Laboratory of Industrial Robotics M

Exploration, mapping and planning for an autonomous robot in a closed environment

CERRI FRANCESCO 951972

23 NOVEMBER 2021

INTRODUCTION – PACKAGE AND CODE ANALYSIS

The following simulations are performed inside a Docker Container running Ubuntu 18.04 and ROS Melodic, while the chosen language is Python 2.7 .

I created a single personal ROS package called **navigationturt**, in parallel with the already existing turtlebot3, turtlebot3_msgs, turtlebot3_simulations even thou I decided to import some files I modified directly inside my package (e.g. costmaps' configurations files, Navigation Stack launchers).
Here follows a detailed description of navigationturt directory tree's folders:
navigationturt/

-action/ : actions' folder, unused

-config/ : move_base configuration

-launch/ : specific task launchers, main launcher (godspeed), task launchers(exploration, navigation), some unused

-include/ : simulation, move_base related and teleoperation launchers

-maps/ : automatically generated and specific maps files

-msg/ : contains Setpoint message definition

-rviz/ : rviz panel configuration

-scripts/ : python programs and libraries for all tasks

-unused/ : tests and TODO code

The main launcher is godspeed.launch which defines all parameters for the simulation and calls the specific nodes and launchers for each task of interest, with respect to the used parameters: it is possible to personalize opened windows (Rviz, Gazebo), robot spawn location, robot model, map and objectives file location, operation mode and relative settings. More about the used parameters, nodes and lauchers will be said in each specific task.

The specific launchers (taskJ.launch) for all tasks simply calls godspeed with an adequate pre-set of parameters.

Note that in the following simulations, Rviz windows is opened by default and Gazebo is closed, since the latter tends to slow down the execution of the machine.

Please also refer to code comments for doubts.

PACKAGE REPOSITORY

Code will be provided contextually; in any case it can be found in this github repository:
github.com/gnoccoalpesto/navigationturt

PACKAGE INSTALLATION

Import this package in a workspace containing **turtlebot3** package.

Install **XTERM** terminal emulator to visualize nodes' dialogs windows.

Inside a terminal, run this command to create an ENV variable with turtle bot model:

```
export TURLTEBOT3_MODEL=burger
```

PACKAGE USAGE

Launch each task with this command, substituting K with the respective number {1,2,3,4} of the task:

```
roslaunch navigationturt taskK.launch
```

ROSLAUNCH ARGUMENTS

Here follows a list of arguments that can be modified in tasks' launchers.

TASK 1, 2, 3, 4

model: use doc to properly choose it, by default selected from ENV variable

x_pos, y_pos: spawning position's coordinates

TASK 2, 3, 4

gui: if true, opens Gazebo window; default false

open_rviz: if true, opens Rviz window; default true

move_forward_only: if true, robot can't go backward; default false

TASK 2

slam_methods: slam algorithm used; default gmapping

TASK 2, 4

teleoperation_mode: if true, manual control is activated; default false

TASK 3, 4

map_file: location on map.yaml file; default automatically found by name

TASK 3

objectves_file_location: location of .txt file containing navigation objectives; default automatically found by name

TASK 1 – SIMULATION SETUP

This is a prerequisite for all the following tasks and simply launches the Gazebo simulation (Gazebo window is open with arg gui:=true). Robot model and spawning position are default, while all the specific behaviours (exploration, navigation and cleaning) are shut down (arg exploration_mode:=false,uv_cleaning:=false, teleoperation_mode:=false, use_navigation_node:=false). No map, slam method, objective file are specified.

This is the only case where the main launcher doesn't request move_base server activation via the move_base.launch file: this file launches all the prerequisites of the Navigation Stack, which is used (via the NavigationNode class) in every other task.

RESULTS

A video showing the results is provided within the additional material.

TASK 2 – ENVIRONMENT EXPLORATION

The robot is asked to autonomously map an unknown environment (turtlebot3_big_house) starting from any point inside of it, by autonomously explore every room of it.
In this work, I used a frontier exploration algorithm with some heuristical parameters.

In this case, arg exploration_mode:=true and slam_methods (default :=gmapping) is passed to the launcher: this triggers the activation of exploration.launch, which is used to start slam routines and spawn exploration (explorer.py) and map saver (mapper.py) nodes.

The latter, consisting of the MapperNode class, simply saves the current map once receives a message(Bool, data=True), on the completed exploration topic (default: /exploration_completed).

The explorationNode class, manages the scouting algorithm. Here follows a brief showcase of the followed steps:

1) Global map information is acquired and used to initialize the local map (to be explored). This data will also be used to convert from global ("continuous") to local ("discretized") points and vice-versa.

2) A local instance of navigationNode class (from action_move_base.py) is created to manage the move_base client-server communication; contextually, the node starts the timer to call the exploration callback.

3.1) The callback updates robot's position, then executes a full rotation to look around.

3.2) New map data is gathered and frontier points are located by checking, for each map point, if a point is empty space (I.e. free from obstacles, represented on the map with 0 value), has some empty space around (by default, only one point in the 3by3 neighbour shall be free space) and has some un-explored points around (-1 value on the map; depending on the selected method, a 3by3 4- or 8-neighbour is scanned; by default, only one point in the neighbour shall be unknown, doubled if considering a 8-neighbour).

3.3) If no frontier is found, exploration timer stops and the node advertises the request to save the map; otherwise, the new frontier to be explored is chosen and the request sent to move_base via the navigationNode instance.

4) Frontier point to explore is empirically chosen by ordering the all frontiers by distance and discarding points too close to robot (two step procedure in the case all points are discarded the first time), then selecting one the closer to the robot the more of the map is explored. I started by picking a point more close than 66% of the other frontiers, since this avoided the robot getting stuck in the beginning and produced the shortest exploration times, then added some heuristic to avoid the robot exploring points too far as the exploration goes on, up to choosing a point closer than 80% of the other frontiers to the robot: this combination produced the best times and the least errors.

5) Computed goal is requested to move_base, specifying a timeout period. More about the navigationNode class behavior will be presented in the following chapter.

6) After move_base request is fulfilled or expires, the algorithm executes another time, starting from step 3).

RESULTS

The exploration can be completed in about 30 minutes, producing a map as much accurate as the ones obtained by manually teleoperating the turtlebot.

A video showing the results is provided within the additional material.

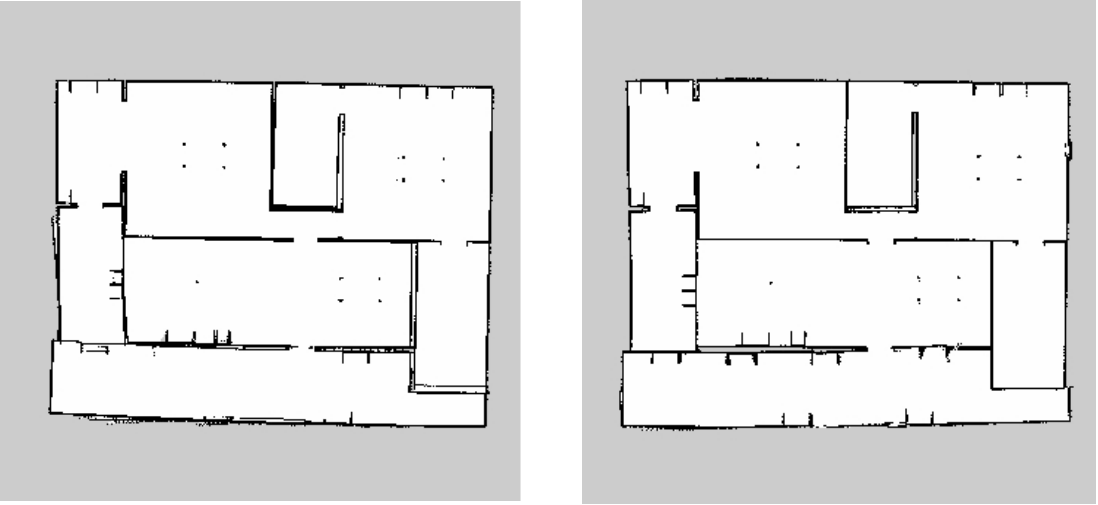

Figure 1: comparison between map autonomously (left) and manually (right) created

FURTHER IMPROVEMENTS

A more precise way to identify the point to explore could be using some particular point (e.g. the centroid) of a connected segment of the frontier: in this point the robot will have greater visibility and a larger part of the frontier could be explored at the same time.

Moreover, the point can the chosen utilizing other parameters besides the distance, like the extension of the connected frontier, further reducing the time spent to explore.

TASK 3 – GOAL TRAVERSING

The robot is asked to reach a sequence of points, contained in a file. Specifically, I used a .txt file, with a different point for each text row: for each one of it, the tuple (x, y, yaw) is specified. However a different file format could be used, e.g. .csv .
For what it concerns navigation algorithms, I preferred to use move_base framework due to the complexity of the environment (rooms, walls, other obstacles), since the proportional controllers I implemented did not rely on costmaps, making the navigation in such an environment impossible.y on costmaps, making the navigation in such an environment impossible.

For this task the arguments exploration_mode and teleoperation_mode are false, while use_navigation_node and use_objectives_file are both true (in this case objectives_file_location string is not empty): this will trigger the launch of navigation.launch, which purpose is to properly setup Navigation Stack (move_base server, amcl, map server, loading the map file from the passed location) and properly initialize the initial position of the robot using the initialPositionSetter node, contained in initial_position.py.

This node simply waits until move_base server is running, then publishes the data, read from the odometry topic, to the /initialpose topic, which will force the robot position on the map to be the same estimed by odometry. Once that is done, the node terminates itself.

Eventually, launcher spawns move_base_client node, contained in action_move_base.py, which manages every aspect of the actual navigation; while being the same used in the task before, still utilizing navigationNode class, but, being initialized with a list of objectives, behaves differently:

1) objectives are parsed by readObjectivesFromTextFile(location) function, contained in read_from_text.py, and passed to navigationNode's loadObjectives(list) method: this will prevent the node to answer to requests coming on the /Setpoint topic until all the objectives of the list are reached (of discharged, in case they can't). This also triggers a timer that serves the list's objectives one at a time.

2.1) If there still is a target to be served, it is splitted from the list and passed to the requestNewGoal(point) method, without any timeout.

2.2) Request feasibility is checked (by default only relatively to environment's limits) and, in the affermative case, the request is forwarded to move_base

2.3) Contextually, reference position is updated and motion monitor timer is activated: this will check if the robot is close to the target (proximity check; in some cases, this might interrupt the motion given the low request of precision of the tasks) or if it gets stuck (collision check: monitors distance travelled; this is relaxed if the robot is close to the target). Until it is shutdown (by timeout or accomplishment of the request), the monitor is able to interrupt the motion and cancel the request. Start over from 2.1).

3) Once all objectives are served, navigation node resumes its regular operation, accepting new targets by listening to /Setpoint and by direct method.

RESULTS

The use of Navigation Stack and move_base frameworks make the task particularly easy to accomplish: the percentage of unreachable objectives is close to zero, also given the high precision of provided map.

A video showing the results is provided within the additional material.

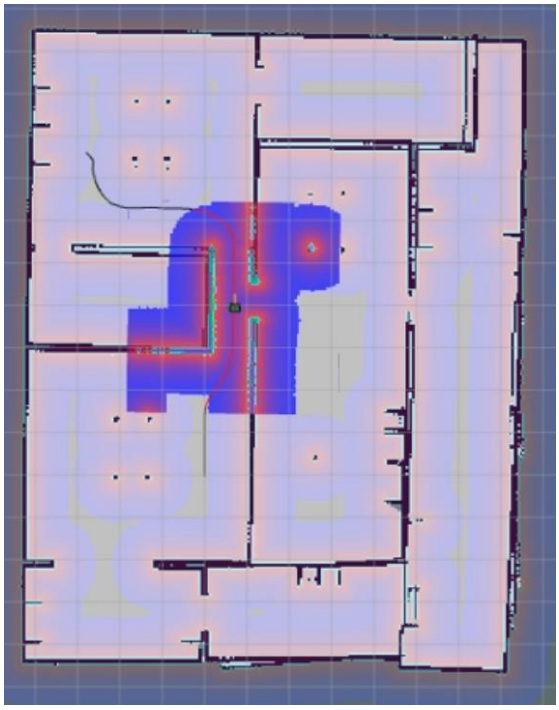

Figure 2: Rviz visualization of automous navigation, with cpmputed path and local costmap highlighted

FURTHER IMPROVEMENTS

Implement a feasibility check on the request by querying move_base plan service (feature currently lowly supported in python). This service also answers providing all the poses necessary for reaching the request's goal, giving a better way to estimate timeout period for reaching it.

Improve the motion monitor: a comparison between expected motion (reading /cmd_vel velocity command) and actual could be more adaptive than empirically estimating parameters like I did.

Moreover the recovery behavior simply interrupts the motion (method that proven efficient in this task), but controllers not based on costmaps could be effective in the case no obstacle is present on the path. At the moment, the tuning of the controller's parameters proven to be too much time consuming.

TASK 4 – ROOM CLEANING

The robot is asked to clean a selected room utilizing an UV map to kill a viral pathogen (in this case Covid19), given an emission power of the light and an absorption function for the surfaces.

For this task, the specific launcher activates the main one with argument uv_cleaning:=true: this triggers the launch of the same navigation.launch as the task before, but this time argument use_navigation_node:=false causes no navigation node to spawn; it will be the cleaningNode contained in uv_cleaner.py (spawned by navigation.lauch) to incorporate an instance of the navigationNode class to manage the requests to move_base. For this particular task, some coefficients (relatively to navigationNode's motion monitor) have been tuned to improve the results.

Its main purpose it to identify which points of the map are yet to be completely cleaned and identify which one to navigate to, utilizing an the distance (the closer, the better) and the average energy quantity already absorbed by the neighborhood of the point: this, together with empirically identified coefficients, gave the best results, with respect to time to accomplish the task and times the robot got stuck.

For this simplified case, the room to be cleaned is defined in terms of position (x, y) of its corners; the selected resolution for the room is, by default, 0.1 m (mostly for computational reason), but I tested it down to 0.06m with good results. It must be noted that, in case of flawed overlapping of the environment and cleaning map (this last one is published by the node and represents the amount of energy already absorbed by the points), tiles outside the selected room could be selected.

Here follows the steps of the selected algorithm:

1) Room and navigationNode instance are initialized. Environmental map data is gathered and used to initialize the uv map: all the points outside the selected room will be ignored (for simplicity setting them at required energy level). Then, the laser data (which simulate the uv emission from the lamp), is acquired and the operation timer is started, together with the timer publishing the uv emission values: this two timers work in parallel, allowing both motion and uv light simulation to work simultaneously.

2) Each time the cleaningTimer object fires, uv intensities are computed, once for each point, based on distance, up to the maximum range of laser scan data, while also considering the blocking effect of obstacles: discretizing the angle and the distance enables the program to understand if some points lay in the shadow of an obstacle, hence are not receiving any light; the points' position is approximated to match a value of discretized angle and distance, with the latter being compared with the maximum distance reached by laser scan ray, for that particular angle to understand if the line of sight is blocked. Then, the updated map, already transformed in map coordinates, is published for visualization.

3.1) The points that still require energy are filtered, removing the ones move_base already tried to reach without success. In the case of an empty list result, the filtering action is reversed.

3.2) For each permitted point, program crates an array containing values of distance from robot (filtered to exclude points too close, causing problems to move_base), local and average (on a given size neighborhood) values of received energy, that will be used to compute the score.

3.3) The score consists in "normalized" (using the average) and discounted (using a specific empiric coefficient. By default local energies are ignored; distance discount increases the more unreachable points move_base discovers) values of the array, added up together with a (slightly significant) random value for each previous point.

4) The point minimizing the score is selected as next objective and forwarded to move_base client node, which will also monitor robot motion and position as in the previous task. In case of aborted motion, point is added to the unreachable ones list and a recovery behaviour (based on proportianal controllers I designed) might be activated. Otherwise, robot stops few seconds at goal position to clean the surrounding area, then checks if room is completely cleaned (computing if the percentage of cleaned points is above a certain threshold). If not, algorithm goes back to step 2).

RESULTS

The chosen empirical parameter and methods enabled the robot to clean the room faster with respect to other tried methods. Overall, the time spent to clean is adequate.

A video showing the results is provided within the additional material.

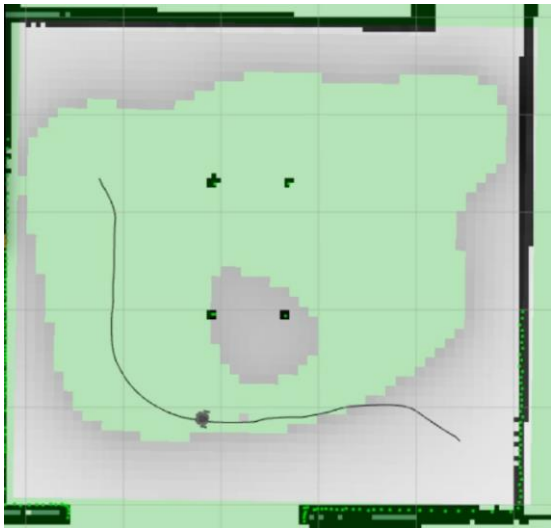

Figure 3: visualization of UV map and computed path during room cleaning

FURTHER IMPROVEMENTS

A major improvement, as said before, could be the use of ad hoc controllers to navigate the room, even more so since the robot only moves inside a single room (adequate remedial measures shall be utilized due to possible obstacles inside the room, or, even, the use of laser scan topic).

For what it concerns the logic behind the selection of goal point, it might be introduced, like in task 2, the possibility to avoid points too close to an obstacle, in such a way that limits the shadow effect created by it.