UNIVERSITY OF BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Autonomous Mobile Robotic M

FINAL PROJECT: SANITIZER ROBOT

Professor: **Palli Gianluca**

Tutors: **Galassi Kevin, Zanella Riccardo**

Students: **Daniele Simonazzi**
Nicoló Musolesi
Francesco Losi

Academic year 2022/2023

Contents

Chapter 1

First and Second Task: Simulation Set-up and Autonomous Mapping

The primary objective of the first task is to set up a simulation of the TurtleBot3-Burger robot using the Gazebo environment, with the Big-House scenario already loaded. In the second task, the goal of the robot is to autonomously map an unknown environment, which, in this case, is once again the Big House environment.

1.1 What is a Launch File

In ROS2, a launch file is a configuration file that defines how to start a set of ROS nodes and configure their parameters. A Python launch file is written using the launch API in Python, instead of the traditional XML format. It typically starts with importing the necessary modules, such as os and launch, and then creates a launch description object to define the nodes and their parameters.

1.2 Our Launch File

We used the amr_proj workspace and created the task1_2_bringup package within it. In the launch folder of this package we placed our launch file, which launches:

- Gazebo, the standalone simulation application, with the Big House map and the TurtleBot3-Burger already loaded.
- the nav2 launch file bringup_launch.py, which launches several useful nodes in the nav2 library, along with the task1_2_nav2_params.yaml parameter file.

- the async_slam_toolbox_node from the slam toolbox package, which handles the localization and mapping of the environment, using an ad-hoc parameter file named task1_2_mapper_params_online_async.yaml.
- the rviz2 node with a custom rviz-configuration to visualize the path, the automatically updating map, the costmap and other useful plug-ins.

1.3 The explore-lite package

Once the simulation environment was set up, the next step was to explore and map the Big House environment. We used an already-implemented package called explore-lite, which is a port of a well-known ROS package called m-explore. At present, the package is still in development and has some instability issues that can sometimes lead to map corruption, incorrect identification of the frontiers and premature termination of the exploration process. However, it is the fastest and smartest package we found in terms of performing the exploration process.

The package revolves around the concept of "frontiers," which are defined as boundaries between known and unknown areas in the environment. Each frontier is given a score based on a combination of different parameters, such as distance from the robot and size. The frontier with the highest score is classified as the main frontier. The robot navigates to the main frontier, sets it as explored target, selects a new main frontier, chases it, and repeats the process until the entire map is explored. Some frontiers are classified as not-explorable and are blacklisted.

Some of the most important parameters we played with are:

- frontiers with size smaller than min_frontier_size [1] are discarded. We reduced min_frontier_size to facilitate the exploration behavior of the robot.
- when the robot, chasing a frontier, does not make any progress for progress_timeout [2] seconds, the current goal is abandoned and a new one is seleceted. We reduced progress_timeout since, having changed min_frontier_size, there are now way more frontiers and we do not want the mapping process to be extremely long: if it takes too much to reach a frontier, abandon it.

[1] inside package explore-lite
[2] inside package explore-lite

- we reduced the robot velocity[3], high values were responsible of many of the episodes of map corruption.
- we increased the inflation_radius[4] parameter to obtain a safer exploration behaviour. The Lidar is indeed mounted on the head of the robot and it is not capable of sensing obstacle which are below a certain height, as for example the bases of some tables.
- we brought the length of the Lidar-rays[5] from 3.5 m to 5.5 m. This choice was a consequence of the fact that if a ray does not hit an obstacle within 3.5 m, the map is not updated. This is a problem because, when the robot gets close to a door, there is the possibility that the Lidar does not sense the opposite wall. If this happens then the map is not updated, no new frontier is detected and the robot does not enter the room.

1.4 Possible upgrades

We are aware that our solution is not flawless. Specifically, the decision to increase the range of the Lidar was forced by the fact that the exploration algorithm has clear limitations and is unable to handle all possible situations that may occur. One possible alternative would require external intervention to support the exploration algorithm: when mapping of the Big House is complete without the entire map being explored, the robot could be moved to the nearest unexplored room and the exploration algorithm relaunched. If this is not acceptable and a better solution is desired, it would be necessary to modify the exploration algorithm itself. As previously specified, the package is still under development and we are confident that these issues will be resolved in the future.

[3] inside task1_2_nav2_params.yaml
[4] inside task1_2_nav2_params.yaml
[5] inside turtlebot3_gazebo/models/turtlebot3_burger/model.sdf

Chapter 2

Third Task: Autonomous Localization and Navigation

Task objective: spawn the robot in a random room, perform localization, and then launch navigation towards pre-defined goals. This problem is also known as the kidnapped robot problem in the scientific community.

2.1 Our three Launch Files

In our three Launch Files, the first launch file sets up the environment in which the robot will localize and navigate (task3_bringup.launch.py). The nodes launched are the same as those launched in the task 1 and 2 launch files, except for the SLAM navigation toolbox which is not necessary.

The second launch file partially comes from a package we downloaded from GitHub called "wall-follower" (wall-follower-left.launch.py). We added an additional node responsible for re-initializing the robot's localization. In particular, the node (client) requests the "reinitialize_global_localization" service provided by the server of the AMCL node. The "wall-follower" package implements autonomous navigation which we use to carry out the autonomous localization process of the robot. The need for the "wall-follower" package was motivated by the inability to use the previous package "explore-lite" which relied on the use of frontiers that do not exist in an already explored map.

The third and final launch file launches the node responsible for autonomous navigation towards the input goals (task3.launch.py).

2.2 Localization

Once the environment is initialized and the robot is positioned in a random position on the map, we proceed to execute wall-follower-left.launch.py from the terminal. First of all, the localization re-initialization service will be requested and executed, which will distribute a specified number of particles uniformly across the map as specified in the task3_nav2_params.yaml file. This number of particles will affect the speed and efficiency of the localization process as well as the computational cost of the process. After a tuning process, we opted for a maximum number of particles of 8000 (selected based on the size of the map).

Subsequently, the robot_controller_left.py and robot_estimator.py nodes are launched, which will allow the robot to move safely and localize itself. In robot_controller_left, we inserted a check that was not present in the original file, which terminates the motion of the robot when the position estimate is sufficiently accurate. In practice, robot_controller_left subscribes to the amcl_pose topic, extracts the pose (position, orientation, and covariance matrix) through a callback (defined by us), and then saves it in a temporary variable. The check we introduced is based on verifing that the eigenvalues of the covariance matrix are less than a certain user-defined threshold (empirically tuned). When this occurs, robot_controller_eft terminates the navigation of the map and publishes a flag equal to True in a topic defined by us called LocalizationCompleted.

2.3 Navigation

The node launched by task3.launch.py waits for the flag published in LocalizationCompleted to become True. When this happens, the node reads the goals from the task3_nav_goals.yaml file. Each goal is characterized by a first triplet of values that represents the position and a second triplet of values that represents the orientation. Once read, the goals are given as input to the follow_waypoint action provided by the waypoint_follower node within the nav2 package. This action allows the robot to navigate the map using the Dijkstra algorithm. Nav2 provides the possibility to select between Dijkstra (default option) and A*. We opted for Dijkstra since our map is not excessively big, for larger maps it is suggested to use A* as it provides faster computation than Dijkstra.

2.4 Possible upgrades

It is worth noting that the localization process is not flawless, and this is because the wall_follower algorithm has some limitations. In particular, the robot always tries to keep an obstacle on the left, if the obstacle is a wall

everything works fine. However, if the obstacle is, for example, the leg of a table, then the robot loops around it forever, preventing the localization process from being optimal. Therefore, in all those cases where the robot is initialized either far from a wall or near a central obstacle, the localization may fail.

A possible upgrade may be to write an ad-hoc "wanderer" algorithm that guides the robot in navigating through the rooms. In this way, the robot is exposed to a wide variety of obstacles/rooms, and the localization becomes more robust.

Chapter 3

Fourth Task: Room Sanification with Reinforcement Learning

The objective of this task is to develop a policy for a robot to sanitize a generic room from Coronavirus.

The maps in which the robot navigates are discretized into square cells of side length 20 cm. The robot is equipped with a UV lamp that emits sanitization rays at 360 degrees. The specification sheet of the lamp provides us with the formula to compute the power per-square-meter which reaches a point at a certain distance from the robot. To compute the total energy a cell has been exposed to we should run an integral which takes into account both the size of the cell and time. Here, for simplicity, we assume that each cell, at each step[1] taken by the robot in the environment, is exposed to an energy equal to $1/distance^2$ (independently from size)[2]. Furthermore, we say that a cell is completely sanitized when it has accumulated an energy of at least 3. We also assume that obstacles completely stop ray propagation and that sanification rays do not reach the cell the robot is currently in.

The RL problem is tackled by setting as state of the robot, also known as the observation, a 9x9 grid centered on the robot. Each cell in the grid is either an obstacle, a partially sanitized cell, or a completely sanitized cell. The agent learns which action to associate with each state to maximize the cumulative reward at the end of an episode. An episode consists of the robot

[1] In our discrete environment, time is marked by the steps taken by the robot. This is equivalent to the robot spending 1 second in each cell and then moving instantaneously to the next one."

[2] We actually do not care about the measurment unit, it can be whatever measurment unit for energy

moving around a randomly-generated map until 800 steps are completed, or the robot successfully sanitizes the whole map. Since the focus is shifted over the RL-aspect of the problem and not on realistically modelling the motion of the robot, we completely neglected its kinematics/dynamics and we model it as an immaterial point moving around the map.

3.1 The learning environment

We created the learning environment using Gym, a standard API for reinforcement learning.

3.1.1 The maps

The maps are 25x25 maps with randomly-generated rectangular obstacles. The location and size of the obstacles are random, exposing the agent to the most heterogeneous set of states during the learning process. We also randomized the initialization of the robot.

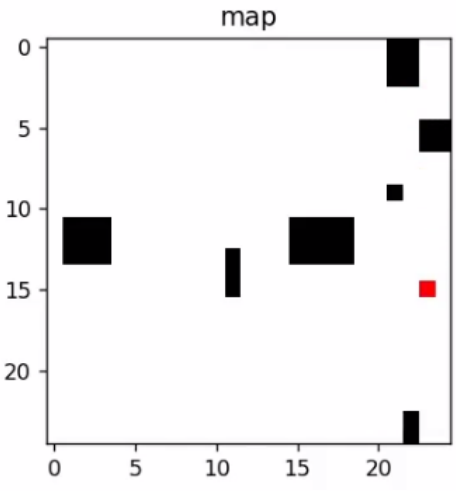

Figure 3.1: Example of a randomly generated map, the position of the robot is highlighted in red

3.1.2 The sanification process

The sanification process involves a function, called cast_ray, which computes the shadows cast by obstacles in the 9x9 window. The sanification process is applied only to the cells that belong to the local window and do not fall inside a shadow region (the central cell is always classified as shadow since the robot is there). Even though the sanification process is done inside the local window, we update the energy values of a global map called sanitized_map.

It is worth pointing out that the dimensions of both the map and the local window are the result of an "empirical tuning" that will be explained later (see 3.3).

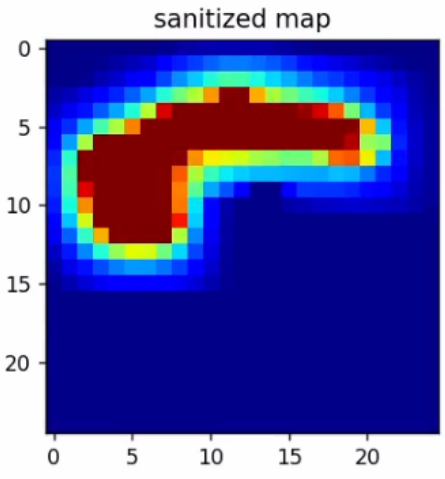

Figure 3.2: Example of sanitized_map

3.1.3 The observation

The observation is a simplified and flattened version of the 9x9 local window of the robot extracted from the sanitized_map (not from the map, we need the energy values). Each cell belongs to one of the following three classes: obstacle cell, not completely sanitized cell, or completely sanitized cell. To create the observation, we assign a number (0, 1, or 2) to each class and substitute the energy value of a cell with the corresponding class number.

3.1.4 The rewards

The rewards are a critical aspect of the learning process. Originally, we had the following set of rewards:

- -1 for each step in the environment.
- +3 for each new completely sanitized cell.
- +100 if the robot successfully sanitizes the whole map.
- -10000 if the robot either hits an obstacle, goes outside the map or does not sanitize the whole map within 800 steps, moreover the episode ends.

However, we realized that the agent had a really hard time learning a good policy, thus we introduced the following modifications:

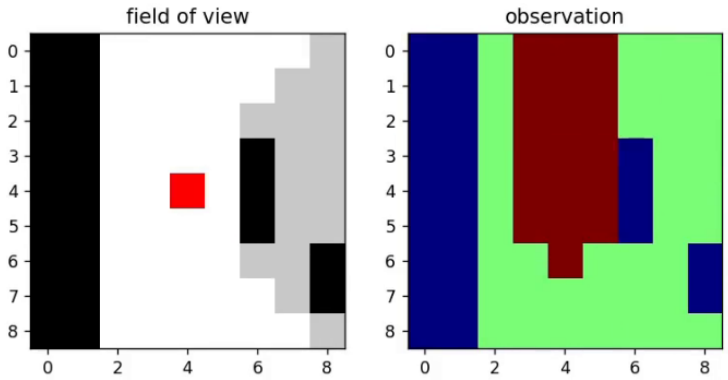

Figure 3.3: On the left: a 9x9 window centered in the robot extracted from the map, shadows are highlighted in gray. On the right: the observation, where red is for completely sanitized cells, green for partially sanitized cells, blue for obstacles

- -10 if the robot either hits an obstacle or goes outside the map, no negative reward if the episode terminates with a partially sanitized map. The intuition is that the robot shouldn't be penalized too much for making an error, otherwise it doesn't make any progress.
- The only circumstance in which an episode ends is when 800 steps are reached, no negative reward is given when this happens. The intuition is that the step penalty is already sufficient to convey the message that the shorter the path, the better.
- The robot is brought back to the cell antecedent the wrong move if it hits an obstacle or exits the map, as it should be allowed to continue navigation even after an error in order to collect more training samples on the same map

With this new set of the rewards, we obtained the result we present in this report.

3.1.5 The action space

The robot is allowed to perform five different actions:

- move up
- move down
- move right
- move left
- stay idle

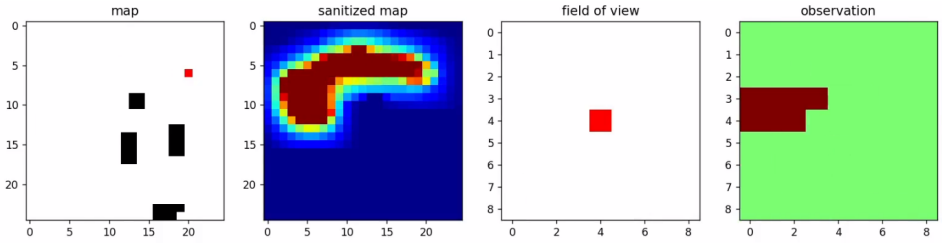

Figure 3.4: What the render() function shows during an episode

3.2 The training

Once the environment was set up, it was time to decide how to train the agent.

3.2.1 Choice of the algorithm: PPO vs DQN

We explored algorithms provided by the well-known Python library, Stable-Baselines3. Stable-Baselines3 is a set of high-quality, reusable implementations of reinforcement learning algorithms in Python built on top of the popular deep learning library TensorFlow.

The two main algorithms used to solve RL problems are PPO (Proximal Policy Optimization) and DQN (Deep Q-Network). We searched a lot online, trying to understand the advantages and disadvantages of each of these two algorithms. We found that PPO is generally more stable than DQN, however, since this was the first time we were using RL algorithms, we weren't able to clearly state which one was best suited for our task. Therefore, we tried both in parallel. The algorithm which provided the expected results first was PPO. Hence, we present the results obtained with the PPO algorithm.

To conclude this section, we report some of the most important differences between PPO and DQN algorithms in StableBaselines3:

- Policy optimization vs. value optimization: PPO is a policy optimization algorithm, while DQN is a value-based optimization algorithm. PPO directly optimizes the policy without explicitly estimating the value function (actually, the StableBaselines3 implementation of the PPO algorithm also estimates the Q-values, however this is a peculiarity of StableBaselines3). In contrast, DQN is a value approximation method aimed at learning the Q-value function by iteratively updating an estimate of the Q-value function based on the observed rewards and transitions.

- On-policy vs. off-policy: PPO is an on-policy algorithm, meaning that it learns from the current policy by exploring the environment and collecting new data. DQN, on the other hand, is an off-policy algorithm that uses a replay buffer to store past experiences and learn from them. In other words, the DQN algorithm learns from experiences generated by a behavior policy that is different from the policy being optimized.
- Actor-critic vs. Q-learning: PPO is an actor-critic algorithm, which means that it learns both a policy (the actor) and a value function (the critic) simultaneously. DQN, on the other hand, just learns the optimal Q-value function.
- Exploration vs. exploitation: PPO encourages exploration by adding noise to the actions, while DQN relies on epsilon-greedy exploration to balance exploration and exploitation.

3.2.2 Choice of Network Architecture: MLP vs CNN

The two main types of neural networks used in learning the policy (we are focusing only on PPO, we have discarded DQN) are MLP (multi-layer perceptron neural network) and CNN (Convolutional Neural Network). The CNN is well-suited for handling images. However, the Stable Baselines implementation requires an input image of minimum size 36x36, which is considerably larger than our observation of 9x9. As a consequence, we opted for the MLP architecture, which requires as input a vector, hence the need to flatten the observation.

3.2.3 Hyperparameters Tuning

Hyperparameters tuning is recognized by many as what distinguishes successful training from failure. Originally, we thought of using a grid-search method to look for the best parameters to use. However, we quickly realized that given how long a single training session took, it was impossible to follow that path. As a consequence, the majority of the hyperparameters we used are the default ones, and we changed only those which seemed to have the most impact on the training. The hyperparameters we changed are:

- learning_rate: The default value is 0.0001. We increased it to 0.001 because we found the default value to be too small, and we hoped to achieve faster convergence.
- gamma: The default value is 0.99, but many people online reported issues with using this value and suggested using 0.9 instead. We followed this recommendation.

- n_steps: This parameter determines the number of steps the agent takes in each environment interaction before the model is updated. It represents the number of steps the agent performs before updating its policy.
- batch_size: This parameter determines how many samples are used in each gradient ascent update. It represents the number of samples the agent learns from in a single update. For example, if we set n_steps to 100 and batch_size to 50, and we train the agent for 1000 steps, we randomly select 50 of the last 100 steps to update the current policy. A sample used in training the agent is typically a tuple consisting of the current state, the action taken in that state, the reward received, and the resulting next state.
- n_epochs: This parameter specifies the number of times the same batch of data is used to train the neural network. In other words, after collecting a batch of data with size batch_size, the neural network is trained for n_epochs times using the same batch of data.

  It is noticeable that the training process was divided into two halves. In the first half, we used a very high n_steps and batch_size of 500000, and we trained the agent for 2 million steps with n_epochs = 40. The reason for this choice was to provide a strong initialization for the gradient ascent method. We know that gradient ascent/descent methods can fail because of the gradient zig-zagging around, failing to point correctly toward the ascent/descent direction. We also know that the more samples used in the computation of the approximated gradient, the more precise the approximated gradient is. For these reasons, we decided to collect four batches of 500000 samples and use each batch 40 times to update the policy. In the second half, we retained the same intuition, but we relaxed the dimensions of batch and n_steps, setting them to 50000. With this specific training, we obtained the following behavior:

3.3 The empirical tuning of map and local window dimensions

Originally, we approached the problem from a global point of view, which resulted in a global observation rather than a local one. This involved providing the state of the entire map and the position of the robot as the observation. In a global set-up, it made sense to use the PPO algorithm in combination with a CNN architecture, especially because the CNN architecture is specific to handle images. However, we found that even for relatively small maps, the learning process was unreasonably slow. Therefore, we transitioned to using a local observation, and as a consequence we

needed to switch to a MLP architecture (the CNN architecture has a constraint on the minimum size of the image, which is 36x36). The challenge with a local observation is that the robot doesn't know its position within the map, so the local window must be sufficiently large to guide the robot's behavior. The robot learns to chase areas of the map where there are still unsanitized cells, and if the local observation window is too small, the robot cannot see far enough ahead. When the robot's surroundings are completely sanitized, it does not know what action to take, even if there are still unsanitized regions in the map. After experimenting with various configurations, we found that the sweet-spot for our system was a map size of 25x25 and a local observation window of 9x9.

3.4 Possible upgrades

The best possible approach would be to use the PPO algorithm in combination with the CNN architecture and a global observation. However, as previously specified, this was not feasible due to the limited computing power of our machines. With more computing power at our disposal, it would be interesting to see the resulting policy.