Nicolas ISNARD

Internship report

from 12/06/2017 **to** 11/08/2017
at
The University of Birmingham

Development of a two mobile robots simulation under ROS environment using regions of interest and Markov's chains to copy and predict probabilistically a natural human pattern of movement

4th year technical internship for the Robotics engineer diploma
at ***Polytech Paris UPMC***
2016-2017

Résumé (en français):

Durant ce stage technique de deux mois, j'ai travaillé au sein du laboratoire de robotique de l'Université de Birmingham. J'ai été assigné à une partie de developpement de simulation de comportement de robot autonome du projet STRANDS (Spatio-Temporal Representations and Activities for Cognitive Control in Long-Term Scenarios). Ce projet inter-universitaire propose de développer des comportements à des robots mobiles d'accueil sur de longues périodes d'autonomie.

Le début du stage a d'abord consisté à s'approprier les outils et l'environnement de travail utilisé pour ce projet qui sont inclus dans un package conséquent: strands-morse-simulation. Ce package contient la mise en commun des outils, codes et données des différentes universités qui travaillent sur ce projet. On y trouve notamment des modèles de robots et de laboratoires à intégrer dans une simulation Morse et RVIZ. C'est pour moi la première approche de l'environnement ROS et je me forme grâce aux tutoriels du ROS wiki et de la plate-forme STRANDS. Ainsi, j'apprends à construire une carte 3D par l'intermédiaire des logiciels Gimp, Inkscape et Blender. Cette carte peut alors être exploité dans l'application de simulation 3D Morse. Afin de sauvegarder les données des cartes et leurs contenus, on utilise l'outil de gestion de base de données MongoDB. Enfin, on peut réunir toutes ces informations et les afficher dans l'application de simulation RVIZ. Les tutoriels fournissent une simulation basique qu'on peut tester et dont on pourra s'inspirer pour la suite.

Le corps du stage consiste à présent à développer notre propre simulation. A terme, notre simulation doit comporter deux entités autonomes:

- Une entité Humain qui se déplace sur la map en choisissant sa destination suivant des probabilités (définies dans une chaîne de Markov).
- Une entité Robot qui va essayer d'identifier, suivre la première entité en utilisant ses capteurs et essayer de deviner où elle se dirige.

Dans la suite, on modélise l'entité Humain par un robot pour que le modèle reste simple et moins lourd qu'un modèle d'humain. On commence donc par faire notre script de simulation Morse pour définir l'environnement où va évoluer les robots. On doit alors développer un nouveau script qui récupère les données des régions d'intérêts de la carte. On extrait les informations de chaque points définissant les régions de la base de données pour calculer les centres de chaque région et construire la chaîne de Markov. Etant dans une certaine région, la chaîne de Markov définit une probabilité de déplacement vers chacune des régions connectées. Une fois qu'on a cela, on peut alors commander le robot pour son déplacement entre les régions.

Pour la fin du stage, on devrait utiliser un code de détection des humains déjà développé mais qui ne fonctionne pas en simulation. Ainsi, la partie qui consiste à détecter la direction du déplacement d'un humain et en deviner la destination par l'intermédiaire de la chaîne de Markov sera donc développé directement sur le robot Scitos A5 du laboratoire par mon collègue Antonin. Pour ma part, je m'occupe de la sauvegarde des trajectoires du robot dans la database MongoDb et leur affichage sur Rviz.

Thanking:

First of all, I want to thank Nick Hawes that allowed me to do this internship in the CS department of the University of Birmingham. He gave me the liberty to choose a robotics subject that really interested me and has been attentive when we needed him.

Then, I thank Ferdian Jovan for the patient help, the time and the guideline he provided me during this internship. He has been a very good support and a solution source when I was blocked.

Finally, I want to thank my colleague Antonin Haulot-Logre that worked with me during this internship for the optimism and motivation he can spread around even when your are blocked for days.

Index

I. Introduction

During this internship, I worked on a simulation that is a part of the STRANDS project (Spatio-Temporal Representations and Activities for Cognitive Control in Long-Term Scenarios). This project is an inter-university program that consists in developing intelligent mobile robots that can run in dynamic human environment for months. These autonomous robots would be able to act like assistants extracting the information they need from the world.

In this context, I've been asked to help a PhD student Ferdian Jovan in his work about the adaptation of autonomous robot in moving human environments. This work is part of the STRANDS project in a way that we need to give the robot the opportunity to recognize pattern in an human movement, register it and try to predict the human behaviour thereafter. The global purpose is to give the tools to the autonomous robot for understanding the world around it with the minimum of information.

My task here was to develop a simulation with two entities that could interact. The first entity should be a human that moves in the simulation map according to a specific kind of pattern : a Markov chain. A Markov chain is a set of connected regions of interest in the map where each connexion between two regions has a probability. This probability defines the chance for an human to be likely to want to go by this connexion and reach the other connected region. The goal of the Markov chain here is to define a pattern of behaviour close to natural human behaviour for the human entity in the simulation. We have decided here to use a robot for the human entity so that it lightens the simulation. The second entity is a robot that is supposed to get the trajectory of the human entity, record it and try to predict the human entity behaviour and his destination.

In order to present my work in this document, I will start with a presentation of all main tools we have to use for running this kind of simulation. Then, I will describe what has been necessary to set up on my computer and what I have done to handle the required tools, understand and build a simulation. Afterwards, I will talk about the main simulation I have build during this internship to study the human behaviour with a Markov chain. Thereafter, I will explain the storing and visualization of trajectories part. And finally, I will conclude this report with a summary of my contributions in this project and the perspectives to consider in the future.

II. Tools presentation

In the first place, we are going to introduce all the tools and applications that we use here. To start, there some needed features that you must have on your computer if you want to run the kind of simulation using ROS (Robot Operating System). It's recommended to have a good computer with a fine processor (i5 and more) and graphic card (Nvidia). Your computer must run under Ubuntu 14.04. It's not the newest version of the environment but it's more stable and support the ROS middleware. This middleware allows us to use many simulation tools. As far as we are concerned, we will use here MORSE (Modular OpenRobots Simulation Engine) and Rviz (ROS visualization).

MORSE is an generic simulator for academic robotics. It generates realistic 3D simulations of small to large environments, indoor or outdoor. This software permits us to build a simulation corresponding to the robotics laboratory at the level -1 of the Computer Science building in the University of Birmingham. Simple python scripts are sufficient to describe robots and environments in the simulation. This software uses Blender Game Engine that includes several lighting options, multi-texturing and libraries that deals with the physics simulation. As for us, we will use MORSE to generate our two robots simulation at the laboratory level.

Rviz is a 3D visualizer for displaying sensor data and state information from ROS. We use that software for different reasons. The first usage is to visualize the content of the topics that our MORSE simulation displays. But we also use Rviz to define polygons that represents the regions of interest on the map loaded from a database. All the regions of interest (roi) are stored in the database continuously with the map.

But we also need an interface to handle the database where we want to store all our data. In order to do that, we use the software Robo 3T (formerly robomongo) that uses MongoDB, a database document to store and manipulate JSON-like documents. It's a NoSQL database easy to handle.

Those three softwares are the central points of the simulation we want to develop during this internship. But the process is more complex and implies other little modules provided by ROS such as amcl for the robot localisation or move_base for the movement. We will give further explanations on the contribution of each module later.

III. Tutorials and getting started

A. Ubuntu 14.04 LTS and ROS indigo

The first thing to do is to install Ubuntu 14.04 LTS on our computer. It's not the newest version but this one is more stable and support ROS. By the way, with a newer version it's more difficult to download strands_morse package because the automatic installation works only with 14.04 so you would have to make the download and installation manually.

Despite of being the most stable version of Ubuntu, usually it still has problems with numerous computers. So, according to my experience, I will describe here the problems I have encountered and the solutions I have found.

Once Ubuntu 14.04 installed, the first problem and the most annoying I encountered is a wifi problem. My computer was unable to connect to the wifi longer than 30 seconds when starting the computer. It is very embarrassing when you want to search a solution on the internet.

The problem was that I didn't have the right driver for my wifi card and my computer was even unable to recognize the model of it. Usually, to identify your wifi card you need to do the command: "**lspci | grep -i net**". But in my case, it only makes crash the terminal so I just searched on internet what was my wifi configuration on a documentation of my computer. According to that, I downloaded new drivers for my RealTek wifi card on a GitHub (https://github.com/lwfinger/rtlwifi_new) and I have to recompile and reinstall it each time there is a kernel modification due to an update of Ubuntu. When it happens the commands are: “**make**” and “**sudo make install**” in the rtlwifi_new folder.

The same type of drivers problem is common with graphic cards especially with separate graphic cards like Nvidia. If there is a problem with that, Ubuntu has a page to fix that problem (https://doc.ubuntu-fr.org/nvidia) and if everything works fine Ubuntu will be working properly after that.

When we are done with Ubuntu installation, we can now install ROS indigo following the tutorial that can be found on ROS site (http://wiki.ros.org/indigo/Installation/Ubuntu). Then, it’s ready to use and have fun. The best way to handle ROS middleware is to start by doing the tutorials they offer on their site (http://wiki.ros.org/fr/ROS/Tutorials). It allows to understand how ROS works and is structured (launch, node, topics…) and how to use basic commands (roslaunch, rosrun, catkin_make, roscd ...). We won’t talk about those tutorials here because it’s not the purpose of this document but it’s really recommended to do it anyway. Spending time in those tutorials will make you save time for the future.

B. Terminator

To simplify our utilisation of ROS, we need to download a virtual terminal so that we can separate, visualize and organize several terminals in a window. This software includes some very useful shortcuts that will make you save time and give clarity when you have to launch a lot of scripts.
The most used shortcuts are :

- Ctrl-Shift-E : Split window v**e**rtically.
- Ctrl-Shift-O : Split the window h**o**rizontally.
- Ctrl-Shift-N ou Ctrl-Tab : Move the cursor to the next window.
- Ctrl-Shift-P ou Ctrl-Shift-Tab : Move the cursor to the previous window.
- Ctrl-Shift-W : Close the current window.
- Ctrl-Shift-X : Enlarge the current window.
- Ctrl-Shift-Arrow : Enlarge or shrink the current window.
- Ctrl-Tab : Change window "split".

C. Strands-Morse tutorials

1. Installation

Now that we have our work environment fully setup, we can download the strands_morse package. It’s the package of every codes and data shared between each university participating in the STRANDS project. We can find several example of simulation with Scitos A5 robot in different locations and with different behaviours. So let’s start with the package installation. Everything is explained on the internet page of the strands-project installation (https://github.com/strands-project-releases/strands-releases/wiki). With Ubuntu 14.04 LTS, it’s quite simple because it only needs to add the public key and the repository of

strands-project with “**curl -s http://lcas.lincoln.ac.uk/repos/public.key | sudo apt-key add -**“ and “**sudo apt-add-repository http://lcas.lincoln.ac.uk/repos/release**”.
Then, it just needs to update the index with “**sudo apt-get update**” and add any package that will be useful with “**sudo apt-get install <pkg-name>**”.

2. A simple simulation

Now that it’s done, the best way to handle the package is to follow the tutorial of the basic_example simulation (http://strands.readthedocs.io/en/latest/setup.html). This complete tutorial gives basis to handle every applications that runs this type of simulation.

First, the tutorial introduces to us the database Robomongo (MongoDB) that is used to store maps and everything we want for our simulations. Then, we can use the GIMP (GNU Image Manipulation Program) to create a basic 2D map just drawing the walls with straight lines. We have to use Inkscape,a free vector drawing software, to export this image in vectorial format. Next, we can transform this 2D map in a 3D map with Blender, a free 3D modeling, animation and rendering software. Once the 3D map is built, we can start to use ROS to have our first simulation. We start with :

“**roslaunch basic_example basic_example.launch**”

It launches a basic simulation with a Scitos A5 robot that can be controlled with the keyboard. It’s a Morse basic simulation in a Blender window. But to have the data from sensors like a lidar and remap from this, we need to launch another application named gmapping with:

“**rosrun gmapping slam_gmapping scan:=scan**”

It allows to build a new map from the data of the lidar scan of a robot we can find in the “*scan*” topic. We can now visualise those data and more in Rviz with:

“**roslaunch basic_example basic_example_rviz.launch**”

It’s now possible to move the robot, explore the map, make a new map from the scan and save it on the a map_server with:

“**rosrun map_server map_saver -f maps**”

Then, we can display the map we just saved on the map server with:

“**rosrun map_server map_server maps.yaml map:=mymap**”

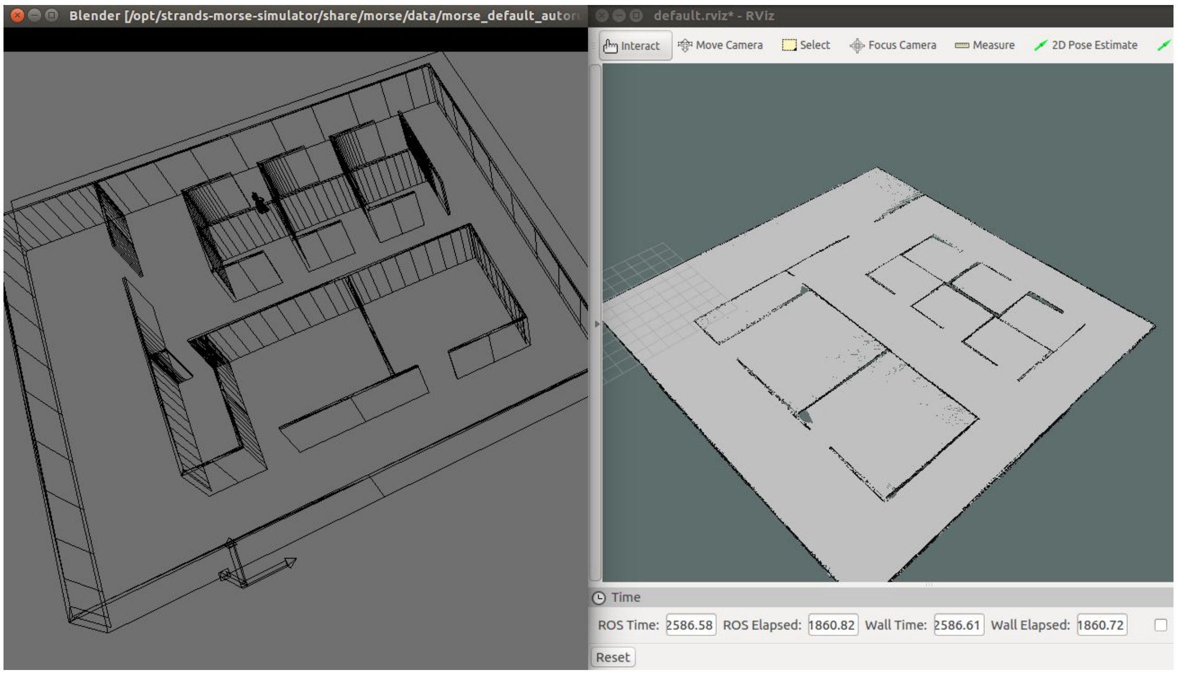
This command grabs our map in yaml format and in this file, we can find the path to find the map.png associated. It publishes the map on the topic /mymap and we can visualise the result in Rviz doing ‘add’ in the interface to see the corresponding topic. You can find the result on the Figure 1 after next paragraph.

The map server is quite useful but, thereafter, we will use an equivalent from SOMA (Semantic Object Maps Application) package with the MongoDB database that is more convenient when we have to manipulate Objects with the map.
So, we use mongoDB with those main commands:

“**roslaunch mongodb_store mongodb_store.launch db_path:=/home/isnard/Documents/my_database_dirj**” : to launch the database

“**rosrun topological_utils load_yaml_map.py ~/catkin_ws/src/basic_example/maps/maps.yaml**” : to load a map in the database

“**rosrun topological_utils list_maps**” : to list the available maps



ROS Time: 2586.58 ROS Elapsed: 1860.82 Wall Time: 2586.61 Wall Elapsed: 1860.72

Figure 1: Blender 3D simulation (left) and Rviz map visualisation (right)

IV. Multiple environment simulation

By multiple environment simulation, we mean here that the following simulation needs to be running in several different applications. Each applications has a specific purpose and a specific contribution for the simulation running. We are going to describe here the main simulation that we have built during this internship taking each applications in cause separately.

A. RVIZ, Regions of interest and Database

1. General launching

The first step of the simulation is the map construction with all the components and the content of it. We use Rviz because it has a particular type of markers named *InteractiveMarkers* that can be used to define the regions of interest we want on our map. Those markers build polygons that can be moved and shaped as many time as we want.

Moreover, we will need to store our regions in the database so we will use MongoDB and an additional package named SOMA (Semantic Object Maps Application) that will handle the manipulation of our objects to store. We have to keep the modifications possible at any time in Rviz. In order to do that, we need 4 commands:

"**roslaunch mongodb_store mongodb_store.launch db_path:=/home/isnard/Documents/database**" : that initializes the database mongoDB. This is the first thing to do for our simulation.

"**roslaunch soma_manager soma_local.launch map_name:=map**" : that launches the interface handling the manipulation of objects to store in the database. We have to specify the map topic with "**map_name:=map**". We notice here that it's a relative topic *"map"* and not an absolute topic like *"/map".*

"**rosrun soma_roi_manager soma_roi_node.py 1**" : that launches the node taking care of the regions of interest (roi). It's a manager that keeps the modifications and updates possible as we want. We have to specify a map configuration at the end, here our map configuration is "*1*".

"**rosrun rviz rviz**" : that simply launches Rviz so that we can start to visualize the simulation and define our roi.

In order to simplify the utilisation, we wrote a script *2Robots_simu_soma.sh* that launches the different commands in a row with different terminals for each command. To do that, we can use the bash commands :

"**xterm -e "your command here" &** " : that launches a new terminal xterm with the command in quotes. We keep the hand for the next command in the script with "*&*". We do that after each command except the last command.

"**sleep 1s** " : that is used to temporize the launching of each command so that the simulation can settle down.

The first advantage is that we don't have to tape the command and execute them in new terminals each time. But the second advantage is that when we want to close all the terminals, we can do it by closing only the terminal where we have launched the general script (don't forget NOT to put the "&" at the end of the last xterm command).

2. Regions of interest definition

With that structure, we can easily define our regions of interest. For the rest of this part, we have to take into account the future usage of zone markers. Indeed, we will use the regions represented by polygons to define a Markov chain. So that means defining a map with each region of interest, every connexion with the neighbor regions and the probability for a human to go by each connexion.

We had to think about a solution in order to give to the code the maximum amount of information through the roi definition. The most simple solution is to develop a dynamic generation of the Markov chain based only on the roi definition. The main advantage is the rapidity of the method because we don't have to define manually all the connexions and their probabilities for each roi. It's also convenient because if we want to start over the Markov chain or completely change the map, we just have to define the roi in Rviz and the code will do the rest.

So, with this method, the important thing is to keep in mind that all the information needed for the Markov chain generation is defined during the roi definition with Rviz. There is a specific method to define that correctly in the way that the code can extract the information needed. I made this code that extracts the informations so I will describe here how it roughly works for you to understand how to use it properly. Then, I will explain more closely in Movement simulation part. This code is in *robot_human_control_morse.py* in my package *multiple_robots_simu* that contains the main part of my internship work.

The code will extract each position of each point of the roi we have defined in Rviz and store in MongoDB. The first thing we want is to get all the connexions between each roi. The best method we found is to carefully place common points of the regions' polygons to mean that there is a connexion between them. In doing so, the code will search for common points between the roi and define a connexion as soon as he finds at least one.

Afterwards, the code will deduce the number of connexion that has each roi and this is also an important piece of information. The fact is that, being on the definite roi, we have now to determine a probability for a human to go by each connexion. In order to make that decision, I have based my reflexion on the fact that a person moving in a building is less likely to want to stay in very connected areas (like hallways) than few connected areas (like offices). According to that logic, we can make a relation between the roi's number of connexions and the probability for a human to want to go by this roi. Indeed, the code will generate probabilities for a connexion to a roi that is proportional to the number of connexion it has.

To sum up, when we define the region of interest in Rviz at the beginning of the simulation (Figure 2), all information for the Markov chain is contained in that and we have to keep in mind two particularities:

- To define a connexion between two regions of interest, make a common point among their polygons points. You just have to superpose one point from one polygon with one from the other one.
- Keep in mind that less connected areas are supposed to be offices where a human is likely to stay and very connected areas are supposed to be hallways. This particularity is more like a note because it's usually naturally the case when we define the roi without thinking about that. But it's important to mention it when the map has a tricky topology.

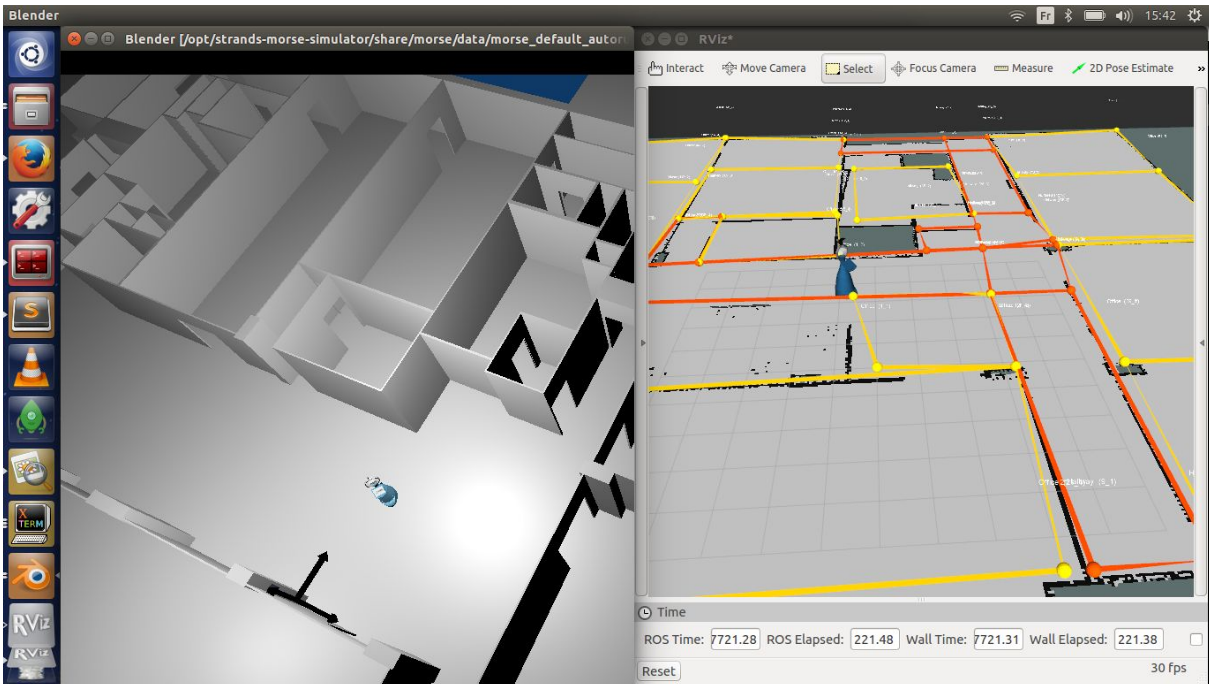


Figure 2 : Blender simulation (left) and Rviz visualisation with the regions of interest (right)

As we seen in the General Launching part, the first thing to do is to initialize the database MongoDB. This is the first thing to do because we have to be able to access to the data of the roi as soon as the simulation start. Otherwise, the generation of the Markov chain will be empty or incomplete. If we want to modify the roi, we can do it at any time in Rviz but the changes in the Markov chain will be taken into account only when we will restart a new simulation.

B. Morse simulation

1. Structure

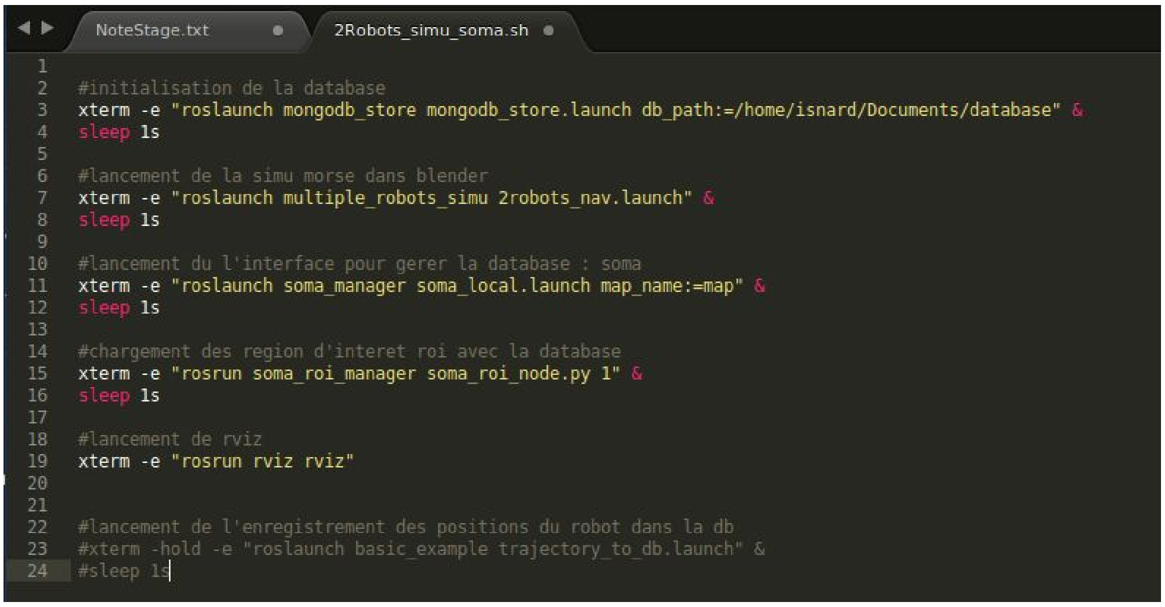
For this part, I am going to describe the simulation structure I have built with Morse simulation. This is the core of the simulation and it mainly consists in 2 python codes *construction_map.py* and *robot_human_control.py*. I have two versions of this structure of 2 codes (respectively *construction_map_morse.py* and *robot_human_control_morse.py*) because I had to try two different movement approaches that I will present later.

The first one is responsible for the definition of the simulation. We can find in this code the map, its content and the robots with all their sensors, actuators and ways to be communicating with the rest of the simulation (sockets, ROS topic …).

The second one consists in the definition of the behaviour of the simulation through the python class *Robot_human_control.py* . We have here the extraction of the roi data from the database, the generation of the Markov chain and its storage and, of course, the order of movements for the robots.

The structure for the launching of the two versions I have developed are the same because there is just different names for the different python codes called. First, there is the global script *2Robots_simu_soma.sh* that contains all the launches we have already told about in the General Launching part (A. 1/). We have to add one single command that calls another *.launch* created to launch the 2 python codes (*construction_map.py* and *robot_human_control.py*). This command must be run in a new terminal like the others and we always put a delay to let the simulation settle down. So, we obtain the almost complete script on Figure 3 for our simulation adding this command at the others :

"xterm -e "roslaunch multiple_robots_simu 2robots_nav.launch" & sleep 1s"

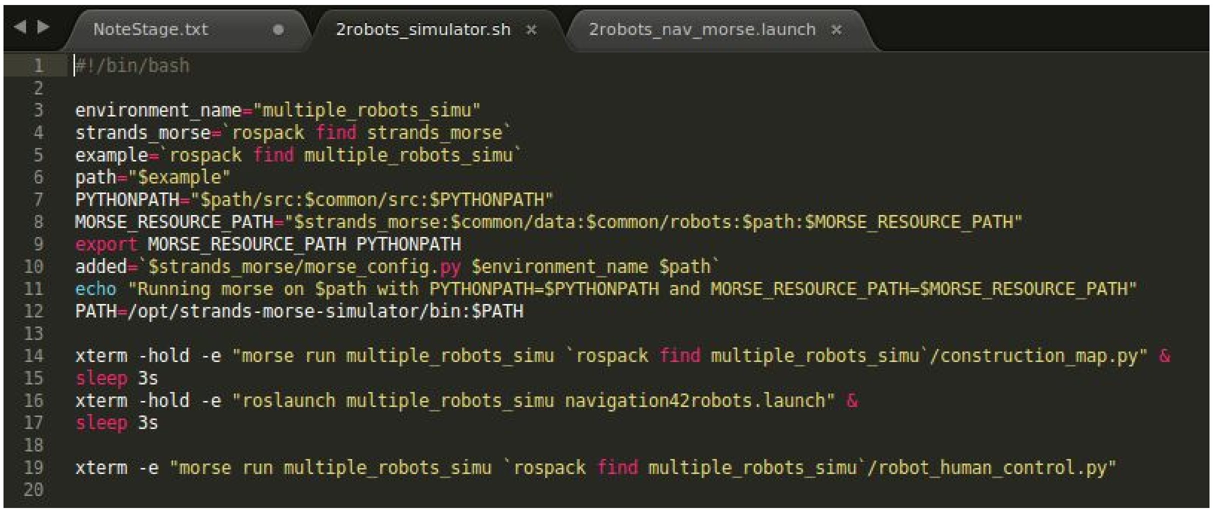


```

#initialisation de la database
xterm -e "roslaunch mongodb_store mongodb_store.launch db_path:=/home/isnard/Documents/database" &
sleep 1s

#lancement de la simu morse dans blender
xterm -e "roslaunch multiple_robots_simu 2robots_nav.launch" &
sleep 1s

#lancement du l'interface pour gerer la database : soma
xterm -e "roslaunch soma_manager soma_local.launch map_name:=map" &
sleep 1s

#chargement des region d'interet roi avec la database
xterm -e "rosrun soma_roi_manager soma_roi_node.py 1" &
sleep 1s

#lancement de rviz
xterm -e "rosrun rviz rviz"


#lancement de l'enregistrement des positions du robot dans la db
#xterm -hold -e "roslaunch basic_example trajectory_to_db.launch" &
#sleep 1s
```

Figure 3 : General Launching script with simulation launch added

Indeed, this .launch is in the launch folder in the *multiple_robots_simu* package we have created. It contains only this few lines :
<launch>
<node pkg="multiple_robots_simu" type="2robots_simulator_morse.sh" respawn="false" name="multiple_robots_simu_morse" output="screen"/>
</launch>

This launch is very simple because it is only used to launch another script called *2robots_simulator_morse.sh* that we can find in the script folder in the same package. This script is more complex. It contains a lot of links that have to be made to use certain simulation tools like PyMorse that is the API (Applications Programming Interface) for Morse simulation. Then, after the links we have to do the same kind of commands than in the general script in order to launch our two python codes and one extra command that I will explain later. So, we have the following script on Figure 4 :



```
#!/bin/bash

environment_name="multiple_robots_simu"
strands_morse=`rospack find strands_morse`
example=`rospack find multiple_robots_simu`
path="$example"
PYTHONPATH="$path/src:$common/src:$PYTHONPATH"
MORSE_RESOURCE_PATH="$strands_morse:$common/data:$common/robots:$path:$MORSE_RESOURCE_PATH"
export MORSE_RESOURCE_PATH PYTHONPATH
added=`$strands_morse/morse_config.py $environment_name $path`
echo "Running morse on $path with PYTHONPATH=$PYTHONPATH and MORSE_RESOURCE_PATH=$MORSE_RESOURCE_PATH"
PATH=/opt/strands-morse-simulator/bin:$PATH

xterm -hold -e "morse run multiple_robots_simu `rospack find multiple_robots_simu`/construction_map.py" &
sleep 3s
xterm -hold -e "roslaunch multiple_robots_simu navigation42robots.launch" &
sleep 3s

xterm -e "morse run multiple_robots_simu `rospack find multiple_robots_simu`/robot_human_control.py"

```

Figure 4 : Sub-level script with links, and 3 commands

2. Map simulation construction

First of all, we have to treat the basic python script that define the simulation. This is the first command of the Sub-level script because it's what define the environment of the simulation. It's possible to learn easily this part of the simulation with the Morse tutorials on their site (http://www.openrobots.org/morse/doc/1.3/tutorials.html).

In this part, we will make the difference between the two versions of the simulation I have developed because the content is not exactly the same as we don't use the same organization for the robots movements.

a) Version 2 move_base

At the first time, I wanted to use a particular navigation module from ROS called *move_base* for my two robots in the simulation. This module is a navigation node from ROS navigation stack that allows to move your robot using *Dijkstra's algorithm.* It means that it will take the shortest path to the destination using the sensors data of the robot.

In order to do that, I have to define two robots on the simulation. For this simulation, I choose two Scitos A5 provided in the STRANDS project package but I tried all possible kind of robots and even a human representation or vehicles provided by Morse like in Figure 5 (ATRV, PR2, Pioneer, Hummer, quadcopter …).

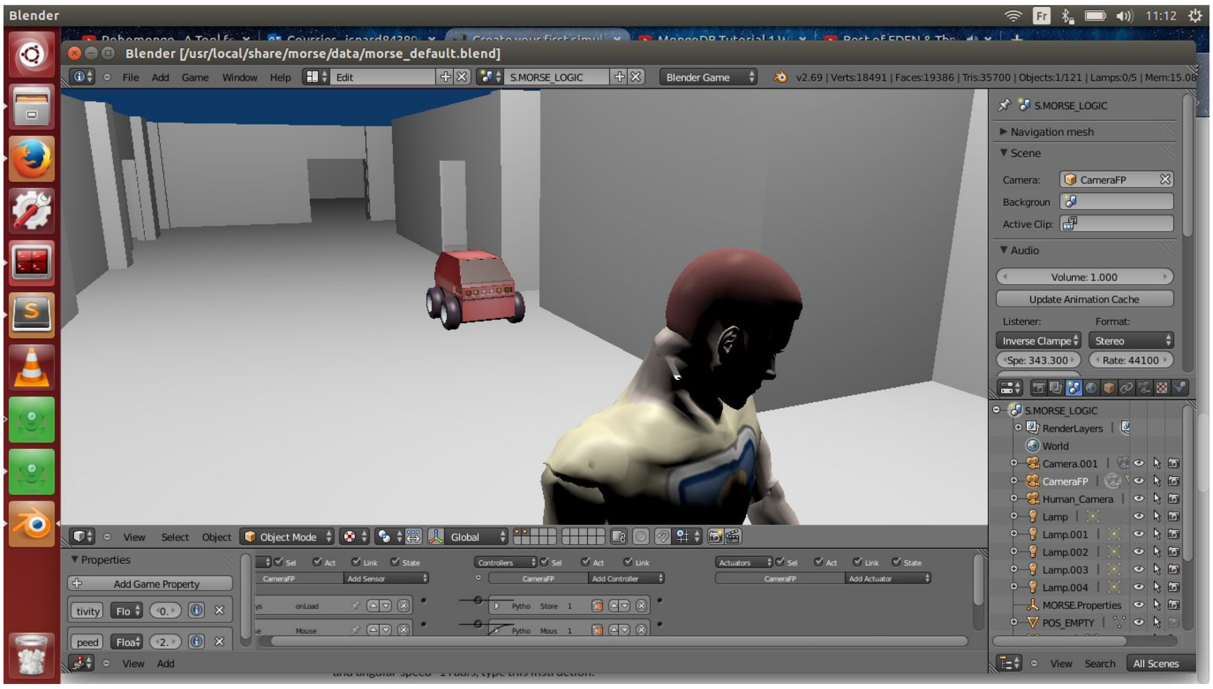
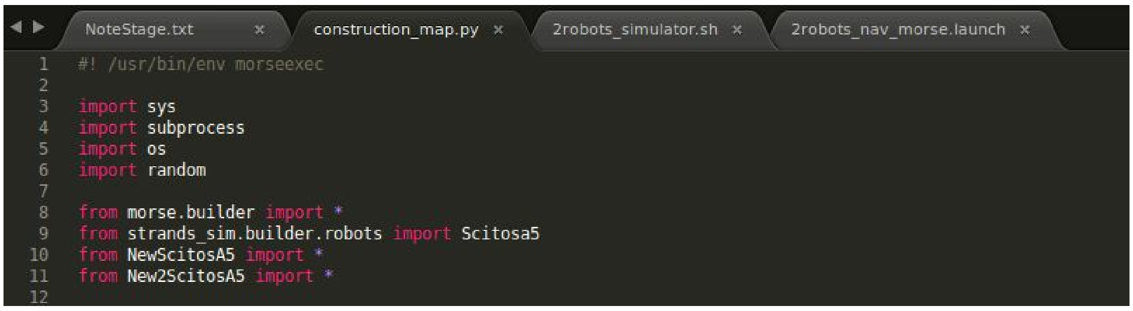


Figure 5 : Example of a Blender simulation with an ATRV and a Human

Nevertheless, we have chosen to use here two Robots Scitos A5. Each one of them has to be moving thanks to *move_base*. And this is the core of the difficulties with this version of the simulation but we will see that later. The navigation node *move_base* needs to be launched independently for each robot. The problem is that we need to define the topic in *namespaces* like *"/robot1/move_base*" and "*/robot2/move_base*". But first, let's explain the *construction_map.py* code starting with the links of the important codes and structures for the simulation in Figure 6 :



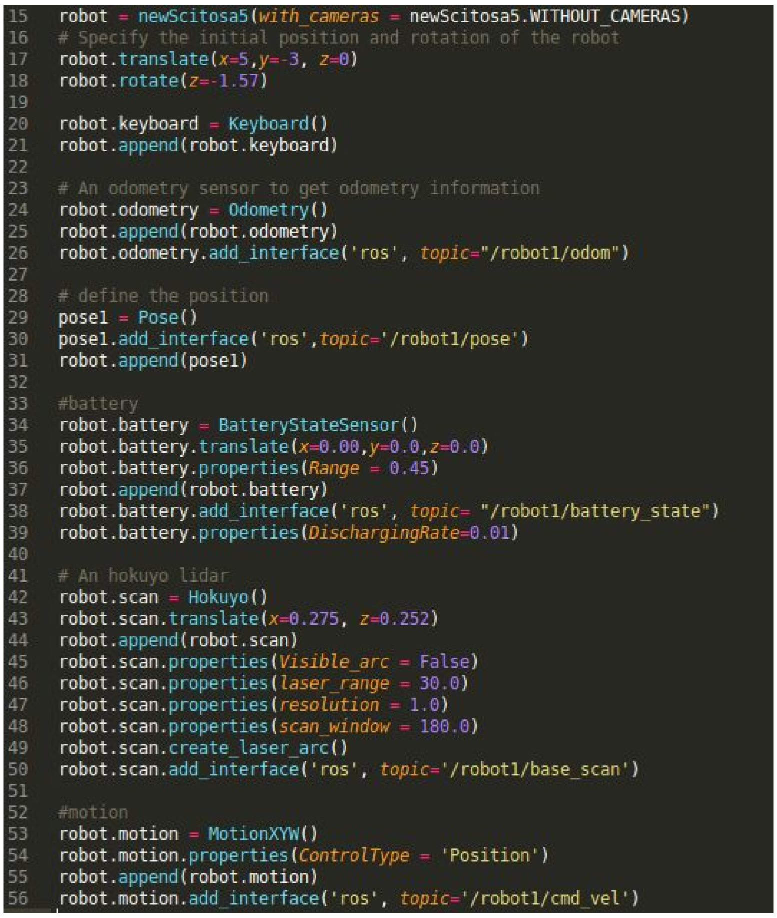
```
#! /usr/bin/env morseexec

import sys
import subprocess
import os
import random

from morse.builder import *
from strands_sim.builder.robots import Scitosa5
from NewScitosA5 import *
from New2ScitosA5 import *
```

Figure 6 : Links of the needed modules to be used in the simulation

We can find at the beginning of the code a command that looks like a comment but is very important because this is what calls the Morse executable in */usr/bin/env*. Without this first command, there is no Blender window opening for the simulation. Then, we have standard imports to use useful python libraries and some special imports that I have to explain here.

- "**From morse.builder import *** " allows to build the simulation here. We can't define a Morse simulation without it as it imports all the functions and tools required. This is the most important import of this python code.

- “**from strands_sim.builder.robots import Scitosa5**” allows to have access to the class that create a scitos A5 robot. *Scitosa5()* is the constructor for this robot class and in this constructor all the topics of sensors and actuators are defined.
- “**from NewScitosA5 import ***” and “**from New2ScitosA5 import ***” have the same usage than Scitosa5 but we needed to create other classes because of the problem of *namespace* for the 2 *move_base* utilisation. So, it’s just copies of the original version *Scitosa5()* but, unlike the original, they are almost empty so that we can define ourselves the topics we want to use (Figure 7).

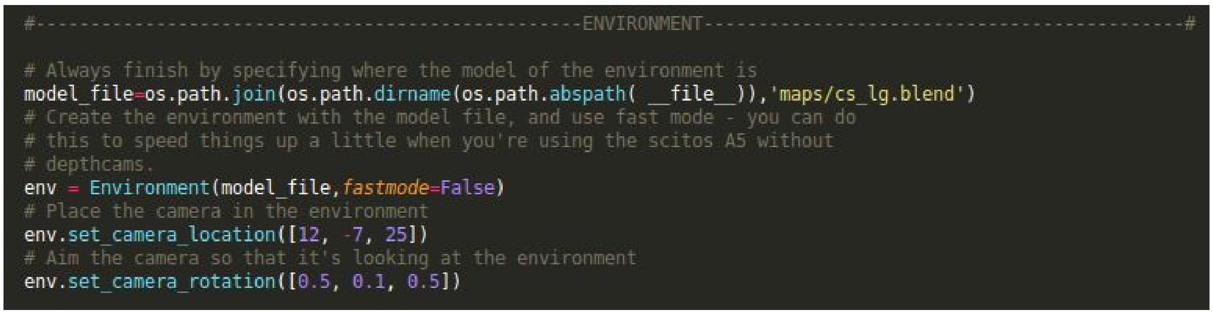


```python
robot = newScitosa5(with_cameras = newScitosa5.WITHOUT_CAMERAS)
# Specify the initial position and rotation of the robot
robot.translate(x=5,y=-3, z=0)
robot.rotate(z=-1.57)

robot.keyboard = Keyboard()
robot.append(robot.keyboard)

# An odometry sensor to get odometry information
robot.odometry = Odometry()
robot.append(robot.odometry)
robot.odometry.add_interface('ros', topic="/robot1/odom")

# define the position
pose1 = Pose()
pose1.add_interface('ros',topic='/robot1/pose')
robot.append(pose1)

#battery
robot.battery = BatteryStateSensor()
robot.battery.translate(x=0.00,y=0.0,z=0.0)
robot.battery.properties(Range = 0.45)
robot.append(robot.battery)
robot.battery.add_interface('ros', topic= "/robot1/battery_state")
robot.battery.properties(DischargingRate=0.01)

# An hokuyo lidar
robot.scan = Hokuyo()
robot.scan.translate(x=0.275, z=0.252)
robot.append(robot.scan)
robot.scan.properties(Visible_arc = False)
robot.scan.properties(laser_range = 30.0)
robot.scan.properties(resolution = 1.0)
robot.scan.properties(scan_window = 180.0)
robot.scan.create_laser_arc()
robot.scan.add_interface('ros', topic='/robot1/base_scan')

#motion
robot.motion = MotionXYW()
robot.motion.properties(ControlType = 'Position')
robot.append(robot.motion)
robot.motion.add_interface('ros', topic='/robot1/cmd_vel')
```

Figure 7 : Definition of the first scitos A5 robot

Thus, we can define our robot using the *Morse.builder* functions and tools. We see that we have to create a scitos, place it on the map as we want and then we can add it a controller by the keyboard, an odometry, a position, a battery, a lidar and finally a kind of motion used to make it move. The very important part here is the command “**robot.odometry.add_interface('ros', topic="/robot1/odom")**” because it’s what link the Morse simulation to ROS environment. We see that we just add the robot odometry interface for *‘ros*’ in the first argument and we precise the topic in the second argument to publish the state of the odometry. This is exactly the same procedure for the other robot except that we have to think about changing the topic like *“/robot2/odom".*

As the result, we have our robots completely defined here and we can now define the environment of simulation in Figure 8 :



```
#-------------------------------------------------ENVIRONMENT-------------------------------------------#

# Always finish by specifying where the model of the environment is
model_file=os.path.join(os.path.dirname(os.path.abspath( __file__)),'maps/cs_lg.blend')
# Create the environment with the model file, and use fast mode - you can do
# this to speed things up a little when you're using the scitos A5 without
# depthcams.
env = Environment(model_file,fastmode=False)
# Place the camera in the environment
env.set_camera_location([12, -7, 25])
# Aim the camera so that it's looking at the environment
env.set_camera_rotation([0.5, 0.1, 0.5])
```

Figure 8 : Definition of the environment of simulation

This final part define the environment *env* loading the map *maps/cs_lg.blend* . The only thing interesting to know is that we can set *fastmode* at True to have less details and lighten the simulation. You have to give an initial position for the camera but you can move it thereafter with the keyboard while the simulation is running. Then, this code is ready to use.

b) Version 1 move_base

Now, let’s talk about the second version of this code *construction_map_morse.py*. The two versions are very similar except that we have chosen here to use only one *move_base* node. This simplifies a lot the structure of the topics because there is no need to use *namespace* like “*robot1/move_base*” because we will use the default setup like just “*/movebase*”. Thus, we have one robot defined exactly the same way with just no *namespace* for its topics and we have another robot that will have to use the sockets to communicate with the Morse simulation. So, we will have to use this other kind of interface with **“robot.odometry.add_interface('socket')”** to use the sockets. Moreover, this second robot of the simulation will have to use another kind of motion than *move_base* and it’s Morse that provides the solution :

waypoint=Waypoint()
waypoint.add_interface('socket')
robot.append(waypoint)

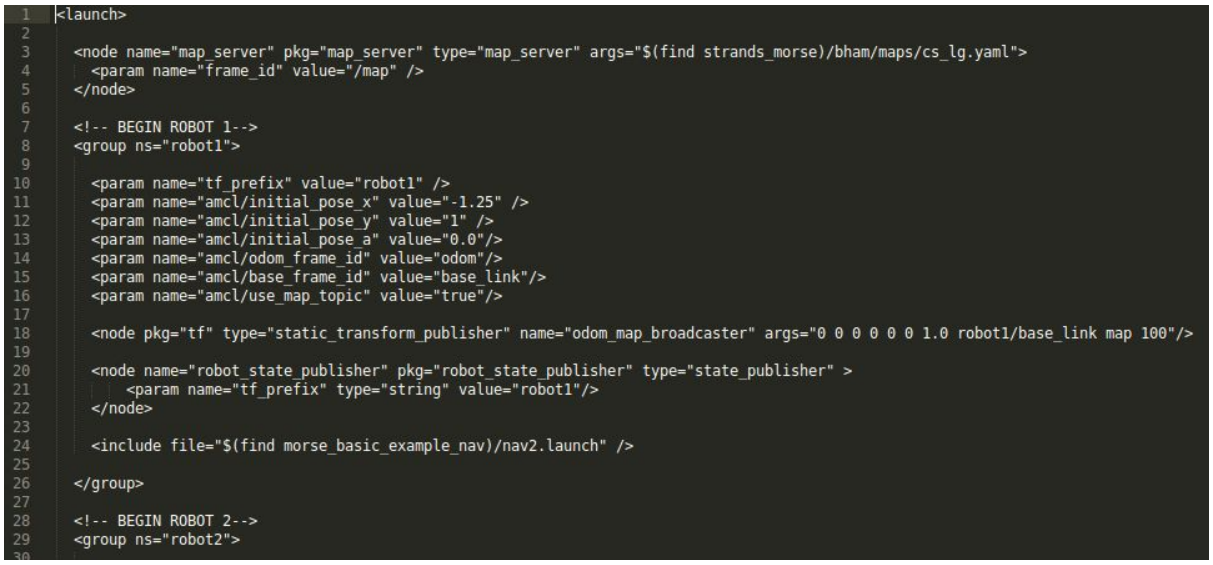
The waypoint is a motion control for robot in Morse simulation that goes by socket instead of ROS. This is quite useful and convenient because it just needs a target point to reach and the robot will move. This uses only Morse functions so it’s quite simple to use but the disadvantage is that there is no navigation optimisation. It means that this motion control will make the robot go only right straight to the target point without any avoidance or shorter path algorithms.

3. Navigation nodes

Now, we can talk about the additional command between the two python codes in the Sub-level script. This is a *roslaunch* for the *move_base* navigation. Thus, as we use 2 *move_base* nodes for the first version and only 1 *move_base* for the second, this launch will be different.

a) Version 2 move_base

The first version uses a *move_base* navigation node for each robot. Then, I had to build a specific structure that would launch the navigation node with all the needed navigation components in *namespaces* that respects the convention we have decided before for the topics (*/robot1/…* and */robot2/…*). I used a tutorial for multiple robots simulation on ROS forum at http://answers.ros.org/question/41433/multiple-robots-simulation-and-navigation/ . So, let's get back to the Sub-level script and see where the middle command leads us. We have "**roslaunch multiple_robots_simu navigation42robots.launch**" so we can see that I have made a launch in my package *multiple_robots_simu* to launch the navigation nodes for my 2 robots (Figure 9).



```
<launch>

  <node name="map_server" pkg="map_server" type="map_server" args="$(find strands_morse)/bham/maps/cs_lg.yaml">
    <param name="frame_id" value="/map" />
  </node>

  <!-- BEGIN ROBOT 1-->
  <group ns="robot1">

    <param name="tf_prefix" value="robot1" />
    <param name="amcl/initial_pose_x" value="-1.25" />
    <param name="amcl/initial_pose_y" value="1" />
    <param name="amcl/initial_pose_a" value="0.0"/>
    <param name="amcl/odom_frame_id" value="odom"/>
    <param name="amcl/base_frame_id" value="base_link"/>
    <param name="amcl/use_map_topic" value="true"/>

    <node pkg="tf" type="static_transform_publisher" name="odom_map_broadcaster" args="0 0 0 0 0 0 1.0 robot1/base_link map 100"/>

    <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" >
        <param name="tf_prefix" type="string" value="robot1"/>
    </node>

    <include file="$(find morse_basic_example_nav)/nav2.launch" />

  </group>

  <!-- BEGIN ROBOT 2-->
  <group ns="robot2">
```

Figure 9 : beginning of navigation42robots.launch

This launch has a similar structure for the 2 robots so we just have to see the first part like in Figure 9 and we repeat the same thing for second robot changing only the *namespace* and eventually the initial position. The first thing to do here is to load the map. The is the only node out of the *namespaces* because the map is general, it's common for the two robots. As we can see, we use the map_server to load the same map *cs_lg.yaml* . The purpose here is to have this structure:

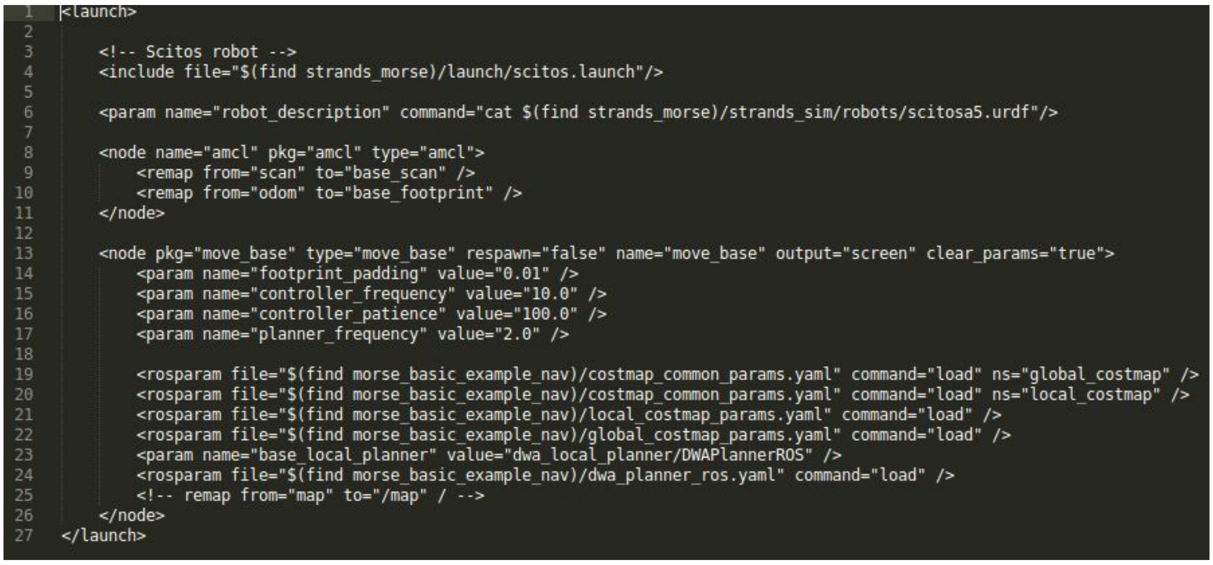
/map

/robot1/robot_state_publisher */robot2/robot_state_publisher*
/robot1/robot_pose_ekf */robot2/robot_pose_ekf*
/robot1/amcl */robot2/amcl*
/robot1/move_base */robot2/move_base*

Thus, we have to define the *namespaces* and launch the different needed nodes in them. The definition of *namespace* is made with **<group ns="robot1">** ... **</group> .** The content will automatically have the prefix "*robot1*".

Then, we have to give this prefix to *tf* node that is the node responsible to make the different landmarks changes : **<param name="tf_prefix" value="robot1" />** and, thereafter in the launching of *robot_state_publisher,* we have the same in parameter. This *robot_state_publisher* is supposed to launch *tf* node but the problems start here because, in my case, I have to launch it myself with **<node pkg="tf" type="static_transform_publisher" name="odom_map_broadcaster" args="0 0 0 0 0 0 1.0 robot1/base_link map 100"/>** . Otherwise, nothing works.

We can also see that we define all initial values for *amcl* node. AMCL (Adaptive Monte-Carlo based module for Localisation) is a probabilistic localization system. We give it the initial values for our robots so that he is not lost at the beginning of the simulation. So, we have the *initial_pose_x* , *initial_pose_x* , *initial_pose_a* for the initial x,y positions and angles. But we also have some setup like *odom_frame_id, base_frame_id* and *use_map_topic* that we have to define. Then, we can go to the final step with the last launch called with **<include file="$(find morse_basic_example_nav)/nav2.launch" />** that we detail in Figure 10 :



```
<launch>

    <!-- Scitos robot -->
    <include file="$(find strands_morse)/launch/scitos.launch"/>

    <param name="robot_description" command="cat $(find strands_morse)/strands_sim/robots/scitosa5.urdf"/>

    <node name="amcl" pkg="amcl" type="amcl">
        <remap from="scan" to="base_scan" />
        <remap from="odom" to="base_footprint" />
    </node>

    <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen" clear_params="true">
        <param name="footprint_padding" value="0.01" />
        <param name="controller_frequency" value="10.0" />
        <param name="controller_patience" value="100.0" />
        <param name="planner_frequency" value="2.0" />

        <rosparam file="$(find morse_basic_example_nav)/costmap_common_params.yaml" command="load" ns="global_costmap" />
        <rosparam file="$(find morse_basic_example_nav)/costmap_common_params.yaml" command="load" ns="local_costmap" />
        <rosparam file="$(find morse_basic_example_nav)/local_costmap_params.yaml" command="load" />
        <rosparam file="$(find morse_basic_example_nav)/global_costmap_params.yaml" command="load" />
        <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
        <rosparam file="$(find morse_basic_example_nav)/dwa_planner_ros.yaml" command="load" />
        <!-- remap from="map" to="/map" / -->
    </node>
</launch>
```

Figure 9 : nav2.launch

The final step is to launch the construction of the robot for each of our scitos with the *scitos.launch* then to load the *urdf* files that contains the whole robot descriptions. To finish, we launch *amcl* node doing remapping and *move_base* with all the parameters of the cost maps.

The complexity of this version leads to errors and malfunctions certainly because of the *tf* problem. I didn't find the origin of the problem and choose to go on to the next version but it would be nice to fix this version with more time. It would allow to use two robots using *move_base* in the same Morse simulation.

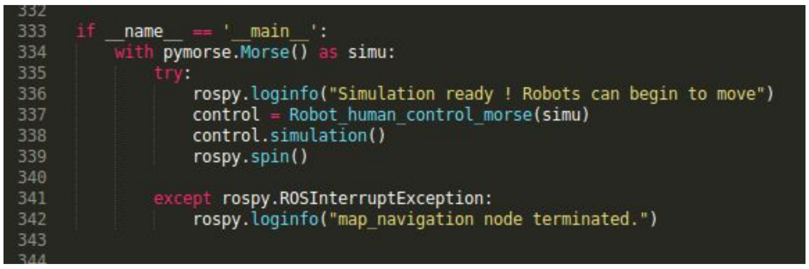
b) Version 1 move_base

For the 1 *move_base* version, it's much easier because we only need one launch to give every parameters and launch the needed nodes. We have exactly the same nodes running but there is no *namespace* so no problem for *tf*. Just don't forget to replace the *roslaunch* in the Sub-level script like **"roslaunch morse_basic_example_nav nav.launch"**. This version with one simple *move_base* works perfectly.

4. Movement simulation

Finally, we can move on to the last python code that defines the simulation. This python code is responsible to handle the movement of the robots and the generation of the Markov chain. We don't separate the two versions of the code here because there are pretty similar. I will only talk about the second version *robot_human_control_morse.py* that works for sure with 1 only *move_base* and 1 *Waypoint*. The only difference will be that, in the the first version, there will be 2 robots using the *move_base* goal sending function.

So, let's start by the main of this python code in Figure 10:



```python
if __name__ == '__main__':
    with pymorse.Morse() as simu:
        try:
            rospy.loginfo("Simulation ready ! Robots can begin to move")
            control = Robot_human_control_morse(simu)
            control.simulation()
            rospy.spin()

        except rospy.ROSInterruptException:
            rospy.loginfo("map_navigation node terminated.")
```

Figure 10 : main of robot_human_control_morse.py

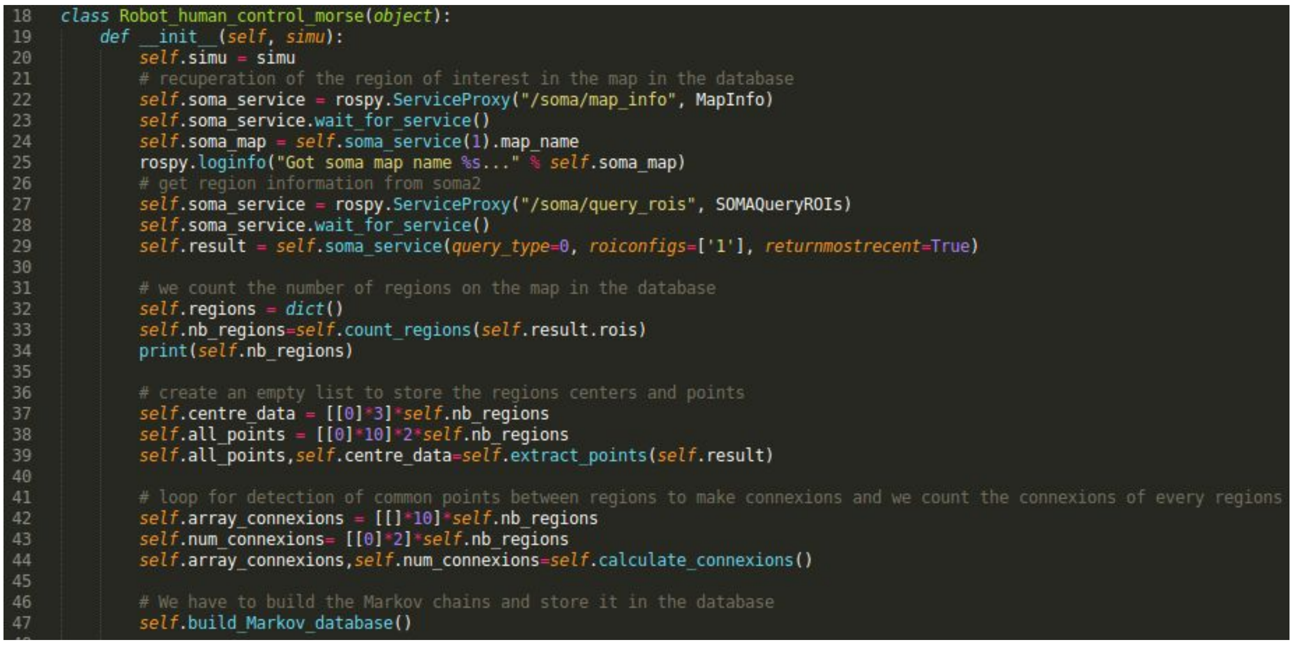
First of all, we have to link at the beginning the important libraries that we will use in this code :

from geometry_msgs.msg import *
from soma_map_manager.srv import MapInfo
from soma_manager.srv import SOMAQueryROIs
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from math import radians, degrees
from actionlib_msgs.msg import *

In this python main, we use the Morse API called PyMorse to access to the Morse simulation that we have initialized in *construction_map_morse.py* . This is very simple, we start by declaring a *Robot_human_control_morse* entity and use a class function called simulation().

a) Initialization

Let's see what *Robot_human_control_morse* class is composed with to understand how it works. The principal of the operation is in the *_init_* of the class, it's the constructor of the class and we are going to see what it builds at this moment in Figure 11:



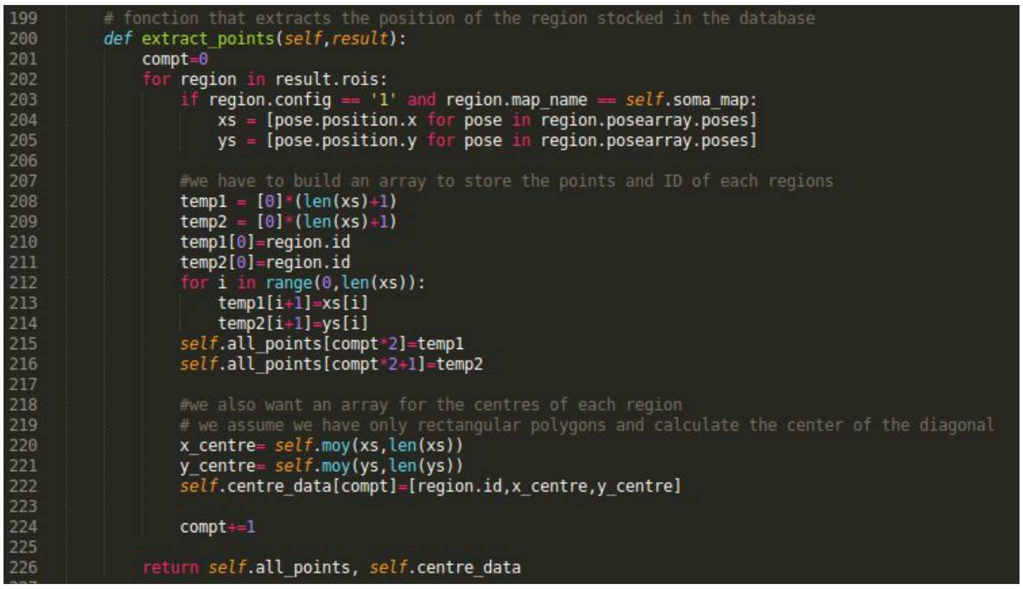
```python
class Robot_human_control_morse(object):
    def __init__(self, simu):
        self.simu = simu
        # recuperation of the region of interest in the map in the database
        self.soma_service = rospy.ServiceProxy("/soma/map_info", MapInfo)
        self.soma_service.wait_for_service()
        self.soma_map = self.soma_service(1).map_name
        rospy.loginfo("Got soma map name %s..." % self.soma_map)
        # get region information from soma2
        self.soma_service = rospy.ServiceProxy("/soma/query_rois", SOMAQueryROIs)
        self.soma_service.wait_for_service()
        self.result = self.soma_service(query_type=0, roiconfigs=['1'], returnmostrecent=True)

        # we count the number of regions on the map in the database
        self.regions = dict()
        self.nb_regions=self.count_regions(self.result.rois)
        print(self.nb_regions)

        # create an empty list to store the regions centers and points
        self.centre_data = [[0]*3]*self.nb_regions
        self.all_points = [[0]*10]*2*self.nb_regions
        self.all_points,self.centre_data=self.extract_points(self.result)

        # loop for detection of common points between regions to make connexions and we count the connexions of every regions
        self.array_connexions = [[]*10]*self.nb_regions
        self.num_connexions= [[0]*2]*self.nb_regions
        self.array_connexions,self.num_connexions=self.calculate_connexions()

        # We have to build the Markov chains and store it in the database
        self.build_Markov_database()
```

Figure 11 : _init_ of the Robot_human_control_morse class

The first thing to do for our simulation movement is to extract the data regions from the database that we have defined previously in Rviz. We need to have access to every points of each polygons of the rois. We use **rospy.ServiceProxy()** in order to make the link between our code and the database. Once it's done, we do a query in the database to have our rois with **self.result = self.soma_service(query_type=0, roiconfigs=['1'], returnmostrecent =True) .** It means that we search all documents (*query_type=0*) with the datafield *roiconfigs='1'.* All these documents are returned in *self.result* . Then we have to extract the data we are interested in and build those which we need thereafter.

We start by counting the number of rois defined on the map with a simple function with a *for* to increment the number as many time as there is rois in *self.result.rois*.

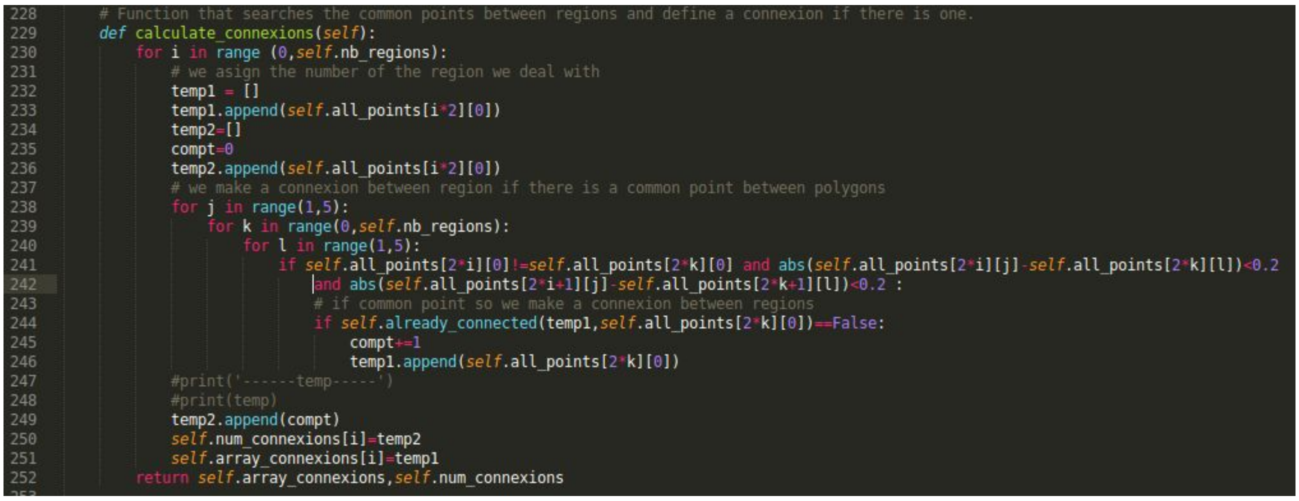
Then, we have to build our arrays containing all the points of all the rois with their IDs (*all_points*) and the position of the all rois centers with their IDs (*centre_data*). We need only one function which is *extract_points()* shown in the Figure 12 :



```
    # fonction that extracts the position of the region stocked in the database
    def extract_points(self,result):
        compt=0
        for region in result.rois:
            if region.config == '1' and region.map_name == self.soma_map:
                xs = [pose.position.x for pose in region.posearray.poses]
                ys = [pose.position.y for pose in region.posearray.poses]

                #we have to build an array to store the points and ID of each regions
                temp1 = [0]*(len(xs)+1)
                temp2 = [0]*(len(xs)+1)
                temp1[0]=region.id
                temp2[0]=region.id
                for i in range(0,len(xs)):
                    temp1[i+1]=xs[i]
                    temp2[i+1]=ys[i]
                self.all_points[compt*2]=temp1
                self.all_points[compt*2+1]=temp2

                #we also want an array for the centres of each region
                # we assume we have only rectangular polygons and calculate the center of the diagonal
                x_centre= self.moy(xs,len(xs))
                y_centre= self.moy(ys,len(ys))
                self.centre_data[compt]=[region.id,x_centre,y_centre]

                compt+=1

        return self.all_points, self.centre_data
```

Figure 12 : function extract_points()

Afterwards, we need to build the array of the connexions between rois. So, I have made a function to detect and count the connexions. This function called *calculate_connexions()* returns *array_connexions* that contains the list of roi IDs with the list of roi IDs connected of each one of them and *num_connexions* that contains the list of roi IDs with the number of connexions of each roi (Figure 13).

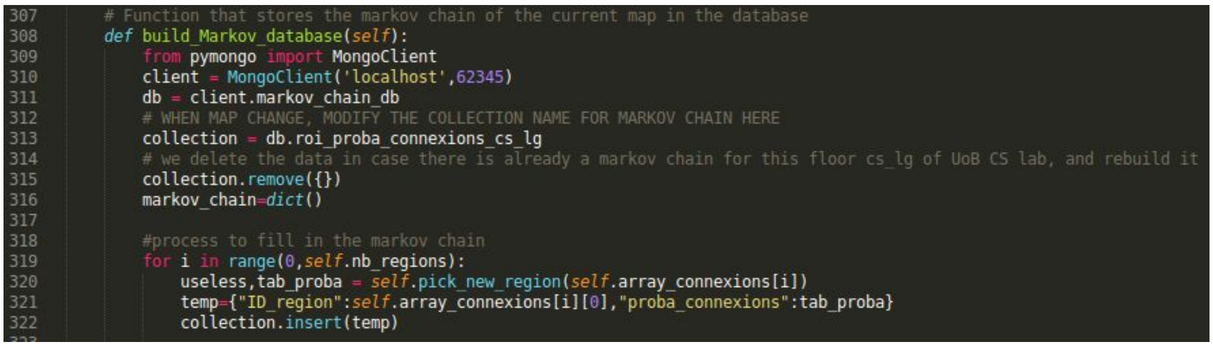


```
    # Function that searches the common points between regions and define a connexion if there is one.
    def calculate_connexions(self):
        for i in range (0,self.nb_regions):
            # we asign the number of the region we deal with
            temp1 = []
            temp1.append(self.all_points[i*2][0])
            temp2=[]
            compt=0
            temp2.append(self.all_points[i*2][0])
            # we make a connexion between region if there is a common point between polygons
            for j in range(1,5):
                for k in range(0,self.nb_regions):
                    for l in range(1,5):
                        if self.all_points[2*i][0]!=self.all_points[2*k][0] and abs(self.all_points[2*i][j]-self.all_points[2*k][l])<0.2
                            and abs(self.all_points[2*i+1][j]-self.all_points[2*k+1][l])<0.2 :
                            # if common point so we make a connexion between regions
                            if self.already_connected(temp1,self.all_points[2*k][0])==False:
                                compt+=1
                                temp1.append(self.all_points[2*k][0])
            #print('------temp-----')
            #print(temp)
            temp2.append(compt)
            self.num_connexions[i]=temp2
            self.array_connexions[i]=temp1
        return self.array_connexions,self.num_connexions
```

Figure 13 : function calculate_connexions()

We review every regions of the map and we check each point of each roi to determine if there is a common point between the regions. We assume here that we have a common point if there is less that 0.2 meters between the values on x and y. Thus, when we define the rois in Rviz, we have to be precise at 20 cm but this is easy. When the function encounters a common point between two regions, we have to check if there is not already a connexion defined (in case there is multiple common points). So, I have made another function *already_connected()* that will be responsible to check that eventuality. If there is no connexion yet between the 2 roi, we increment the counter and add the ID of the roi connected to the list.

Once we have done that, the only remaining task to do is to use the function that will build the Markov chain and store it in the database. The best way to learn how to store something in the database is to do the PyMongo tutorial at http://api.mongodb.com/python/current/tutorial.html . Let's see the *build_Markov_database()* in the Figure 14 :



```
# Function that stores the markov chain of the current map in the database
def build_Markov_database(self):
    from pymongo import MongoClient
    client = MongoClient('localhost',62345)
    db = client.markov_chain_db
    # WHEN MAP CHANGE, MODIFY THE COLLECTION NAME FOR MARKOV CHAIN HERE
    collection = db.roi_proba_connexions_cs_lg
    # we delete the data in case there is already a markov chain for this floor cs_lg of UoB CS lab, and rebuild it
    collection.remove({})
    markov_chain=dict()

    #process to fill in the markov chain
    for i in range(0,self.nb_regions):
        useless,tab_proba = self.pick_new_region(self.array_connexions[i])
        temp={"ID_region":self.array_connexions[i][0],"proba_connexions":tab_proba}
        collection.insert(temp)
```

Figure 14 : build_Markov_database()

We have to import PyMongo here to use the *MongoClient()*. This is the moment to choose the name of our database and collection (here '*markov_chain_db*' and '*roi_proba_connexions_cs_lg*') so if you have to change it thereafter, it's in this function. We remove all we have in the collection if there is already something and start reviewing all the rois again to determine the appropriate probabilities of each connexions and store it in the database with **collection.insert(temp)**.

The last function of this section is *pick_new_region()* that has a double usage. It can be run to have the array of probabilities for one region in *tab_proba* like in this case. The first result is useless and we store the second one *tab_proba*. Or, in the other hand, we can use it dynamically and it will give us only the ID picked to be the next target according to the Markov chain in *next_id*. This is a quite long function and not very transparent so I will just give it in the annexes and explain how it works here.

This function takes just the line with the considered roi ID with the list of the connected rois, we call that line *line_connexions.* As I explained in the Region of interest definition part (A. 2/) before, we have made the choice to define the probability of a connexion according to the number of connexions the connected roi has. We assume that a person moving in a building is less likely to want to stay in very connected areas (like hallways) than few connected areas (like offices). So, we define here a coefficient proportional to the number of connexions for each connected area of *line_connexions.* We bring it back to 100 so that it gives us a probability in percentage to go by each connexions for the considered roi (it's also possible to stay in the same roi for a while). And finally, we pick randomly a number between 0 and 100 with **pick=random.uniform(0,100)** to make a choice of next target and give it as the result in *next_id*.

b) Simulation

Now that our *Robot_human_control_morse* class is fully initialized, the *main* will go on to the *control.simulation()* function and it's the last part of the simulation. This is where we will give the motion to our robots. This function is simple too because it's just a loop while the node is not shut down where we pick a region to target with *pick_new_region()* function as explained before.

Then, we send the target position with *MoveToGoal()* function. The rest of the time it does nothing but wait to reach the goal and restart the same process. The *MoveToGoal()* function (furnished in annexes) is more interesting because it uses a simple action client for move_base like "**ac = actionlib.SimpleActionClient("move_base", MoveBaseAction)**" . It allows to use the class functions like "**ac.wait_for_server(rospy.Duration.from_sec(5.0))**", "**ac.send_goal(goal)**", "**ac.wait_for_result(rospy.Duration(30))**" and "**ac.get_state() == GoalStatus.SUCCEEDED**" that are completely transparent to use.

Finally, we obtain 1 robot using *move_base* to move in the simulation and 1 robot that we can manipulate as we want with the *self.simu.robot.waypoint.publish()* function. The only note for this function is that we have to make a landmark change between Morse landmark (used by this function) and Rviz landmark. The changes are :

X_blender = Y_rviz + 3.8
Y_blender = -X_rviz - 3.8
Z_blender = Z_rviz (unused)

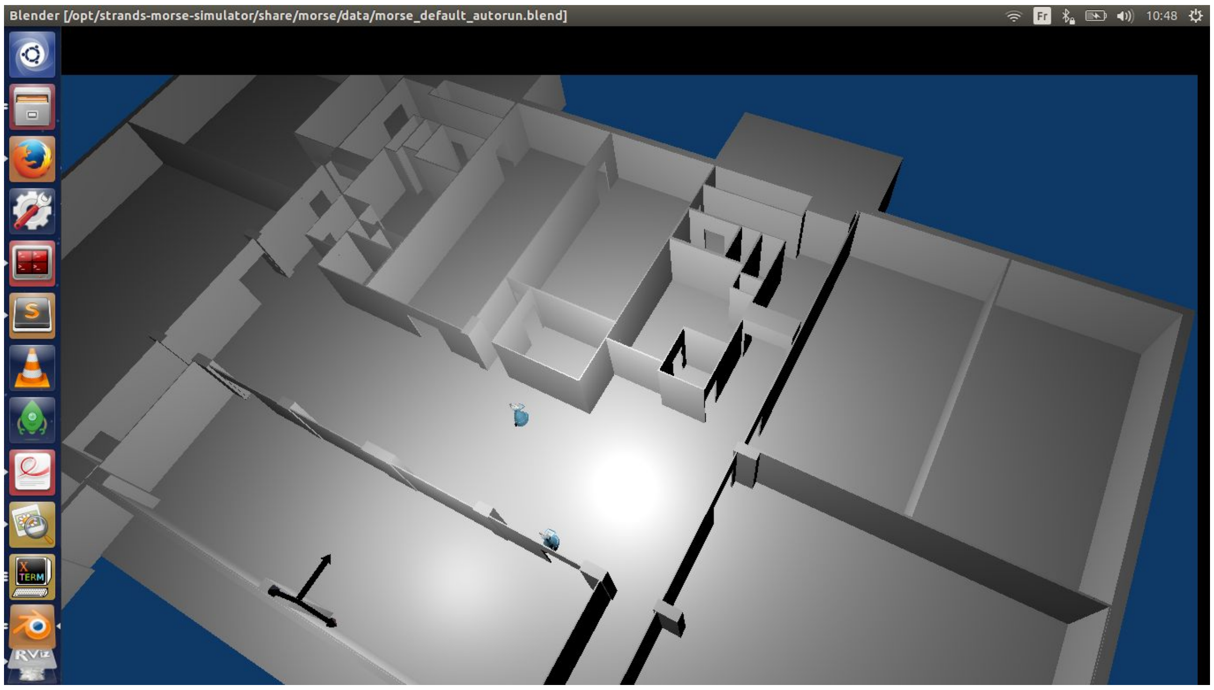
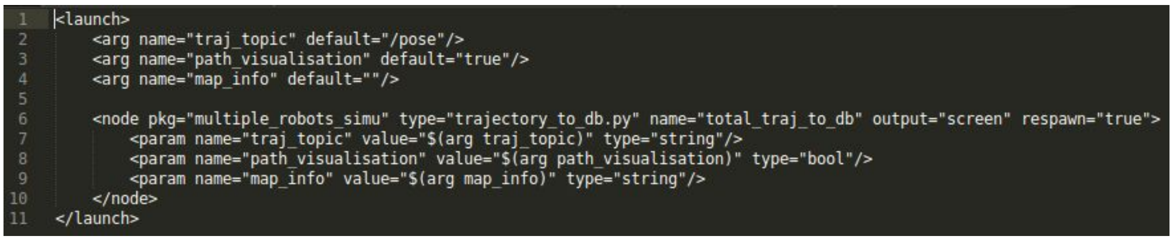

Blender [/opt/strands-morse-simulator/share/morse/data/morse_default_autorun.blend]
Fr
10:48
RViz

Figure 15 : Two robots simulation in Blender at the ground floor of the University of Birmingham

C. Trajectory storage and visualisation

For the last part of this internship, I had to be able to catch the trajectory of the robot, store it in the database and visualize it in Rviz. In order to do that, I had to build a new *roslaunch* that we have to add to General script and it leads us to the *launch* files Figure 16.

xterm -hold -e "roslaunch multiple_robots_simu trajectory_to_db.launch" &
sleep 1s



```
<launch>
    <arg name="traj_topic" default="/pose"/>
    <arg name="path_visualisation" default="true"/>
    <arg name="map_info" default=""/>

    <node pkg="multiple_robots_simu" type="trajectory_to_db.py" name="total_traj_to_db" output="screen" respawn="true">
        <param name="traj_topic" value="$(arg traj_topic)" type="string"/>
        <param name="path_visualisation" value="$(arg path_visualisation)" type="bool"/>
        <param name="map_info" value="$(arg map_info)" type="string"/>
    </node>
</launch>
```

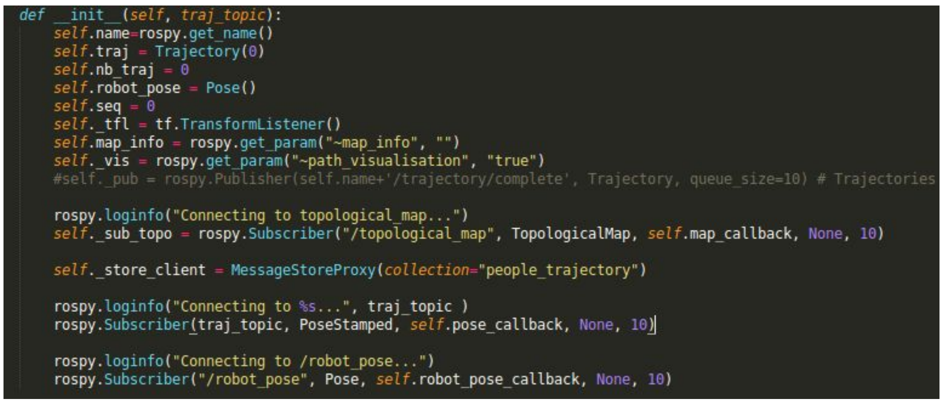
Figure 16 : trajectory_to_db.launch

This *roslaunch* allows to give some arguments to the node launching the python code *trajectory_to_db.py*. We can give the topic where the trajectory of the robot will be published and choose if we want to visualize the path or not. We can also add info about the map.

Let's see here the most important parts of the code and I join the whole code in the annexes. This code has been inspired by a the class *TrajectoryManager* in a package named *human_trajectory* made by Ferdian Jovan that was supervising our work. The original code was responsible to catch the data on a topic from a people tracker, store them in the database and give them available to print in Rviz. It's quite the same thing that I had to do here but I had to catch a different type of data on a different topic. So, it changes also the way to store the data. The advantage is that the type of data that I catch in this code is simple, it's a *PoseStamped* and for the storing I can use the same class *Trajectory* than in Ferdian's code.

Therefore, at the beginning of my code in the imports, we can find again "**from mongodb_store.message_store import MessageStoreProxy**" to handle the database. We have also "**from geometry_msgs.msg import PoseStamped**" that is the ROS message we catch on the topic */pose* that is our robot position.

Then, we have a *main* pretty simple just like previously, it's composed by only a class *InlinePoseTrajectory* constructor and, after that, a class function calling for *publish_trajectory().* First, let's see what the class contain. To know that, we go in *_init_* function like (Figure 17).



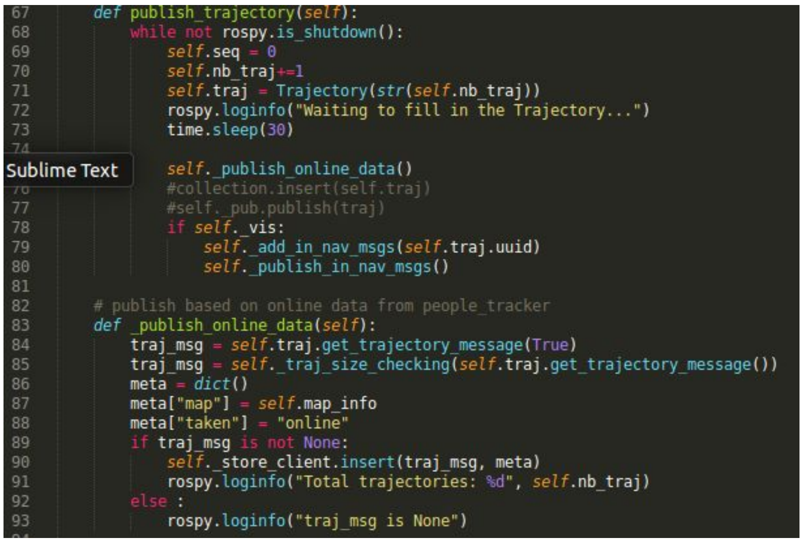
```
def __init__(self, traj_topic):
    self.name=rospy.get_name()
    self.traj = Trajectory(0)
    self.nb_traj = 0
    self.robot_pose = Pose()
    self.seq = 0
    self._tfl = tf.TransformListener()
    self.map_info = rospy.get_param("~map_info", "")
    self._vis = rospy.get_param("~path_visualisation", "true")
    #self._pub = rospy.Publisher(self.name+'/trajectory/complete', Trajectory, queue_size=10) # Trajectories

    rospy.loginfo("Connecting to topological_map...")
    self._sub_topo = rospy.Subscriber("/topological_map", TopologicalMap, self.map_callback, None, 10)

    self._store_client = MessageStoreProxy(collection="people_trajectory")

    rospy.loginfo("Connecting to %s...", traj_topic )
    rospy.Subscriber(traj_topic, PoseStamped, self.pose_callback, None, 10)

    rospy.loginfo("Connecting to /robot_pose...")
    rospy.Subscriber("/robot_pose", Pose, self.robot_pose_callback, None, 10)
```

Figure 17 : _init_ function in trajectory_to_db.launch

We subscribe to several topics but the most important is *traj_topic* which is the one that we want to store and visualize in Rviz. We find a command “**self._store_client = MessageStoreProxy(collection="people_trajectory")**” that is the equivalent for *Message* of **rospy.ServiceProxy()** seen before. We create an empty *Trajectory* entity and we use the command “**rospy.Subscriber(traj_topic, PoseStamped, self.pose_callback, None, 10)**” to catch the publication on the *traj_topic*. Each time there is something published on the topic, the *self.pose_callback* function will be called. In this function, we will use a class function *append_pose()* of the *Trajectory* entity created before to add our *PoseStamped* data in the *Trajectory* entity (after a landmark change).

Now, we can move on to *publish_trajectory()* function called in the *main*. This function is a loop that doesn’t stop while the node is running. Each 30 seconds we create a new *Trajectory* and we publish the data collected during this time. There is two ways to publish the data in this case :

The first one calls *_publish_online_data()* that convert the *Trajectory* in ROS trajectory message, check the size of the message and store it in the database with **insert()**. We have this function in Figure 18 after *publish_trajectory()* function.



```
    def publish_trajectory(self):
        while not rospy.is_shutdown():
            self.seq = 0
            self.nb_traj+=1
            self.traj = Trajectory(str(self.nb_traj))
            rospy.loginfo("Waiting to fill in the Trajectory...")
            time.sleep(30)

            self._publish_online_data()
            #collection.insert(self.traj)
            #self._pub.publish(traj)
            if self._vis:
                self._add_in_nav_msgs(self.traj.uuid)
                self._publish_in_nav_msgs()

    # publish based on online data from people_tracker
    def _publish_online_data(self):
        traj_msg = self.traj.get_trajectory_message(True)
        traj_msg = self._traj_size_checking(self.traj.get_trajectory_message())
        meta = dict()
        meta["map"] = self.map_info
        meta["taken"] = "online"
        if traj_msg is not None:
            self._store_client.insert(traj_msg, meta)
            rospy.loginfo("Total trajectories: %d", self.nb_traj)
        else :
            rospy.loginfo("traj_msg is None")
```

Figure 18 : publish_trajectory() and publish_online_data() functions in trajectory_to_db.launch

The second one is active if the *path_visualization* argument is *True*. It calls the function *_add_in_nav_msg()* that will define a *Publisher* for our trajectory in a *Path* format that can be displayed in Rviz. Then, this *nav_msg* will be published with *_publish_in_nav_msgs()* function.

The last thing to do to display the result in Rviz is to run “**rosrun human_trajectory traj_visualisation.py all**” and add the */traj_visualisation/update* topic. We obtain all the trajectories like in Figure 19.

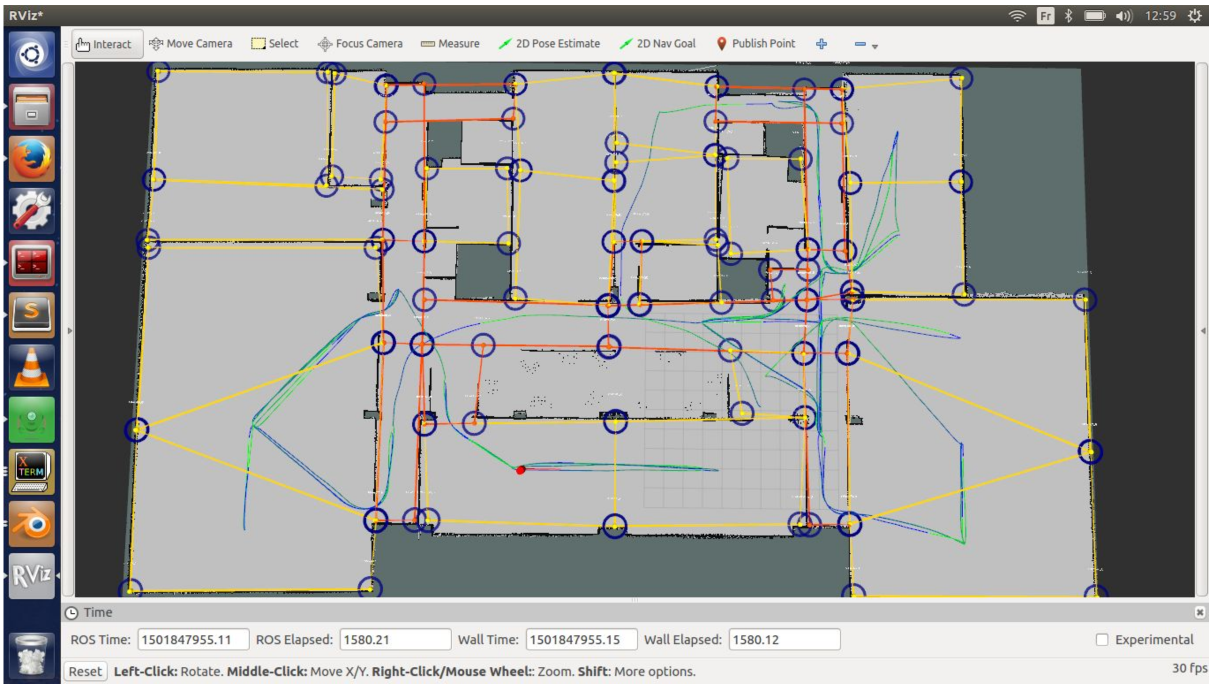


Figure 19 : Visualisation Rviz of the robot Human entity trajectory after one hour of simulation

V. Conclusion

As a conclusion of this internship report, I will summarize my work and explain what it remains to be done. The main goal of this internship was to build a two robots simulation with one moving like a human and another one trying to predict where the first one want to go. The first part has been done successfully but the fact is that the *people_tracker* code didn't work in the simulation so we didn't have the opportunity to do the second part here.

Nevertheless, we have a stable basis for the simulation and there is only few things missing to complete the challenge of this complete simulation. Fixing the problem of the 2 *move_base* version of the simulation would be a good step forward. But the second version works fine so we could just use this one and adapt the *people_tracker* to the simulation. That will allow us to have one robot catching the position of the other one. Then, we would compare the proportion of the total trajectory (which one we have now available the data) that the *people_tracker* has to get in order to predict where the first robot will go.

So, during this internship, I have handled a lot of new applications for me and I have learnt the best part of what I wanted on ROS environment. I have built a simulation that can be easily used and modified. The fact that I can generate dynamically the Markov chain and deduce all the needed information from the rois defined in Rviz is a real advantage for the adaptability and the reusability of my code. I have learnt to solve my problems with autonomy which is essential for an engineer. This internship has been very educating and useful for me and I hope my work will be too when it will be reuse.

VI. Bibliography

Ubuntu installation:

- https://doc.ubuntu-fr.org/cohabitation_ubuntu_windows

ROS installation:

- http://wiki.ros.org/indigo/Installation/Ubuntu

Terminator:

- https://doc.ubuntu-fr.org/terminator

Strands tutorials:

- http://strands.readthedocs.io/en/latest/setup.html

MORSE tutorials:

- https://www.openrobots.org

MORSE through PyMorse tutorials:

- https://www.openrobots.org/morse/doc/stable/pymorse.html

MongoDB with Robo 3T (RoboMongo):

- https://robomongo.org/

MongoDB through PyMongo tutorials:

- http://api.mongodb.com/python/current/tutorial.html

Multiple robots simulation:

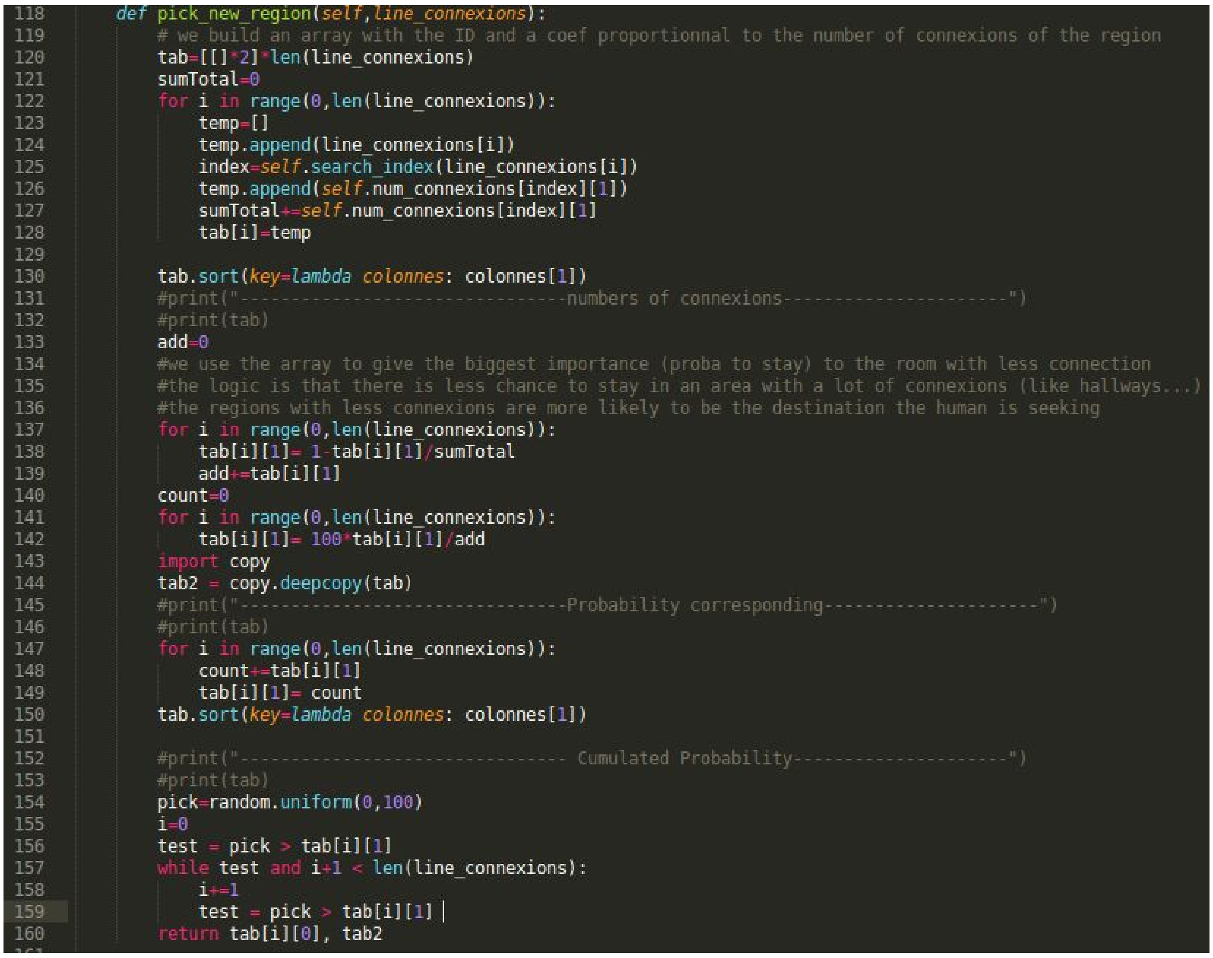
- http://answers.ros.org/question/41433/multiple-robots-simulation-and-navigation/

AMCL (Adaptive Monte-Carlo based module for Localisation):

- http://wiki.ros.org/amcl

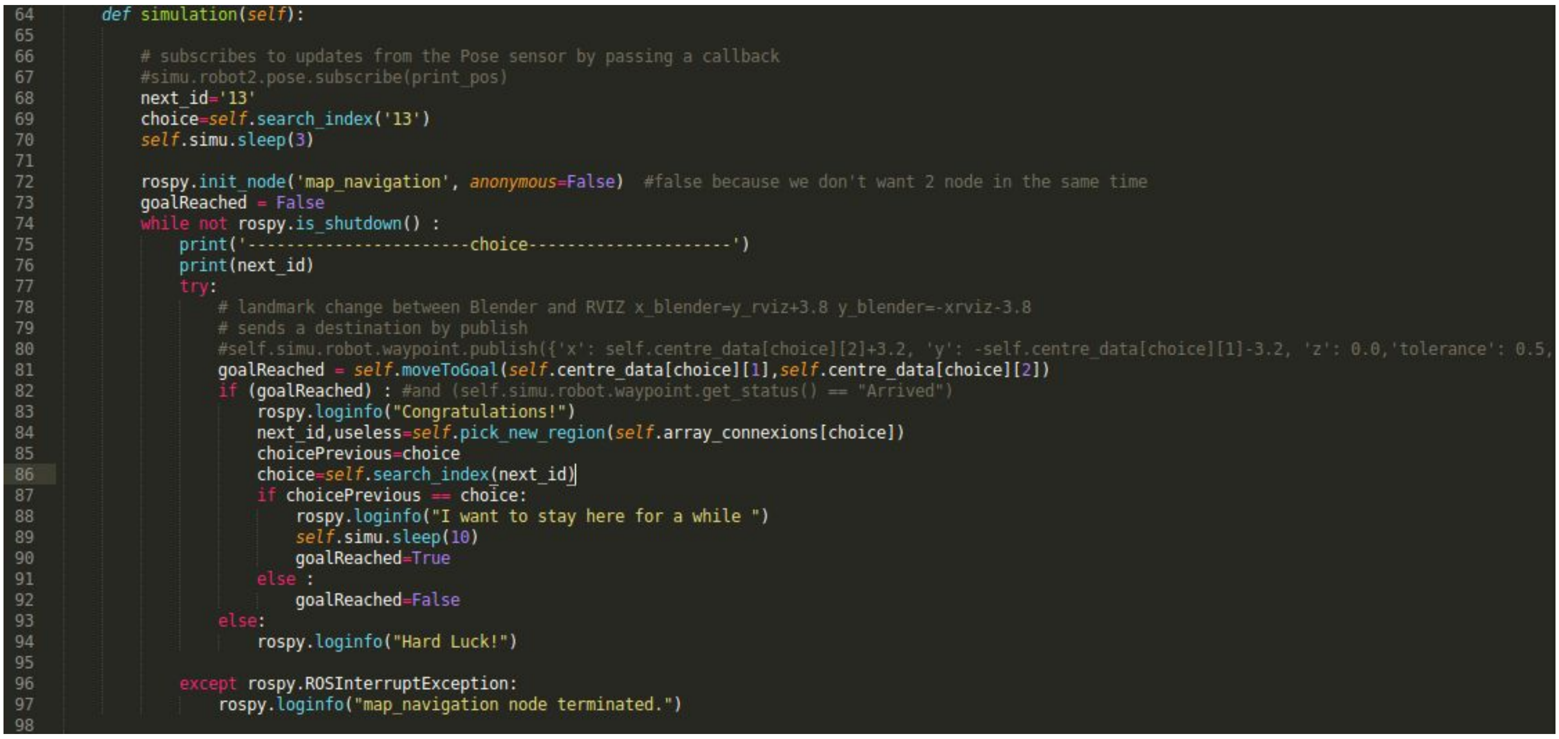
VII. Annexes

- **Function *pick_new_region()* in *robot_human_control_morse.py***



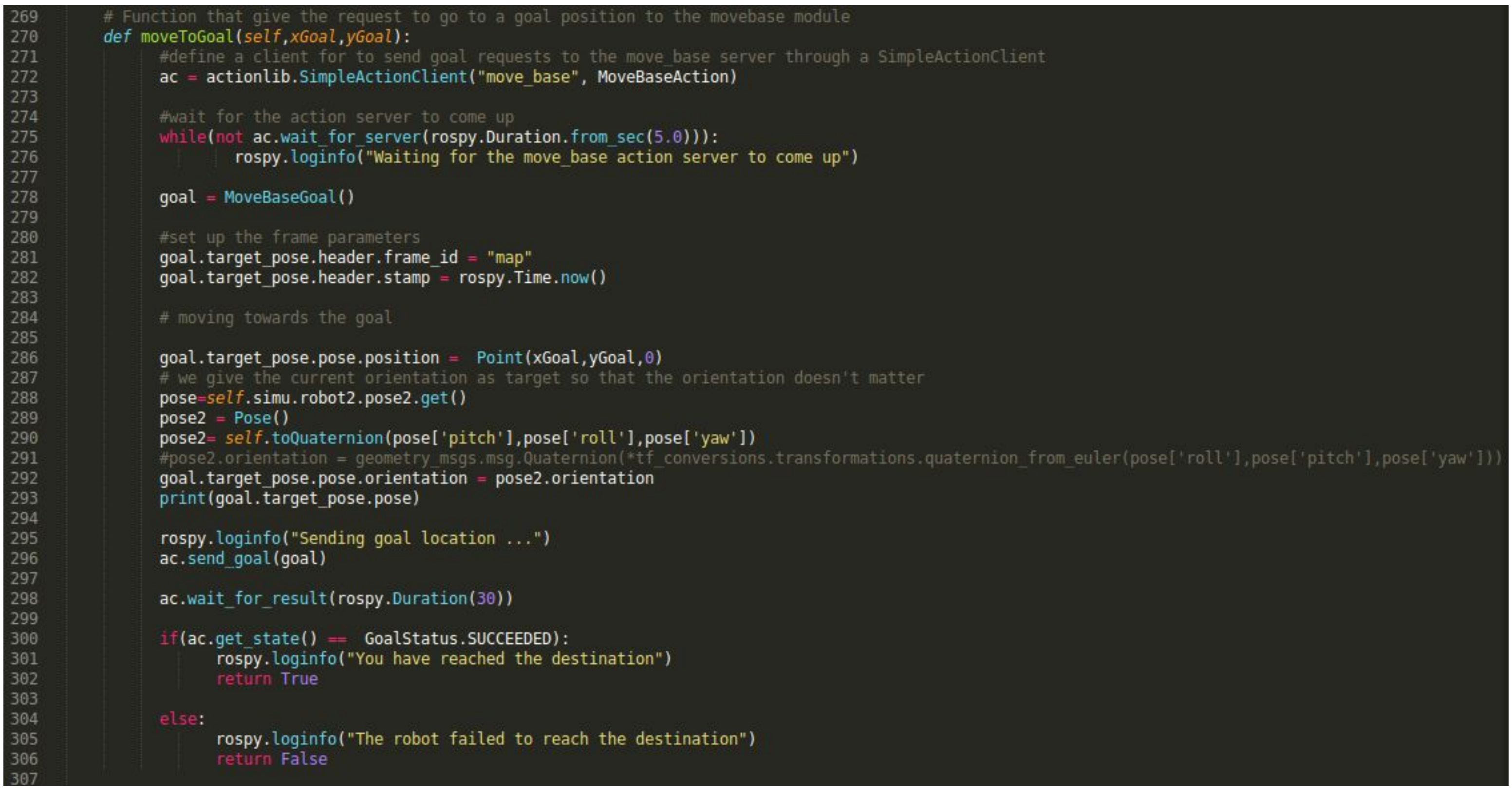
```python
    def pick_new_region(self,line_connexions):
        # we build an array with the ID and a coef proportionnal to the number of connexions of the region
        tab=[[]*2]*len(line_connexions)
        sumTotal=0
        for i in range(0,len(line_connexions)):
            temp=[]
            temp.append(line_connexions[i])
            index=self.search_index(line_connexions[i])
            temp.append(self.num_connexions[index][1])
            sumTotal+=self.num_connexions[index][1]
            tab[i]=temp

        tab.sort(key=lambda colonnes: colonnes[1])
        #print("-------------------------------numbers of connexions---------------------")
        #print(tab)
        add=0
        #we use the array to give the biggest importance (proba to stay) to the room with less connection
        #the logic is that there is less chance to stay in an area with a lot of connexions (like hallways...)
        #the regions with less connexions are more likely to be the destination the human is seeking
        for i in range(0,len(line_connexions)):
            tab[i][1]= 1-tab[i][1]/sumTotal
            add+=tab[i][1]
        count=0
        for i in range(0,len(line_connexions)):
            tab[i][1]= 100*tab[i][1]/add
        import copy
        tab2 = copy.deepcopy(tab)
        #print("-------------------------------Probability corresponding---------------------")
        #print(tab)
        for i in range(0,len(line_connexions)):
            count+=tab[i][1]
            tab[i][1]= count
        tab.sort(key=lambda colonnes: colonnes[1])

        #print("------------------------------- Cumulated Probability--------------------")
        #print(tab)
        pick=random.uniform(0,100)
        i=0
        test = pick > tab[i][1]
        while test and i+1 < len(line_connexions):
            i+=1
            test = pick > tab[i][1]
        return tab[i][0], tab2
```

- **Function *simulation()* in *robot_human_control_morse.py***



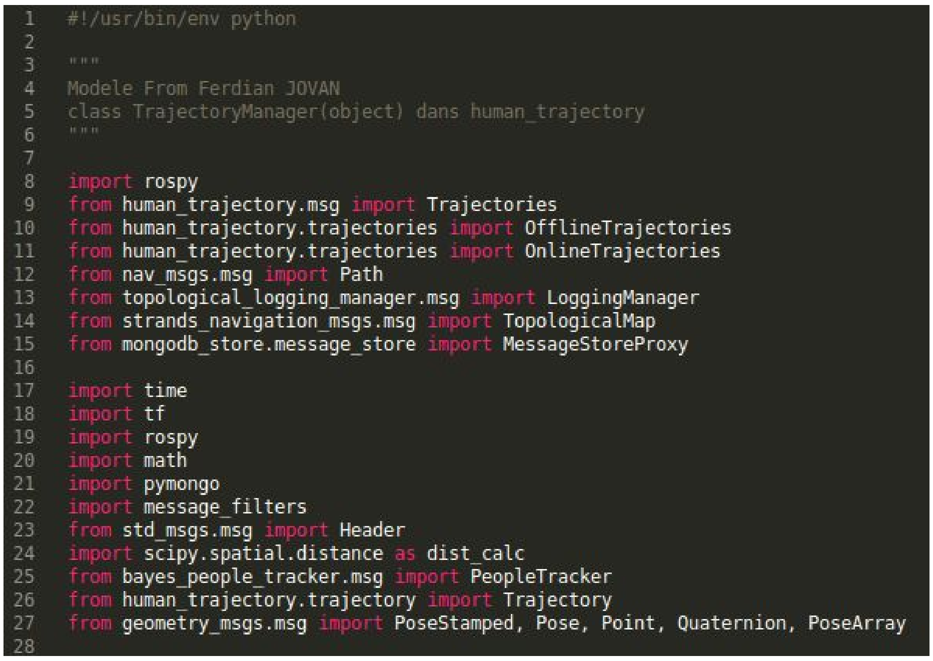
```python
    def simulation(self):

        # subscribes to updates from the Pose sensor by passing a callback
        #simu.robot2.pose.subscribe(print_pos)
        next_id='13'
        choice=self.search_index('13')
        self.simu.sleep(3)

        rospy.init_node('map_navigation', anonymous=False)  #false because we don't want 2 node in the same time
        goalReached = False
        while not rospy.is_shutdown() :
            print('----------------------choice--------------------')
            print(next_id)
            try:
                # landmark change between Blender and RVIZ x_blender=y_rviz+3.8 y_blender=-xrviz-3.8
                # sends a destination by publish
                #self.simu.robot.waypoint.publish({'x': self.centre_data[choice][2]+3.2, 'y': -self.centre_data[choice][1]-3.2, 'z': 0.0,'tolerance': 0.5,
                goalReached = self.moveToGoal(self.centre_data[choice][1],self.centre_data[choice][2])
                if (goalReached) : #and (self.simu.robot.waypoint.get_status() == "Arrived")
                    rospy.loginfo("Congratulations!")
                    next_id,useless=self.pick_new_region(self.array_connexions[choice])
                    choicePrevious=choice
                    choice=self.search_index(next_id)
                    if choicePrevious == choice:
                        rospy.loginfo("I want to stay here for a while ")
                        self.simu.sleep(10)
                        goalReached=True
                    else :
                        goalReached=False
                else:
                    rospy.loginfo("Hard Luck!")

            except rospy.ROSInterruptException:
                rospy.loginfo("map_navigation node terminated.")

```

- **Function *MoveToGoal()* in *robot_human_control_morse.py***

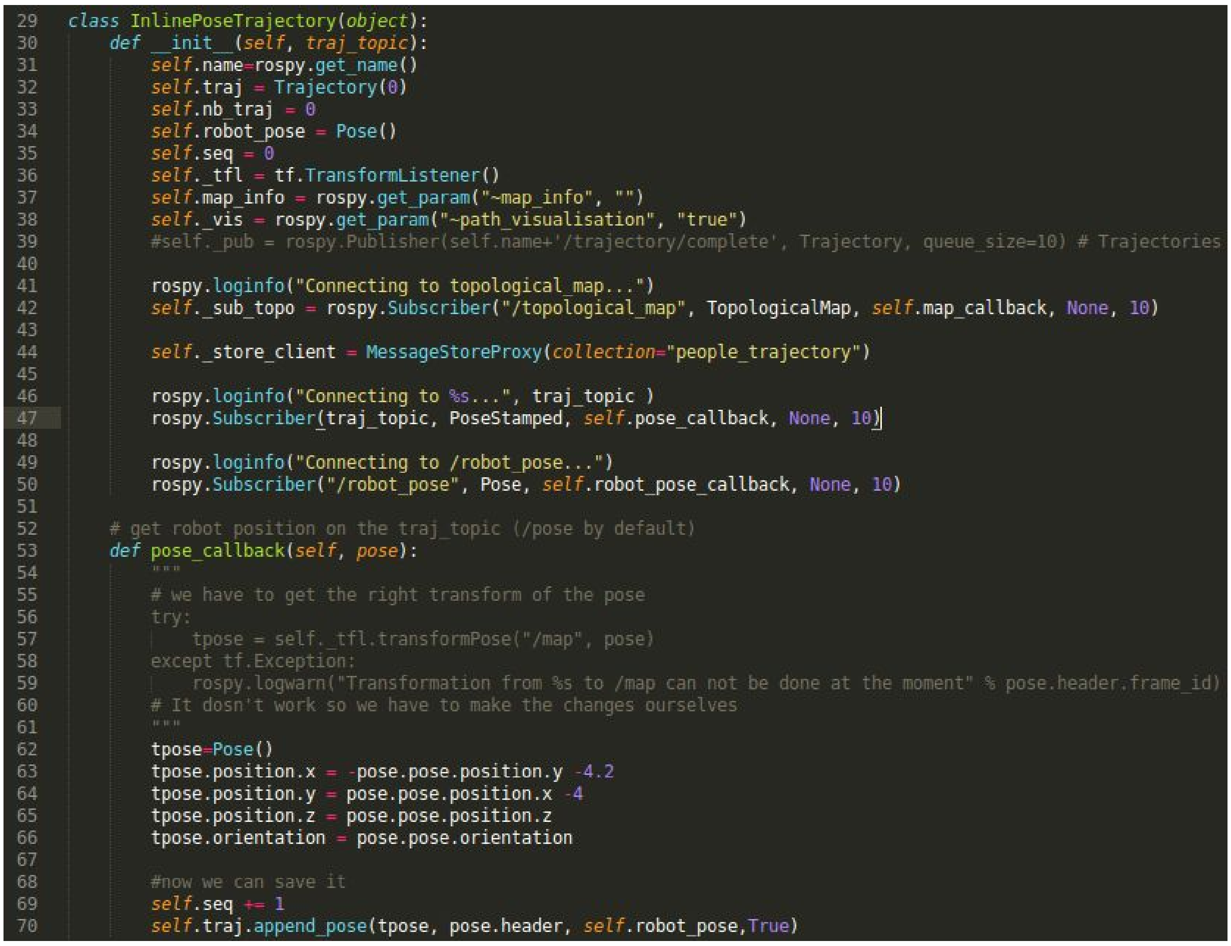


```python
    # Function that give the request to go to a goal position to the movebase module
    def moveToGoal(self,xGoal,yGoal):
        #define a client for to send goal requests to the move_base server through a SimpleActionClient
        ac = actionlib.SimpleActionClient("move_base", MoveBaseAction)

        #wait for the action server to come up
        while(not ac.wait_for_server(rospy.Duration.from_sec(5.0))):
                rospy.loginfo("Waiting for the move_base action server to come up")

        goal = MoveBaseGoal()

        #set up the frame parameters
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()

        # moving towards the goal

        goal.target_pose.pose.position =  Point(xGoal,yGoal,0)
        # we give the current orientation as target so that the orientation doesn't matter
        pose=self.simu.robot2.pose2.get()
        pose2 = Pose()
        pose2= self.toQuaternion(pose['pitch'],pose['roll'],pose['yaw'])
        #pose2.orientation = geometry_msgs.msg.Quaternion(*tf_conversions.transformations.quaternion_from_euler(pose['roll'],pose['pitch'],pose['yaw']))
        goal.target_pose.pose.orientation = pose2.orientation
        print(goal.target_pose.pose)

        rospy.loginfo("Sending goal location ...")
        ac.send_goal(goal)

        ac.wait_for_result(rospy.Duration(30))

        if(ac.get_state() ==  GoalStatus.SUCCEEDED):
            rospy.loginfo("You have reached the destination")
            return True

        else:
            rospy.loginfo("The robot failed to reach the destination")
            return False

```

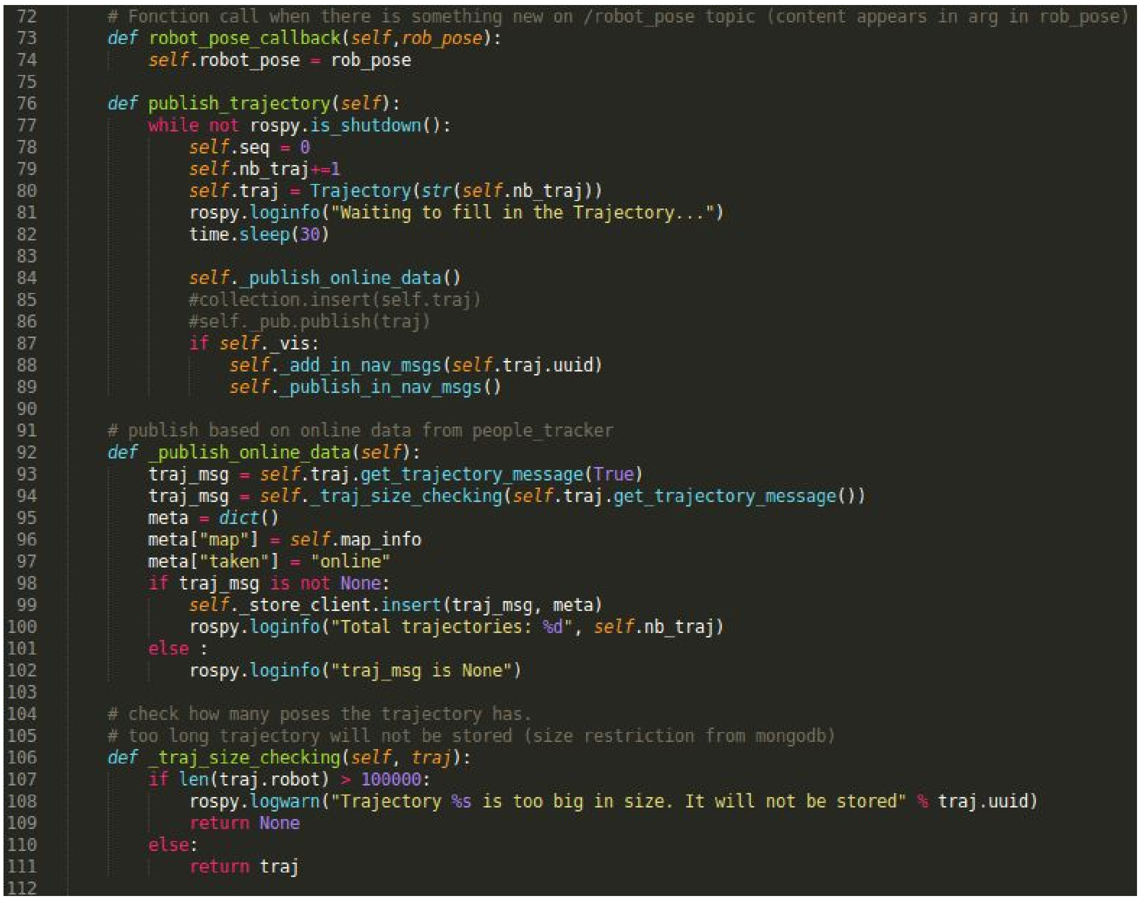
- **Trajectory_to_db.py**



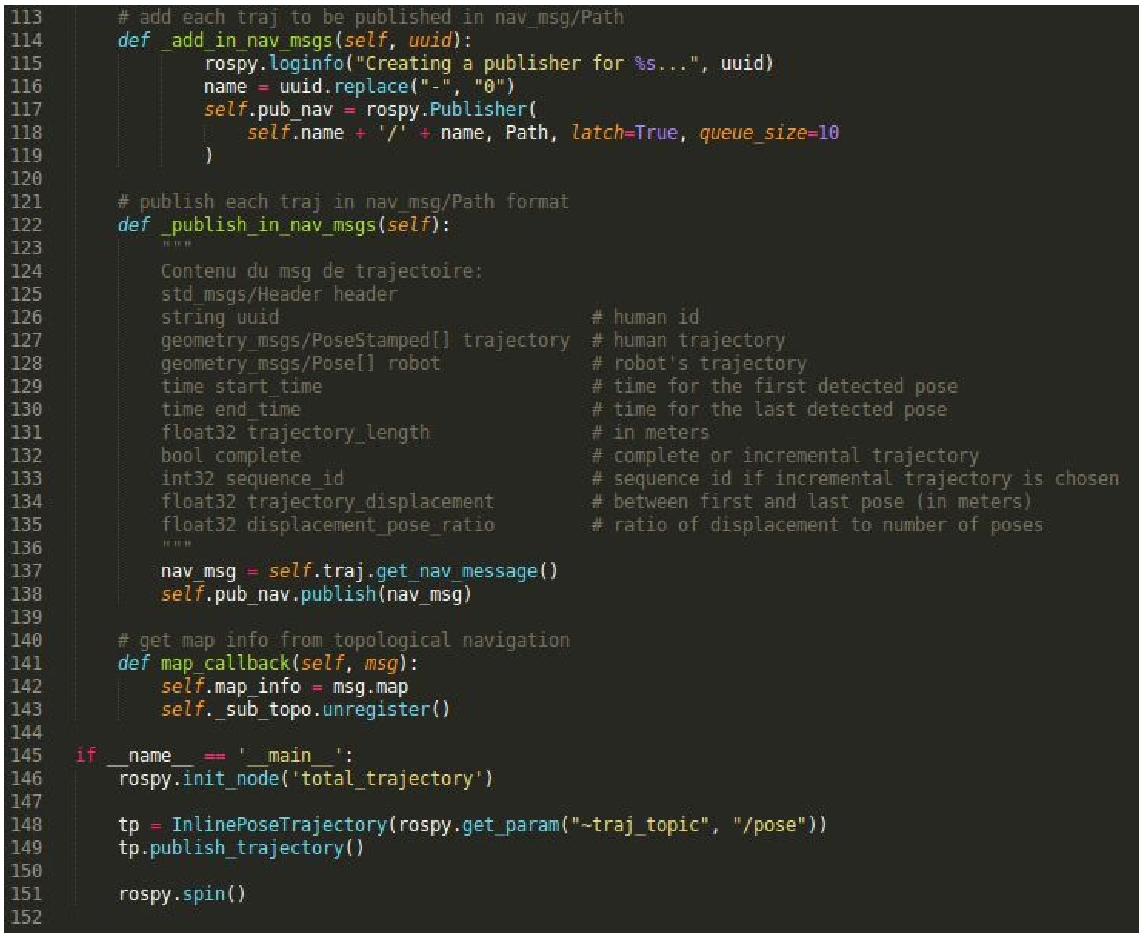
```python
#!/usr/bin/env python

"""
Modele From Ferdian JOVAN
class TrajectoryManager(object) dans human_trajectory
"""

import rospy
from human_trajectory.msg import Trajectories
from human_trajectory.trajectories import OfflineTrajectories
from human_trajectory.trajectories import OnlineTrajectories
from nav_msgs.msg import Path
from topological_logging_manager.msg import LoggingManager
from strands_navigation_msgs.msg import TopologicalMap
from mongodb_store.message_store import MessageStoreProxy

import time
import tf
import rospy
import math
import pymongo
import message_filters
from std_msgs.msg import Header
import scipy.spatial.distance as dist_calc
from bayes_people_tracker.msg import PeopleTracker
from human_trajectory.trajectory import Trajectory
from geometry_msgs.msg import PoseStamped, Pose, Point, Quaternion, PoseArray

```



```python
class InlinePoseTrajectory(object):
    def __init__(self, traj_topic):
        self.name=rospy.get_name()
        self.traj = Trajectory(0)
        self.nb_traj = 0
        self.robot_pose = Pose()
        self.seq = 0
        self._tfl = tf.TransformListener()
        self.map_info = rospy.get_param("~map_info", "")
        self._vis = rospy.get_param("~path_visualisation", "true")
        #self._pub = rospy.Publisher(self.name+'/trajectory/complete', Trajectory, queue_size=10) # Trajectories

        rospy.loginfo("Connecting to topological_map...")
        self._sub_topo = rospy.Subscriber("/topological_map", TopologicalMap, self.map_callback, None, 10)

        self._store_client = MessageStoreProxy(collection="people_trajectory")

        rospy.loginfo("Connecting to %s...", traj_topic )
        rospy.Subscriber(traj_topic, PoseStamped, self.pose_callback, None, 10)

        rospy.loginfo("Connecting to /robot_pose...")
        rospy.Subscriber("/robot_pose", Pose, self.robot_pose_callback, None, 10)

    # get robot position on the traj_topic (/pose by default)
    def pose_callback(self, pose):
        """
        # we have to get the right transform of the pose
        try:
            tpose = self._tfl.transformPose("/map", pose)
        except tf.Exception:
            rospy.logwarn("Transformation from %s to /map can not be done at the moment" % pose.header.frame_id)
        # It dosn't work so we have to make the changes ourselves
        """
        tpose=Pose()
        tpose.position.x = -pose.pose.position.y -4.2
        tpose.position.y = pose.pose.position.x -4
        tpose.position.z = pose.pose.position.z
        tpose.orientation = pose.pose.orientation

        #now we can save it
        self.seq += 1
        self.traj.append_pose(tpose, pose.header, self.robot_pose,True)
```



```python
    # Fonction call when there is something new on /robot_pose topic (content appears in arg in rob_pose)
    def robot_pose_callback(self,rob_pose):
        self.robot_pose = rob_pose

    def publish_trajectory(self):
        while not rospy.is_shutdown():
            self.seq = 0
            self.nb_traj+=1
            self.traj = Trajectory(str(self.nb_traj))
            rospy.loginfo("Waiting to fill in the Trajectory...")
            time.sleep(30)

            self._publish_online_data()
            #collection.insert(self.traj)
            #self._pub.publish(traj)
            if self._vis:
                self._add_in_nav_msgs(self.traj.uuid)
                self._publish_in_nav_msgs()

    # publish based on online data from people_tracker
    def _publish_online_data(self):
        traj_msg = self.traj.get_trajectory_message(True)
        traj_msg = self._traj_size_checking(self.traj.get_trajectory_message())
        meta = dict()
        meta["map"] = self.map_info
        meta["taken"] = "online"
        if traj_msg is not None:
            self._store_client.insert(traj_msg, meta)
            rospy.loginfo("Total trajectories: %d", self.nb_traj)
        else :
            rospy.loginfo("traj_msg is None")

    # check how many poses the trajectory has.
    # too long trajectory will not be stored (size restriction from mongodb)
    def _traj_size_checking(self, traj):
        if len(traj.robot) > 100000:
            rospy.logwarn("Trajectory %s is too big in size. It will not be stored" % traj.uuid)
            return None
        else:
            return traj

```



```python
    # add each traj to be published in nav_msg/Path
    def _add_in_nav_msgs(self, uuid):
            rospy.loginfo("Creating a publisher for %s...", uuid)
            name = uuid.replace("-", "0")
            self.pub_nav = rospy.Publisher(
                self.name + '/' + name, Path, latch=True, queue_size=10
            )

    # publish each traj in nav_msg/Path format
    def _publish_in_nav_msgs(self):
        """
        Contenu du msg de trajectoire:
        std_msgs/Header header
        string uuid                             # human id
        geometry_msgs/PoseStamped[] trajectory  # human trajectory
        geometry_msgs/Pose[] robot              # robot's trajectory
        time start_time                         # time for the first detected pose
        time end_time                           # time for the last detected pose
        float32 trajectory_length               # in meters
        bool complete                           # complete or incremental trajectory
        int32 sequence_id                       # sequence id if incremental trajectory is chosen
        float32 trajectory_displacement         # between first and last pose (in meters)
        float32 displacement_pose_ratio         # ratio of displacement to number of poses
        """
        nav_msg = self.traj.get_nav_message()
        self.pub_nav.publish(nav_msg)

    # get map info from topological navigation
    def map_callback(self, msg):
        self.map_info = msg.map
        self._sub_topo.unregister()

if __name__ == '__main__':
    rospy.init_node('total_trajectory')

    tp = InlinePoseTrajectory(rospy.get_param("~traj_topic", "/pose"))
    tp.publish_trajectory()

    rospy.spin()

```