Report Laboratory Mission coordination.

Table des matières

Introduction:

In recent years, we have observed significant advancements in technological fields such as artificial intelligence, the computing power of embedded systems, and the development of more efficient sensors. These innovations have led to the emergence of autonomous systems. Such systems are widely used and developed in fields like robotics and aeronautics. Their development is crucial for carrying out dangerous missions or accessing areas that are unreachable by humans.
It is in this context that the concept of coordinated missions has emerged. It allows for the control of fleets of Unmanned Ground Vehicles (UGVs) or Unmanned Aerial Vehicles (UAVs) to perform tasks such as material delivery or area surveillance.
During this practical work, we will first explore the ROS software, which is widely used in robotics, and then attempt to create robust missions.

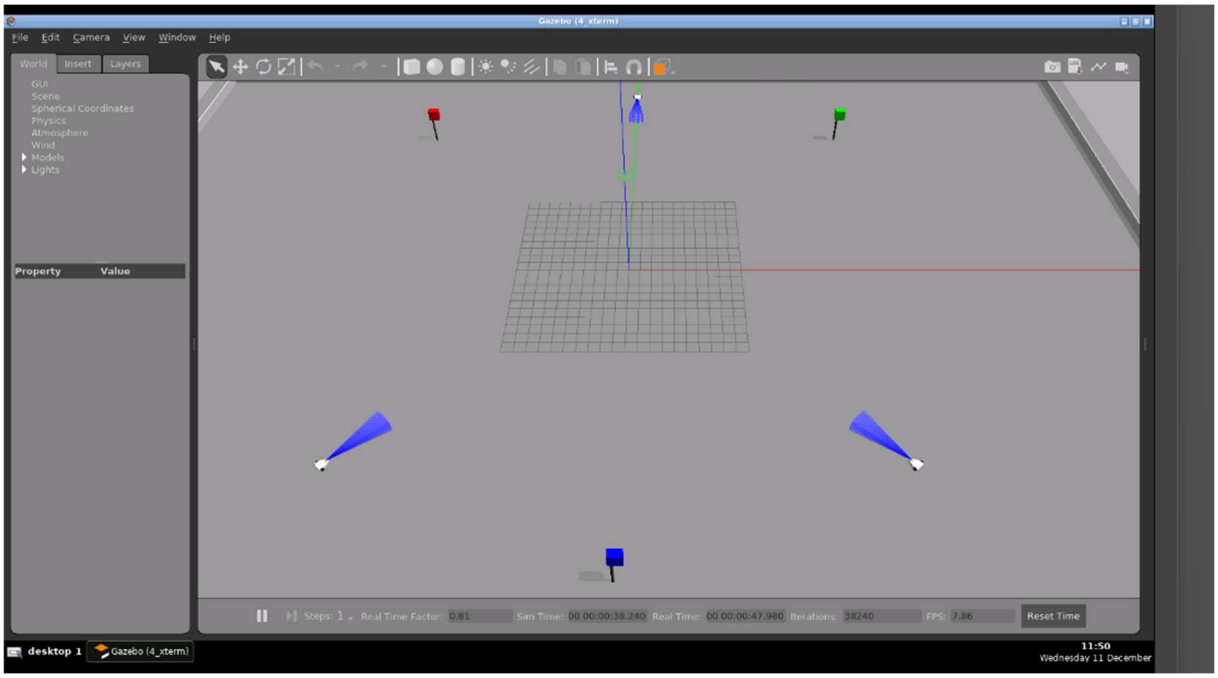
Laboratory 1:

First: Gets started.

We begin to run the simulation with the current command :

roslaunch evry_project_description simu_robot.launch

And we obtain this :



This represents a area with 3 robots and with 3 flags (one green , one red and one blue). Now in a second terminal, we run a second script containing a strategy :

roslaunch evry_project_strategy agent.launch nbr_robot:=1

Q1: When we run the command and the script , we can observe that the robot located in the right side begin to move straight forward. In the terminal some values are display. This value corresponds to his distance from the red flag.

Step 2: Visualisation of some concepts.

Now in this second terminal, we will execute some instruction :

Q2 : We will use the command rostopic list. In theory , this command will display a list of active topic. (Recall : a topic is a chanel of communication used in ROS to send some message between node.) So we will use it and observe the result :

```
user:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/robot_1/cmd_vel
/robot_1/odom
/robot_1/sensor/sonar_front
/robot_2/cmd_vel
/robot_2/odom
/robot_2/sensor/sonar_front
/robot_3/cmd_vel
/robot_3/odom
/robot_3/sensor/sonar_front
/rosout
/rosout_agg
/tf
user:~$
```

Theses result show that we have 3 robots (1 ,2 and3) , each robot has 3 topic :

- One for the control: cmd_vel (maybe velocity)

- Second for the localisation: odm
- Third for the sensor: sonar_front

Now we will try to gain some information about the sonar topic. We will use the command rostopic info /robot_1/odom.

Q3 : After executing this command, we obtain this :

```
user:~$ rostopic info /robot_1/odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://4_xterm:40517/)

Subscribers: None
```

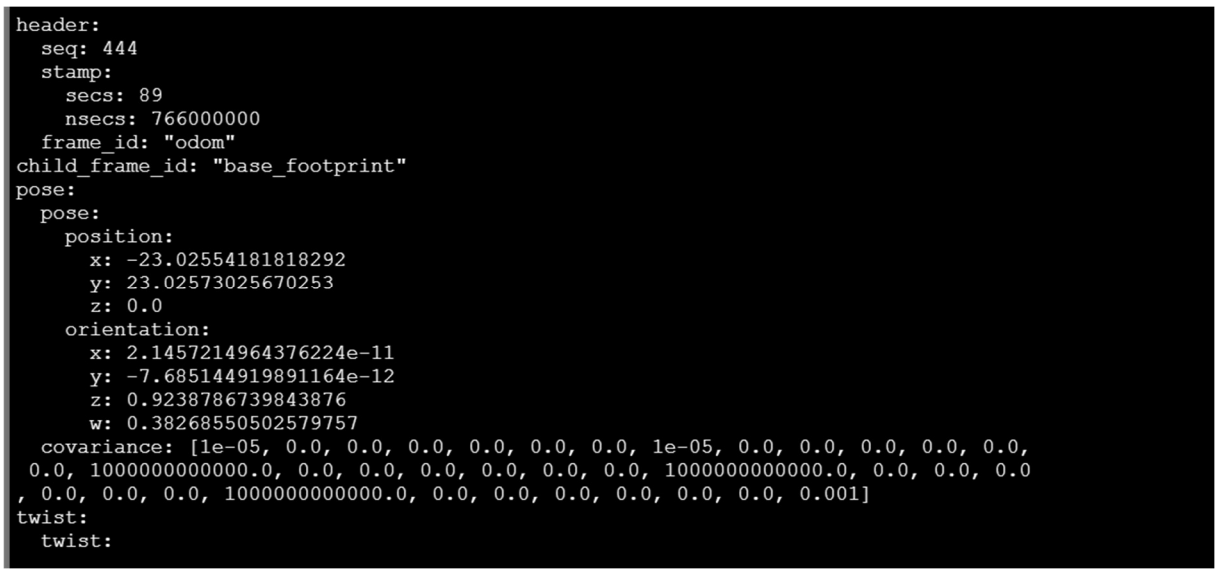
We can observe that the publisher is : /gazebo (http://4_xterm:40517/) and it doesn't have a subscriber.

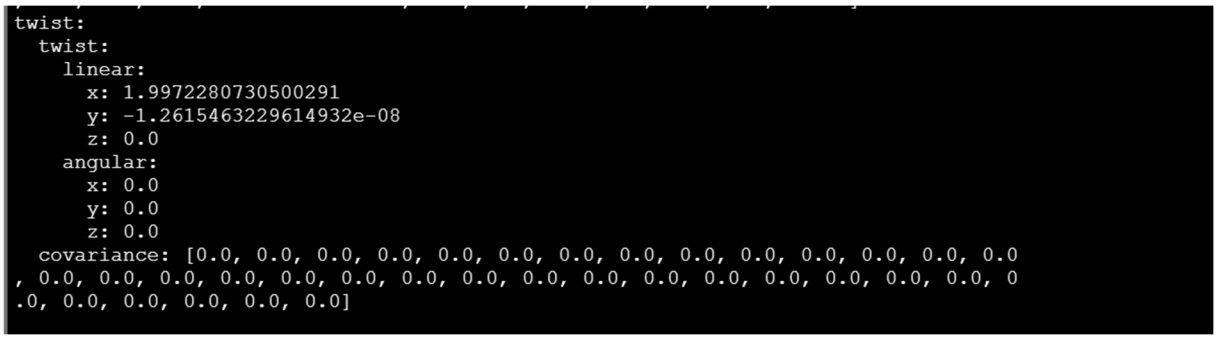
Q4 : We can also the type of message : nav_msgs/Odometry. We used the command osmsg info nav_msgs/Odometry. We obtain this

```
user:~$ rosmsg info nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
```

This type is for the navigation and localisation of the robot.Its contain the position , the orientation , the speed and the incertity.

Q5: We will listening to the topic. For we will use this command: rostopic echo /robot_1/odom.
The command echo is used to display in real time the message publish in a topic inside a terminal. It's useful because it can allow us to check if the topic is working well. So this is what we obtain executing this command :





It's display some information related to the robot. It display :

- His speed
- His position
- His orientation

Step 3: Move one robot.

In this part, we will see how to make the robot move. For that, we acced to the file agent.py. We use the command : cd ~/catkin_ws/src/Mission_Coordination_project/evry_project_strategy/nodes/
To obtain the access of the script agent.py. To modifie it we can use the command nano agent.py.
This file contain a file with a class call Robot which can :

1. get the pose of the robot (with *get_robot_pose()*);
2. get the value of the front sonar (with *get_sonar()*)
3. get the distance to its own desired flag (with *getDistanceToFlag()*)
4. set the desired velocity and angle (with *set_speed_angle(speed, angle)*)

For that part, we can modify the behaviour of the robot with the main function run demo().
At first, we change the velocity and his valocy. In the github some video are provide to display the result if we change the velocity or the angular velocity.

Recall : To reset the position we used this command : rosservice call /gazebo/reset_world.

Step 4 : Move one robot to the corresponding flag.

Inside the class, we can observe that the robot moves to a designed flag. Now we can modify the program to stop the robot to the designed flag.

Q7 : In this part, we will implement a PID controller :
The code for the PID controller is giving inside github.

Step 5: Implementation of one strategy – Timing solution

We were able to make one robot move but was happen with all the 3 robots? If the 3 robots move at the same time they crash each other. So, to avoid this we need to implement a strategy that will make the robot avoid each avoid. Here, we will implement the timing solution. The timing solution is an approach who allow us to plan the decision or action of the robot.

Laboratory 2:

During the previous lab, we explored the functionality of the ROS software through the example of a robot. We were able to make it follow a trajectory and eventually implemented a strategy allowing three robots to follow their trajectories and reach their respective flags.
However, this strategy remains simple and inefficient, as in any environment, the robots may encounter obstacles. Our initial strategy lacks robustness. Here, we will attempt to implement two new robust strategies to coordinate our mission. We will implement an initial strategy, and then we will attempt to implement the Artificial Potential Field strategy. We have placed an obstacle at the center of the map, and the goal of this mission is to reach the flag while avoiding the obstacle.

First strategy: Reactive Obstacle Avoidance.

We start with a simple strategy called reactive obstacle avoidance. The steps of this strategy are as follows:

- Detect the obstacle (using the sensor).
- Avoid the obstacle after detection (for instance, by turning).
- Resume the initial trajectory once the obstacle is bypassed.

Conclusion:

Unfortunately, we were unable to implement a robust control method that allows avoiding an obstacle while reaching the flag. We attempted to implement two techniques, *Artificial Potential Field* and *Reactive Obstacle Avoidance*, but we did not achieve conclusive results that allowed obstacle avoidance. However, we plan to continue this work later to successfully implement these techniques, as studying this topic is fascinating.