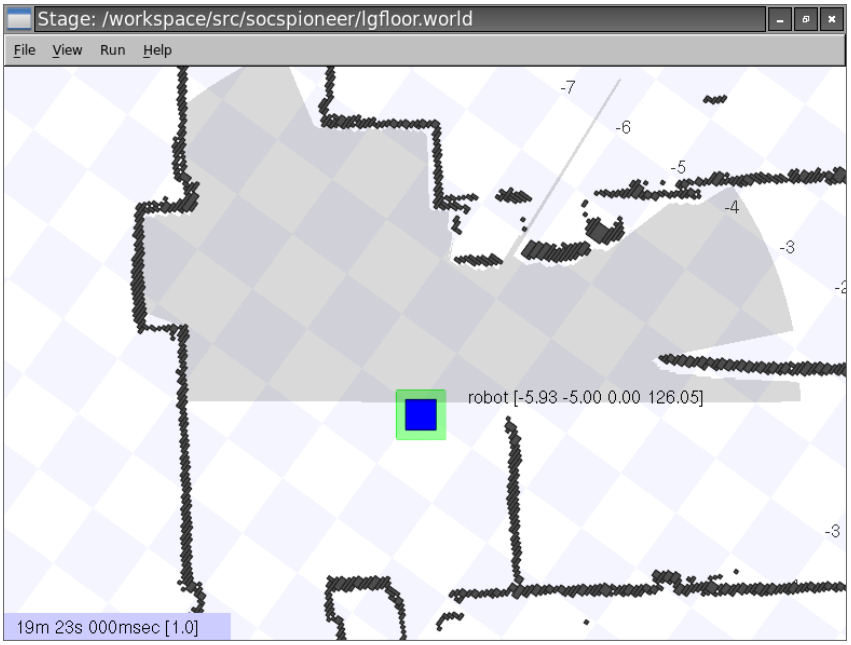
```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

A.3 Viewing Laser Ranges

To view the laser scanner data in stage perform the following steps:






Stage: /workspace/src/socspioneer/lgfloor.world
File View Run Help
-7
-6
-5
-4
-3
robot [-5.93 -5.00 0.00 126.05]
-3
19m 23s 000msec [1.0]

A.4 RQT

Rqt (http://wiki.ros.org/rqt) is a collection of various GUI tools (plugins) for working with ROS. It can be run with:

```
$ rqt &
```

Initially the window will be blank, but using the toolbar, various 'plugins' can be loaded into panels. For example, rviz is actually a plugin of rqt.

A.5 Transforms/Transform Tree

The transform tree viewer is useful for diagnosing problems with the ROS transforms which convert between reference frames (coordinate systems). For example there is a reference frame fixed at the laser scanner on the robot (usually called base_link, and another called base_laser_link only in the stage simulator) and one which is fixed at the origin of the map (usually called map, there is also a similar frame called odom which is based on the odometry information). Coordinates can be provided relative to any of these reference frames, but sometimes they have to be represented in another frame, which requires the ROS transform tree to tell the software how to transform between these frames. You don't have to interact with the transform tree directly, but various parts of ROS depend on the tree being correct.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

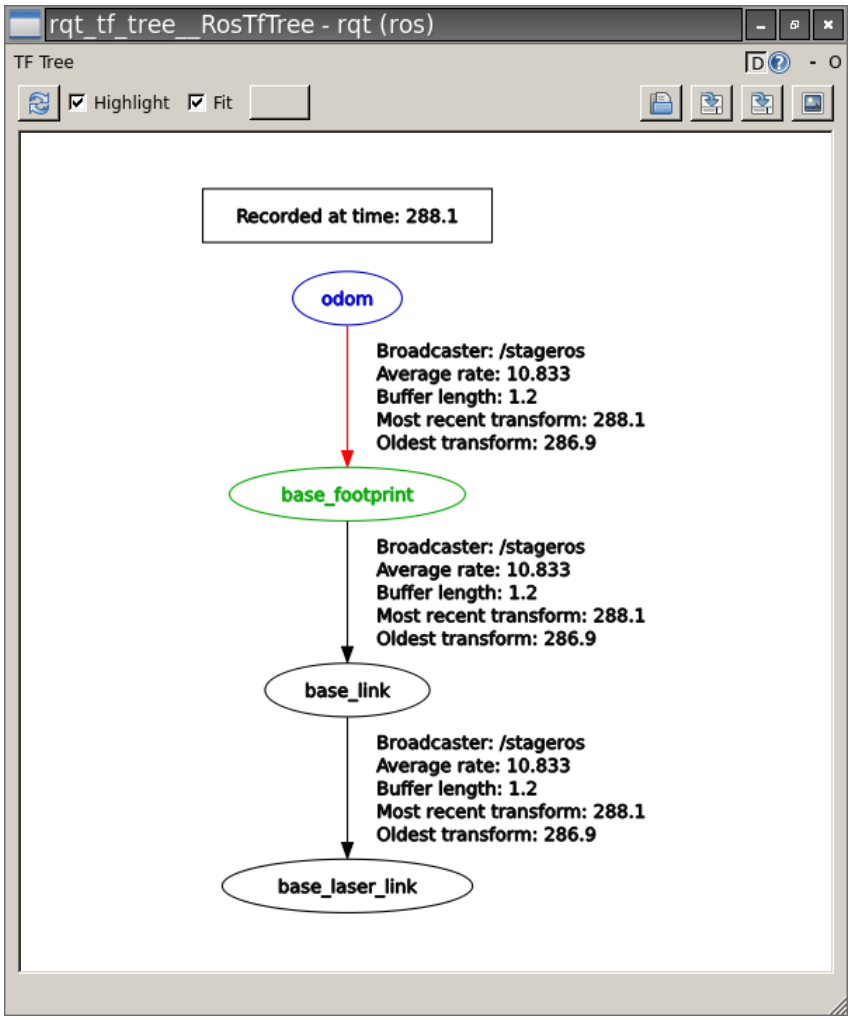
Below is an example transform tree when running in the stage simulator: Here are just some examples of transform issues that you might run into (in rviz).

Problem 1

There is no transform between the map (as provided by the map server) and odometry (provided by stage) frames. This is because the robot is not localised on the map and so doesn't know where it is. A hacky 'fix' is to introduce a node which supplies a fixed/static transform (which should be configured based on your particular situation). The better approach would be to run a node which localises the robot.

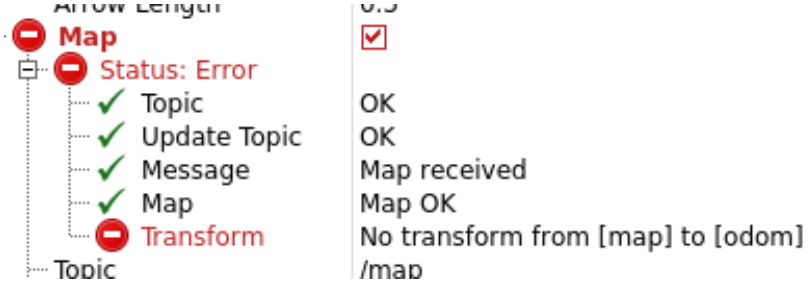
```
$ rosrun tf static_transform_publisher 0 0 0 0 0 0 1 /map /odom 100
or add to a launch file
```

```
<launch>
    <!-- http://wiki.ros.org/tf#static_transform_publisher -->
    <!-- arguments are:
        x y z qx qy qz qw frame_id child_frame_id period_in_ms
    -->
    <node pkg="tf" type="static_transform_publisher" name="odom_to_map"
        args="0 0 0 0 0 0 1 /map /odom 100" />
</launch>
```



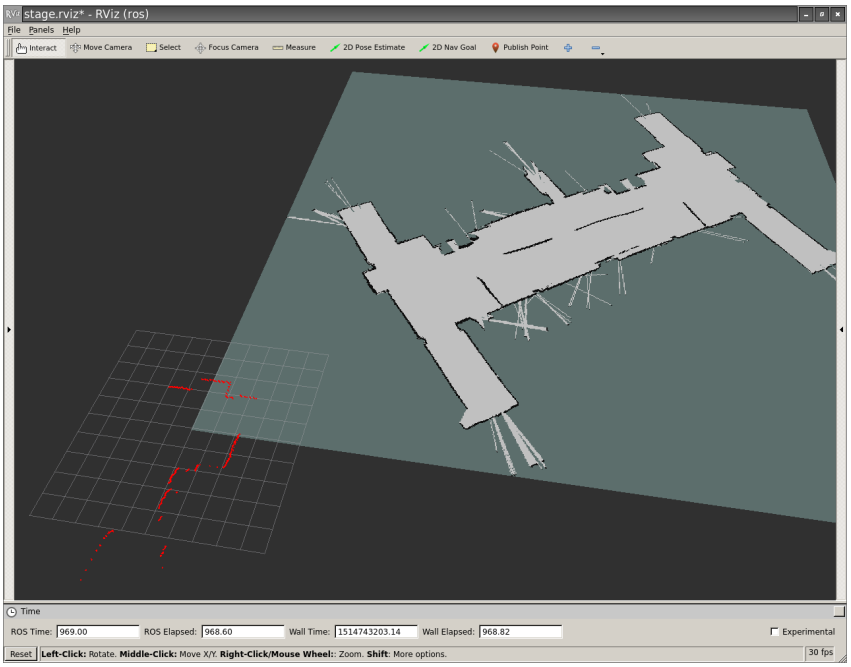
Problem 2

Rviz is currently configured to base the global view around the */map* frame, however it doesn't



exist in the transform tree. The map server provides an occupancy map in the /map frame but doesn't explain how it relates to the rest of the scene. When the robot is localised, e.g. using the navstack, then the localisation node publishes the transform between */base_link* (the robot) and */map*.

Problem 3 If transforms are present, but wrong, situations like this are common, where things



don't line up correctly or move in the right direction. Here you can see the laser scan data does not line up with the map.

A.6 Ubuntu System Installing

A.6.1 Virtual Machines

If you are not already using Ubuntu, a popular choice is to install virtual machines. The desktop image(.iso) for Ubuntu can be found here `https://cdimage.ubuntu.com/focal/daily-live/current/`. You need `focal-desktop-amd64.iso` or `focal-desktop-arm64.iso` according to your own machine.

If you are using Windows or other AMD64/Intel64 machine (like an old Macbook), check `https://www.virtualbox.org/` for download and `https://www.virtualbox.org/manual/UserManual.html` for installation and configuration. It should be easy to use and easy to find answers online when you encounter problems.

If you are using ARM machine (like M1/M2 chip Macbook), then the most convenient approach is to use `https://www.parallels.com/`. But unfortunately, this software is not free. To install 20.04, select "install Windows or another OS from a DVD or image file" and select the `.iso` file you just downloaded as shown in Figure A.1. Then choose all the recommend settings if you are not sure. This might take several minutes.

Also, you may not want to try ROS2 on M1 chip MacOS if you are not very confident with systems. Even though ROS2 says it supports Mac system officially, you have to compile everything from source. And we cannot guarantee everything runs smoothly in ROS2 even you manage to install it. But if you feel like challenging, here is something to get you start with:

https://sugae.github.io/envs/ros.html.

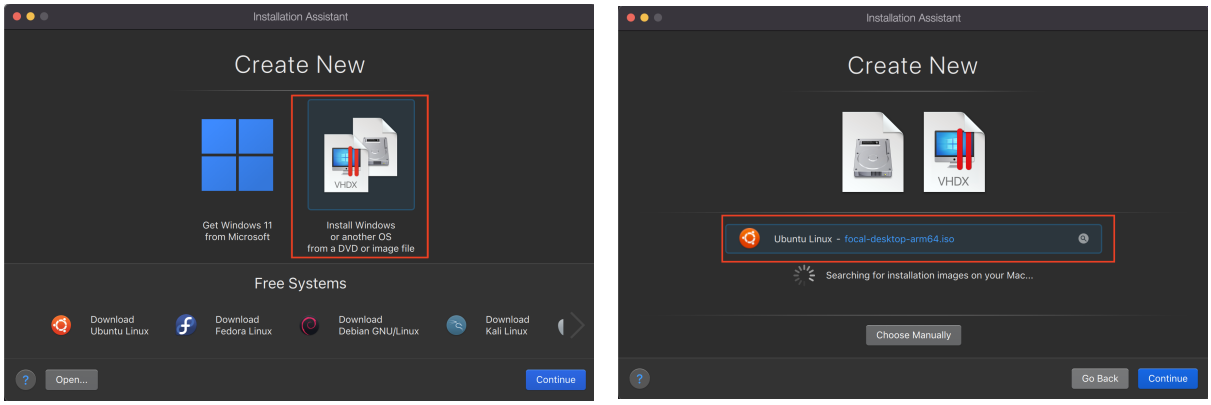


Figure A.1: Install Ubuntu 20.04 in Parallels.

A.6.2 Dual boot systems

Another option is to use Dual boot systems. You can search for plenty of step by step tutorials online. Please refer to A.6.3. However, we can not provide further assistance regarding this.

A.6.3 very helpful links

- very detailed guide on installing Ubuntu system on various machines provided by AFNOM: https://linux.afnom.net/
- PyCharm + Ros Setup Guide: https://youtu.be/lTew9mbXrAs