Lab sheet- Lesson 8 Intelligent mobile robotics
Assignment 2
Localisation of a robot in an Environment

Aims

- Install dependencies for mobile robot localisation
- Run a simulated environment and open a previously saved map
- Writing a python code to localise a robot in an environment

1. Install dependencies for mobile robot localisation.

- Note that this experiment should be done using ROS Noetic. When you create a project in ROS development studio.
- Run the following commands to install the dependencies. Note that you only need to install them once:

```
$sudo apt update
```

```
$sudo apt install ros-$ROS_DISTRO-pr2-teleop ros-$ROS_DISTRO-joy ros-$ROS_DISTRO-slam-gmapping ros-$ROS_DISTRO-map-server
```

- Build the package: Clone the following repo to the src directory of your catkin workspace.

```
$cd catkin_ws/src/
$git clone https://github.com/justagist/socspioneer.git
```

- Build the catkin workspace (you need to be in catkin_ws folder, using *$cd catkin_ws*):

```
$catkin_make
```

2. Run a simulated environment and open a previously saved map

- **NOTE: The catkin workspace should be sourced each time a new terminal session is loaded (run** source devel/setup.bash**). Alternatively, add the line** source <catkin_ws>/devel/setup.bash **to your** .bashrc **file to avoid repeating it every time.**

```
$ cd catkin_ws/
$ source devel/setup.bash
$roscore
```

Running rosrun stage_ros stageros <.world file> will start the simulator with a robot and the provided world:

```
$ cd catkin_ws/src/socspioneer/data
$ rosrun stage_ros stageros lgfloor.world
```

You should see a robot in a simulated environment. After opening the Graphical tools in Ros Development studio

  - Pressing R on the keyboard toggles between 2D and 3D views.

- D key toggles laser field of view visualisation.

- Run the following command to navigate the robot in the simulated environment. Note that when you push a key the terminal window running this comment should be focused (by clicking on the terminal window that run this command):

$roslaunch socspioneer keyboard_teleop.launch[1]

- 'map_server' is a ROS node that reads a map from the disk and offers it via a ROS service.

$cd catkin_ws/src/socspioneer/data

$ rosrun map_server map_server lgfloor.yaml

Use 'rostopic list' before and after running the previous command to see which topics create after running that comment.

- To see the map, run rviz:

$rosrun rviz rviz

  - Set the Fixed Frame to */map*.
  - Add map topic to be able to see the map

- Note that there is a built-in localisation package in ROS called AMCL that could be used to localise a robot. See Appendix A for more information. Note that you can not use the ACML package for your assignment.
- The next steps describe how to write your code for localisation.

3. Writing a python code to localise a robot in an environment

- Download the packed file on LZ called 'pf_localisation0.tar' and unpack it into your *~/workspace/src* directory.
- This package contains a ROS node 'node.py' which handles all the ROS backend stuffs, and an abstract class PFLocaliserBase (src/pf_localisation/pf_base.py), and incorporates 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 (see src/pf_localisation/pf.py)** which will extend PFLocaliserBase and provide localisation given a map, laser readings, and an initial pose estimate.
- In this experiment, you must complete the class provided in 'src/pf_localisation/pf.py'.
- To compile and run your node use the following command:

```
$ cd catkin_ws/
$ catkin_make
$ rosrun pf_localisation0 node.py
```

[1] If you use ROS Noetic and get error use the following comments to instal:
$sudo apt update
$sudo apt-get install ros-noetic-pr2-teleop

- Note that you need to make node.py an executable file (chmod +x node.py)
- After running the previous commands, you will see the results shown in appendix B. Go to the next step (step 4) to write your python code in ‘pf.py’ using the tips in step 4.
- We also use C++ in this project. There is not any extra task for this. See Appendix C for more information.

4. Tips to complete the class provided localisation in ‘src/pf_localisation/pf.py’, i.e. PFLocaliser

4.1. Constructor (__init__(self))

PFLocaliser is a subclass of PFLocaliserBase. The constructor of PFLocaliser, as when implementing any subclass, calls the superclass constructor.
Your constructor for PFLocaliser needs to assign the following 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 *src/pf_localisation/pf_base.py:PFLocaliser:predict_from_odometry* function. An example value is 0.1.
You'll then need to implement the following three abstract methods:

4.2. initialise_particle_cloud(self, initialpose)

- Called whenever a new initial pose is set in *rviz*.
- This should instantiate and return a *PoseArray* [3] object, which contains a list of *Pose* objects. Each of these *Pose*s 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 much more complex than you will need as the Pioneer only rotates around the yaw axis, so to make it easier for you, you have been provided with the following methods for handling rotations in 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)*

      Performs the reverse conversion and gives you the heading (**in radians**) described by a given Quaternion q.

4.3. update_particle_cloud(self, scan)

- Called whenever a new LaserScan message is received. This method does the actual particle filtering.

- The PFLocaliserBase will already have moved each of the particles around the map approximately according to the odometry readings coming 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. .poses* using the **self.sensor_model.get_weight**(scan, pose) 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 probabilities (weights) 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. (you could use the method used for random selection in RANSAC). **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.

4.4. estimate_pose()

- This should return the estimated position and orientation of the robot based on the current particle cloud, i.e. self.particlecloud.

  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 wich is a ronge degre, It should be 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.

4.5. Important advice!

- As well as seeing your code running as you drive your robot along a specific route which we'll give you, we will look for evidence that you have experimentally (i.e. in the form of records in your log books) investigated how the particle filter behaves, including (but not limited to): adding noise to the particle cloud, visualising the sensor model function

    - 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 use *rostopic echo <topic>* to listen to the messages which are being passed between all nodes, including yours. The relevant topics are:
    - *scan* -- laser scans coming in
    - *initialpose* -- the initial pose estimate created when you use rviz to set the robot's location
    - *map* -- the map
    - *amcl_pose* -- the estimated pose calculated and published by your node
    - *particlecloud* -- your node's cloud of particles created by the particle filter
- 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 doesn't call your *PFLocaliser* update methods until odometry update messages have been received (i.e. it doesn't update automatically whenever there is a laser scan, **only when the robot has actually moved**). So you will need to run a ***keyboard_teleop*** launch file,i.e. *$roslaunch socspioneer keyboard_teleop.launch*, and drive the robot around before your particle cloud will appear on the map in *rviz*.
- Rotations around the compass are in radians, as shown. 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: **using a built-in package called AMCL to localise the robot**

ROS comes with a built-in localisation package called AMCL. To see how localisation within ROS should work, first, you should get the AMCL localisation software up and running:

Note that you could not use this built in package for your assignment.

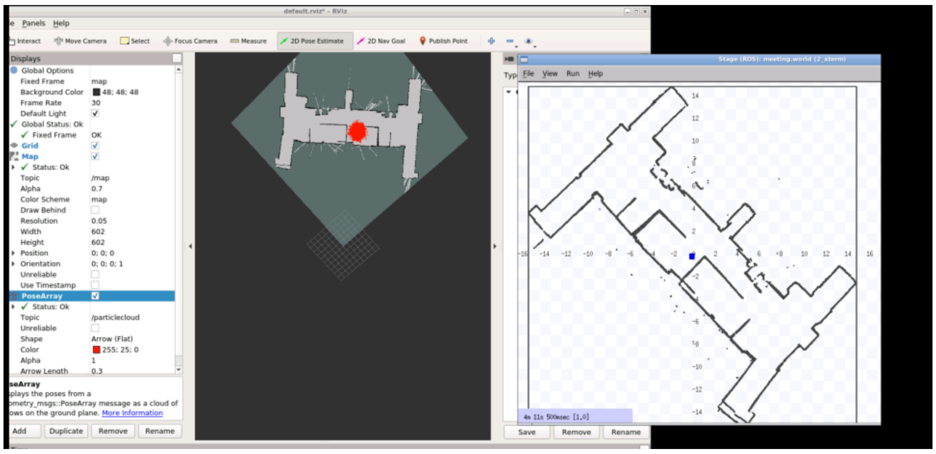
To run this bult in ACML package when you perform step 2 after running the '$rosrun rviz rviz', do the following:

- Run the AMCL localisation package:

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

    - Now we want to be able to visualise the localisation after running the rvis using `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.
    - (see the video for more information)

- Activate the window that you have run ‘roslaunch socspioneer keyboard_teleop.launch’ to move the robot in the environment.

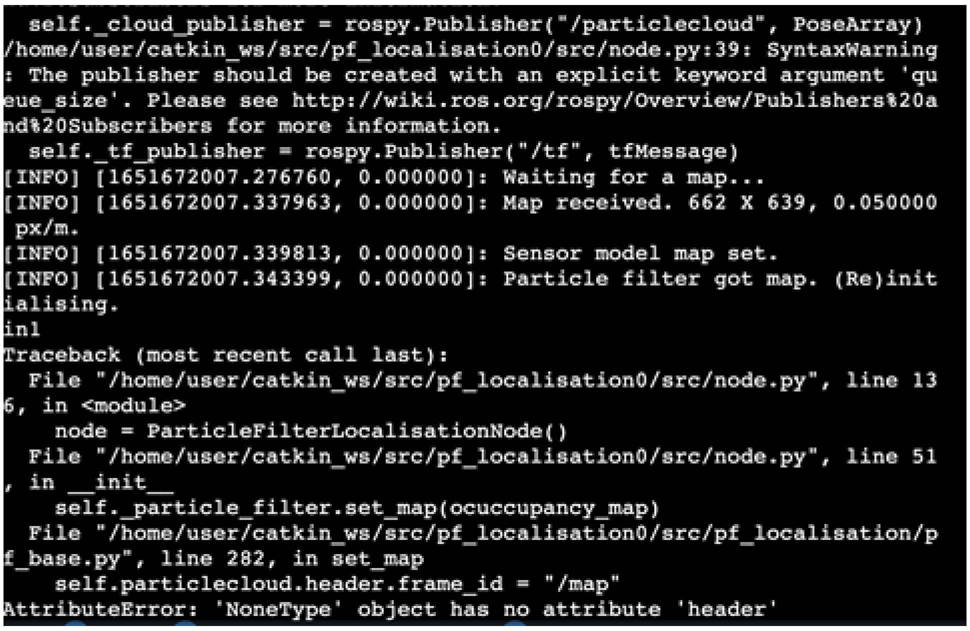


```
Displays
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
  Default Light        ✓
✓ Global Status: Ok
  ✓ Fixed Frame        OK
Grid                   ✓
Map                    ✓
  ✓ Status: Ok
  Topic                /map
  Alpha                0.7
  Color Scheme         map
  Draw Behind
  Resolution           0.05
  Width                602
  Height               602
  Position             0; 0; 0
  Orientation          0; 0; 0; 1
  Unreliable
  Use Timestamp
PoseArray              ✓
  ✓ Status: Ok
  Topic                /particlecloud
  Unreliable
  Shape                Arrow (Flat)
  Color                255; 25; 0
  Alpha                1
  Arrow Length         0.3
seArray
splays the poses from a
ometry_msgs::PoseArray message as a cloud of
ows on the ground plane. More Information
```

- If you want to write your code forlocalisation stop runing the command ‘rosrun amcl amcl scan:=base_scan’ using ‘ctrl+c’, and follow the step 3

Appendix B: **The output of step 3 before writing your code in pf.py**

At the end of step 3 you will get the following output:



```
  self._cloud_publisher = rospy.Publisher("/particlecloud", PoseArray)
/home/user/catkin_ws/src/pf_localisation0/src/node.py:39: SyntaxWarning
: The publisher should be created with an explicit keyword argument 'qu
eue_size'. Please see http://wiki.ros.org/rospy/Overview/Publishers%20a
nd%20Subscribers for more information.
  self._tf_publisher = rospy.Publisher("/tf", tfMessage)
[INFO] [1651672007.276760, 0.000000]: Waiting for a map...
[INFO] [1651672007.337963, 0.000000]: Map received. 662 X 639, 0.050000
 px/m.
[INFO] [1651672007.339813, 0.000000]: Sensor model map set.
[INFO] [1651672007.343399, 0.000000]: Particle filter got map. (Re)init
ialising.
in1
Traceback (most recent call last):
  File "/home/user/catkin_ws/src/pf_localisation0/src/node.py", line 13
6, in <module>
    node = ParticleFilterLocalisationNode()
  File "/home/user/catkin_ws/src/pf_localisation0/src/node.py", line 51
, in __init__
    self._particle_filter.set_map(ocuccupancy_map)
  File "/home/user/catkin_ws/src/pf_localisation0/src/pf_localisation/p
f_base.py", line 282, in set_map
    self.particlecloud.header.frame_id = "/map"
AttributeError: 'NoneType' object has no attribute 'header'
```

You need to complete the code in ‘pf.py’ based on the tips described in step 4.

Appendix C: calling a C++ function in python in ROS:

- We used a function written in C++. The original C++ code is in src/ laser_trace folder. A ‘laser_trace.so’ file (shaired object) created from the `laser_trace.cpp` file using the following coment:

```
g++ -shared -fPIC -o laser_trace.so laser_trace.cpp \
    -I/usr/include/python3.8 \
    -I/usr/lib/python3/dist-packages/numpy/core/include \
    -lboost_python38 \
    -lpython3.8
```

- Then the created file was moved to the folder that the python codes are there. See the src/ pf_localisation/laser_trace.so'
- The laser_trace was imported in ~/catkin_ws/src/pf_localisation0/src/pf_localisation/sensor_model.py and used in the python code.

Appendix D: summary of the comments:

-install:
sudo apt update
sudo apt install ros-$ROS_DISTRO-pr2-teleop ros-$ROS_DISTRO-joy ros-$ROS_DISTRO-slam-gmapping ros-$ROS_DISTRO-map-server

-install
cd catkin_ws/src/
git clone https://github.com/justagist/socspioneer.git
catkin_make

-install ros-noetic-pr2-teleop:
sudo apt update
sudo apt-get install ros-noetic-pr2-teleop

---------run roscore-------------------
cd ~/catkin_ws/ && source devel/setup.bash && roscore

---- open robot simulated environment
cd catkin_ws/src/socspioneer/data && rosrun stage_ros stageros lgfloor.world

------------run key operator:
source /opt/ros/noetic/setup.bash
source ~/catkin_ws/devel/setup.bash

roslaunch socspioneer keyboard_teleop.launch

--------open map:
cd ~/catkin_ws/src/socspioneer/data && rosrun map_server map_server lgfloor.yaml

rosrun rviz rviz

----------run python code after uploading python code:

rosrun pf_localisation0 node.py

References:

[1] **map_server**
http://wiki.ros.org/map_server
[2] **The SOCSPIONEER Package**
https://github.com/justagist/socspioneer
[3] ROS /PoseArray

http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/PoseArray.html