ARTIFICIAL INTELLIGENCE AND ROBOTICS

LM IN ARTICIAL INTELLIGENCE

The Final Challenge

SUBMISSION 1

Students :
Álvaro ESTEBAN MUÑOZ

Teacher :
Alessandro SAFFIOTTI
Andrea GOVONI

July 2, 2024

Contents

1 Executive summary

The following report describes how it was addressed "The Final Challenge" assignment. The tasks covered in this report are, in addition to the main one, optional tasks #1 and #2. In order to perform a good evaluation of the system, the robot was tested in many different configurations from start to end, this means, changing the starting point and world configurations like the doors and boxes state. It is important to notice that, testing cannot prove the absence of bugs, therefore even if the system passes all the tests it does not imply it is completely robust.

1.1 Limitations

Dead ends The system is not able to avoid death ends, this is a limitation mainly imposed by the design chosen for the navigation behavior. The behaviors related to reach a target and to avoid obstacles are merged on the same one, hence when reaching a dead end the `GoTo` and `Avoid` behavior rules enter in conflict, removing the possibility for the robot to escape.

Sensor box detection Before picking up a box, the system tries to detect it. This technique is simply performed by evaluating the value of the three sonar sensors located on the front part of the robot (330º, 0º and 30º, therefore leading to imprecisions when other obstacles are around or when the box is out of range of those sensors.

Door detection When detecting the door in order to cross it may happen that the plan fails due to the front sonar detecting the angle of the door's frame or an obstacle at the other side of the door instead of the hole. This does not suppose a huge problem since the system will replan and try to cross again, hence just leading to perform some unnecessary computations, but still executing the plan.

Failure detection and recovery The system uses an approach for "Failure detection and recovery", which means it replans from the current state after failing an action. This approach has an intrinsic problem, and it is the necessity to fail in order to replan, which means the robot will never replan before failing, even if the conditions for the failure are satisfied before executing the action. This can be seen in deep on the optional tasks #1 and #2.

1.2 Improvements

Some or most of the limitations exposed above could be solved applying the following improvements.

Map-guided navigation As shown in laboratory 2, a fuzzy map could be constructed from sonar data and then used to guide navigation. This would possibly solve all the limitations listed above if correctly implemented.

Continuous box detection If the positions of boxes is continuously monitored, we can fail the action in charge on navigating to the target and replaning in advance, leading to a more efficient navigation.

Door state update We could use a different strategy to detect doors' state and contunously monitor doors state, in this way we update the state of doors instead of sending the `Cross` action to fail.

2 The main task

2.1 Challenges

For the correct completion of this task we need to cleverly merge the plans generated by our implemented planner in assignment 4 with the behaviours from assignment 3. The task may seem trivial but it actually involves the correct implementation of an architecture able of creating a plan from a state built from robot sensors and designing accurate behaviors matching the actions defined by the planer with curated rules for each of them, a cognitive architecture.

2.2 Solution

The implemented *cognitive architecture* is a basic **2-layer hybrid architecture**. We merge a given dummy version of the **SPA** (Sense-Plan-Act) loop with the already designed **RCP** (Robot Control Program) taken from the previous assignments. In this way, the SPA will take the information acquired by the RCP using the sensors, generate a plan and send it back to the RCP which will execute it by means of fuzzy rules.

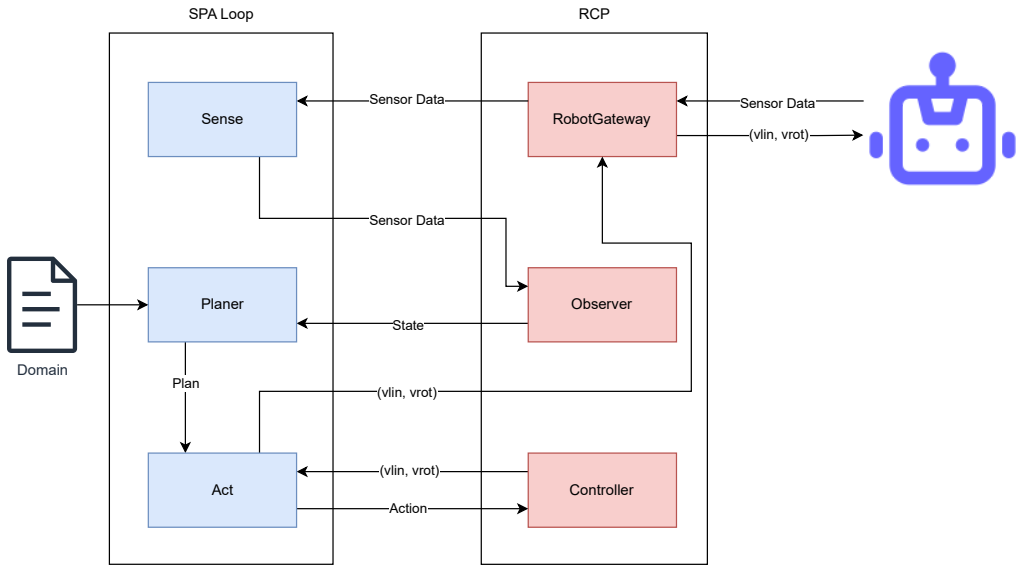

Figure 1: Flow of information between the layers of the cognitive architecture

Take a look at the architecture diagram on figure [1]. The information flow is controlled by the SPA loop, which uses the components of the RCP to acquire information, create the state and compute the `vlin` and `vrot` parameters. There are two main loops to identify here, the top one, **the SPA loop**, takes the information, generates a plan, and executes the second or inner one. **The inner loop** computes the parameters for each time cycle until the action on execution is completed or failed, this loop is executed for each action inside the generated plan.

One important challenge here is how to merge the previously implemented `GoTo` and `Avoid` behaviors, there are two main ways of addressing this, either implementing a behavior compressing the rules of both, or connecting them using a Macro-behavior. Both solutions have strong and weak points. On the one hand, merging the rules of both in a unique behavior leads to more robustness in achieving the main goal, while increasing the complexity of the overall behavior, this means the rules will have to be more elaborated since they can lead to conflicts, a balancing of the priority on the rules needs to be established. On the other hand, connecting both behaviors, while making the design more modular an simple to implement, may lead to not finding the solution in some cases since the achievement of the partial goal does not mean the final objective is granted, this is, once avoiding an obstacle, the robot may run into the same obstacle once more.

The chosen approach for this task was the first one, this way we guarantee our robot will be always looking to reach the main goal even if it leads to a poorer robustness on the avoid behavior, which we can afford since we are running a simulation. To sum up, we have two main behaviors.

- **GoTo**: Behavior in charge of guiding the robot to the target and avoid obstacles on the way. It computes the distance to the target using the estimated position of the robot and the target coordinates given in the robot map. The obstacles are detected using the sonars present on the base of the robot, namely those ones on positions 30º, 60º, 330º and 300º. Moreover, the front sonar (position 0º) is used to detect when the robot is too close to crash and moves backwards. As stated previously, the main problem with this implementation is that some of the rules may enter in conflict, e.g. if the target is right behind an obstacle, the robot might try to apply the `AvoidLeft` rule, entering in conflict with the rule `ToRight` which insist in guiding the robot towards the target.
- **Cross**: This behavior activates when a door needs to be crossed, hence assuming the initial point for this behaviour will be in front of the door. Sensors on first and fourth quadrant (see figure 2) are used to compute a calibration parameter which is just the difference between the summatory of sensors on each quadrant. This leads to negative calibration when the robot it is slightly uncalibrated to the left side (cause total value of sensors on fourth quadrant is higher and therfore the distance to the wall is higher) and positive when it is uncalibrated to the right. To detect when the door is crossed, the minimum value between sensors on positions 150º and 210º is taken. The function used to measure when the door is crossed is shown on figure 3. The implemented function establishes the satisfac-

tory values inside a range, this means values too close and too far to the door are discarded, setting the range $[0.5, 1.5]$ as the ideal distance from the back of the robot to the door to be considered crossed.

It is important to notice that the robot keeps a map of every object's symbolic position and their coordinates, as well as the area of every room. There are also some functions implemented to compute in what room or symbolic location there is an object depending on its coordinates. Here we added also a field to determine if the robot is holding or not something, this will be useful to keep track of the situation related to the box when transporting it and recover from failure.

2.3 Testing

To test our implementation we have selected five different starting points (figure [4]) and assigned a different task for each of them and different door configurations. The executions are listed below:

Configuration #1 This is the most basic configuration. We set all doors to open and use the default starting point $P_1 = (3.0, -2.0)$. The task executed is `transport(box3, bed1)`. The execution of the task and its real trajectory can be seen on figures [5] and [6].

Configuration #2 With this goal we want to test the performance when opening and closing doors, hence we simplify the task, while closing the doors on the way to add more difficulty. The starting point is now set on Room2 $P2 = (-6.0, -8.0)$, and doors D3 and D4 are set to `'closed'`. The new task to be executed is `navigate_to(wardrobe1)` (Figures [8] and [9]).

Configuration #3 This goal allow us to test the ability of the system to avoid obstacles on the way. Door2 is set to closed and the robot will try to execute the task `transport(box3, wardrobe1)`. The starting point is set to $P3 = (4.0, -7.0)$, meaning the robot will have to avoid the table3 on the way to pick up box3. Results can be seen on figures [11] and [12]

Configuration #4 This configuration aims to test the overall robustness of the system by merging everything from the previous configurations. doors D2 and D4 are set to 'Closed' while D3 is left open, hence the robot will cross both, open and closed doors. The starting point on position $P4 = (7.0, -1.0)$, and the task is set to `transport(box1, stove1)` allowing us to also test once more the ability of the system to avoid obstacles (table1 is on the way in this case). Once more, the path and trajectories followed by the robot can be seen on figures [14] and 15.

Configuration #5 The last configuration's scope is to test the robustness of the system to understand its position inside doors. D1 is set to closed while starting point is set

inside D4, namely on $P5 = (-3.0, -5.0)$, which remains open as the rest of the doors. The task will simply be `navigate_to(box2)`. See figures [17] and [18]

For every configuration we measured the average execution time of each action as well as the number of times each one was completed or failed. This data can be seen on table [1]. Notice execution time is the average of all the configurations.

	ex_time	**n_complete**	**n_fail**
GoTo	10.77	16	0
Cross	7.03	9	4
Open	0.10	10	0
Close	0.10	7	0
PickUp	0.10	3	0
PutDown	0.10	3	0

Table 1: Results on behavior analysis for the main task

2.4 Assessment

From the results obtained we need to underscore a few important things. First of all, the execution time of the GoTo action gives us a view of how long does it take the robot to reach its objectives, however this may vary depending on the defined goal. The average time of the Cross action also gives us a good overview of how fast the robot is able to cross doors, we see it is close to the time of the GoTo, meaning the robot might be loosing a lot of time just crossing doors. Doors are critical sections, hence it is fair to take time to cross them, however, this action could be improved designing a set of better rules and adjusting the parameters according to those new rules.

From the number of completed and failed actions we understand that our robot is quite robust. Notice the Cross action has failed 4 times to complete, this is due to the front sonar, which is used to detect whether the door is open or not. Sometimes the robot understands a part of door's frame as if the door itself was closed, sending the Cross action to fail and replan, this does not lead to further problems since the robot just retries to cross the door. A better detection system for the door would easily solve this issue.

In addition, the minimum distance from the robot to obstacles was monitored during the execution of each configuration, using the sonar sensors around the base of Tiago. We can observe the robot rarely approaches more than 0.25 meters to an obstacle, except on doors. Minimal peaks usually match with crossing of doors (notice on figure [16]).

3 Optional task #1

3.1 Challenges

On the previous section we mention that the inner action loop executes until the action is completed or fails. On this task we are required to implement a recovery strategy to avoid failing the plan because of one action, namely in the case the box we want to transport unexpectedly changes location.

3.2 Solution

To solve this task we focused on two important points; i) fail detection and ii) recovery. The solution was constructed by isolating both tasks, easily achievable by relying on the given service to pick up the box which returns `False` if the box cannot be collected. In this way we could work comfortably on implementing the recovery system and once done, we focused on the failure detection. On the one hand the recovery strategy is a simple replanning after failure, meaning after the failure is detected we call the planer to generate a new list of actions to be executed from the current state. On the other hand, fail detection is simply an evaluation of the front sonar data, i.e. if the sonar gives a value above a threshold then the box is not in front of us and sends the action to fail. The selected threshold for the box is of 0.6 meters.

3.3 Testing

The testing performed is very similar to the one of the previous task. To simplify, we always keep the door opens, we only want to test the ability of the robot to detect failure and recover due to the unexpected change of position on boxes. It is not worth saying that we use only those goals from the main task [2] which include the `PickUp` action, hence those whose upper goal is `transport(box, target)`, namely goals #1, #3 and #4 (see configurations 2.3).

Trajectories followed by the robot can be checked on figures [21], [22], [23]. We can notice how the trajectory changes a lot since the robot tries to pick up the box on different locations.

Results of the performed experiments can be seen on table 2

	ex_time	**n_complete**	**n_fail**
GoTo	11.15	15	0
Cross	9.42	6	0
Open	Nan	0	0
Close	Nan	0	0
PickUp	0.10	3	50
PutDown	0.10	3	0

Table 2: Results on behavior analysis for optional task #1

3.4 Assesment

From the results shown on 2, we can underscore a few things:

Successful completion of tasks We can say the robot completes every task even when the box unexpectedly changes position, this exhibits the ability of the robot to replan after failure. Furthermore, this can be seen also as there is not a single `GoTo` action failing. The performed replanning can be seen on figure [20].

Imprecisions in Failure Detection The frequency of `PickUp` action failures is notable. The detection mechanism may fail when the box is slightly outside the sonar's field of view. Therefore, the action is attempted multiple times because the robot is at the target box's location, leading to repeated failures due to the slight misalignment. We can see most of this failed attempts come from the test performed on configuration #4 (results can be seen on listing [1]). These imprecisions could be overcome by building a map using sonar data, in this way we would be able to differ walls and obstacles from the box itself.

Doors assumption Notice execution time of `Open` and `Close` actions was not measured due to our assumption of always keeping doors open to simplify this task.

Listing 1: Generated JSON file with the results of the testing for configuration #4


```
{
    "GoTo": {
        "ex_time": 9.440269061497279,
        "n_fail": 0,
        "n_complete": 7},
    "Cross": {
        "ex_time": 9.417209327220917,
        "n_fail": 0,
        "n_complete": 4},
    "Open": {
        "ex_time": NaN,
        "n_fail": 0,
        "n_complete": 0},
    "Close": {
        "ex_time": NaN,
        "n_fail": 0,
        "n_complete": 0},
    "PickUp": {
        "ex_time": 0.10040600932374293,
        "n_fail": 48,
        "n_complete": 1},
    "PutDown": {
        "ex_time": 0.10277366638183594,
```

```
        "n_fail": 0,
        "n_complete": 1}
}
```

4 Optional task #2

4.1 Challenges

In this second optional task we are required to extend our system to deal with the cases in which the robot map is wrong and the doors set to be open are actually closed. Hence, a strategy needs to be implemented to detect those situations. After the implementation of the previous task [3], this one becomes much simple since we will be re-using the recovery strategy and we only need to implement the strategy for failure detection.

4.2 Solution

The approach implemented for failure detection is very similar to the one used for the box. In this case is much robust since the robot activates the `Cross` behavior right in front of the door, which is also a plane and therefore it is easier to detect than the box. In fact, for this approach we just used the sonar at position 0^o, its value is thresholded on 0.5 to avoid detection of obstacles in the other side of the room as if they were the door.

4.3 Testing

To test our new failure detection strategy we used the same configurations as the ones defined in the main task [2]. Since we are dealing with detection of closed doors we changed configuration #1 [2.3] and set door D2 to closed. The rest of the configurations remain unchanged.

Results after testing can be seen on table 3.

	ex_time	**n_complete**	**n_fail**
GoTo	11.71	16	0
Cross	3.18	9	19
Open	0.10	19	0
Close	0.10	8	0
PickUp	0.10	3	0
PutDown	0.10	3	0

Table 3: Results on behavior analysis for optional task #2

4.4 Assesment

The experiments conducted demonstrate the behavior of the newly implemented failure detection strategy. Ideally, the robot should fail once for every completed cross action (e.g., 9 completions should result in 9 failures). However, the results indicate that the robot is failing slightly more often than expected. We analyzed this by examining the execution logs and found that in configuration #1, there is a higher-than-usual number of failures in the `Cross` action [2].

To investigate further, we combined the trajectory plot with the minSonar plot to correlate the minSonar values with the robot's position. As shown in Figure [24], the sonar value drops below the threshold near the door at position D2=(1.0, 2.6), even when the door is open. This occurs because the robot mistakenly detects the door frame as if the door were still closed, leading to repeated failures of the `Cross` action. Although this issue only results in replanning and multiple attempts until the robot correctly detects the open door, it could become problematic if the system scales and replanning becomes costly. For now, this issue remains manageable.

Listing 2: Generated JSON file with the results of the testing for configuration #1 on optional task #2


```
{
    "GoTo": {
        "ex_time": 9.418585062026978,
        "n_fail": 0,
        "n_complete": 3},
    "Cross": {
        "ex_time": 1.3481365740299225,
        "n_fail": 7,
        "n_complete": 1},
    "Open": {
        "ex_time": 0.10211031777518136,
        "n_fail": 0,
        "n_complete": 7},
    "Close": {
        "ex_time": 0.10168957710266113,
        "n_fail": 0,
        "n_complete": 1},
    "PickUp": {
        "ex_time": 0.10410809516906738,
        "n_fail": 0,
        "n_complete": 1},
    "PutDown": {
        "ex_time": 0.10487866401672363,
        "n_fail": 0,
        "n_complete": 1}
}
```

A Figures

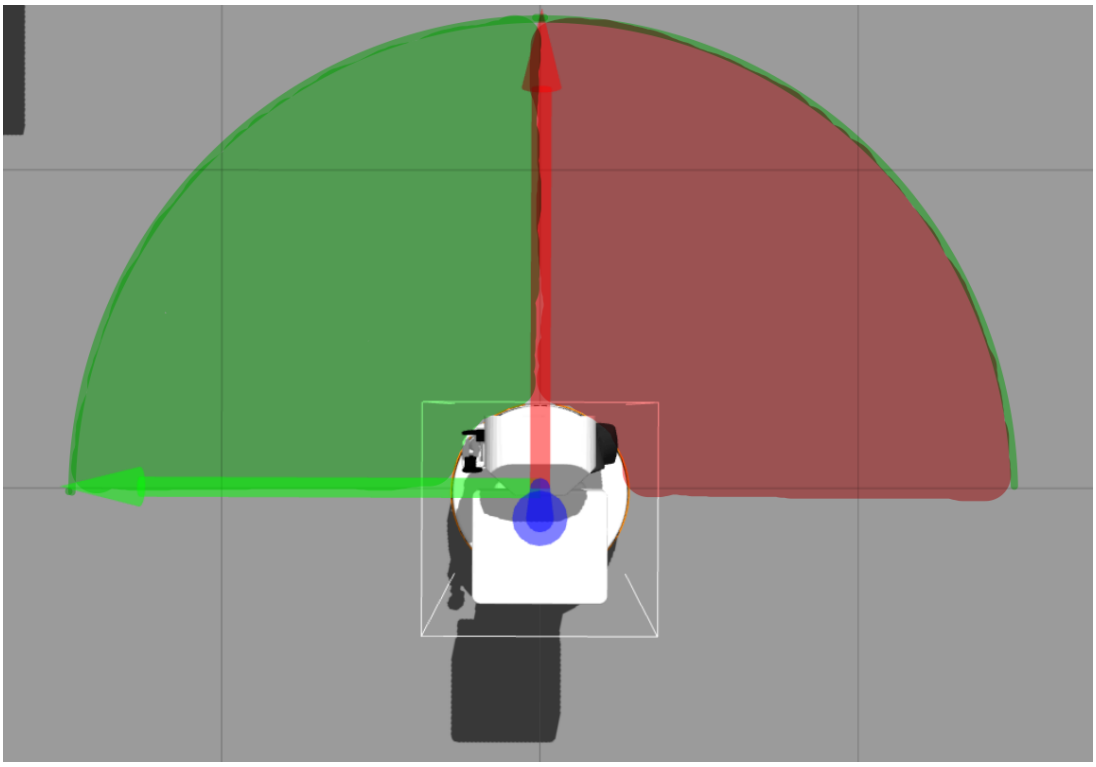

Figure 2: Division of sensors for computation of the calibration parameter

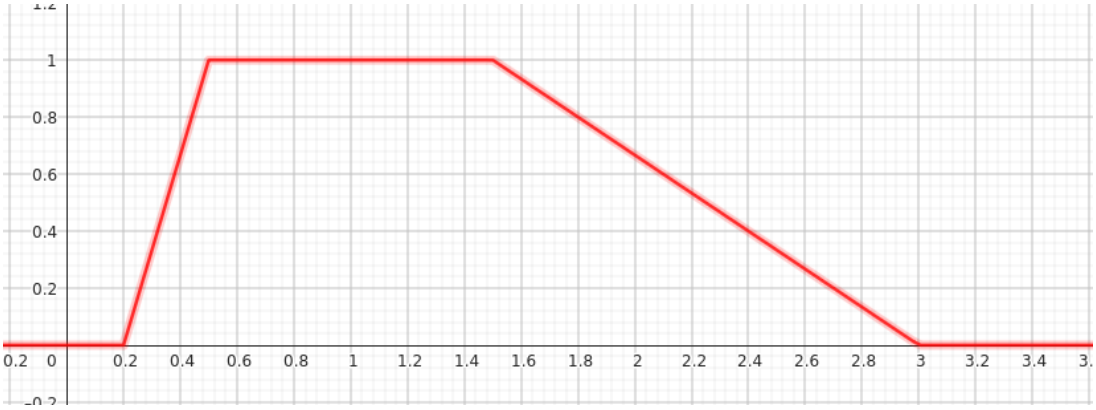

Figure 3: Doordist function to determine when the door is considered "crossed"

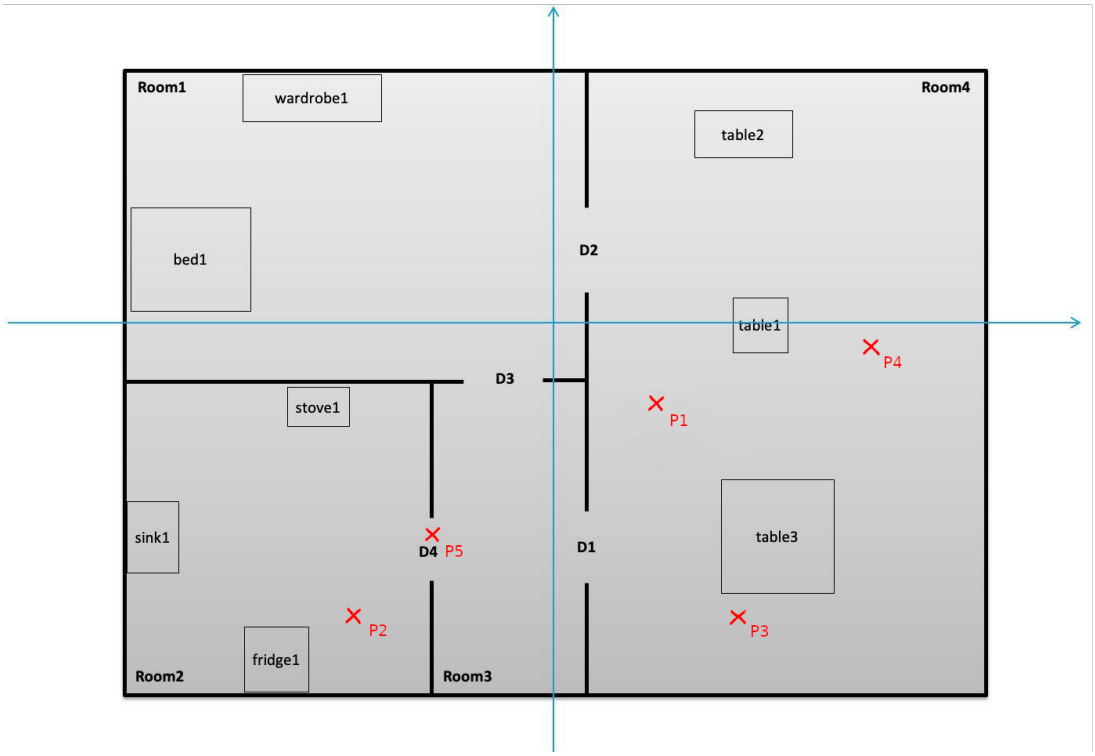

Figure 4: Selected starting points for the testing

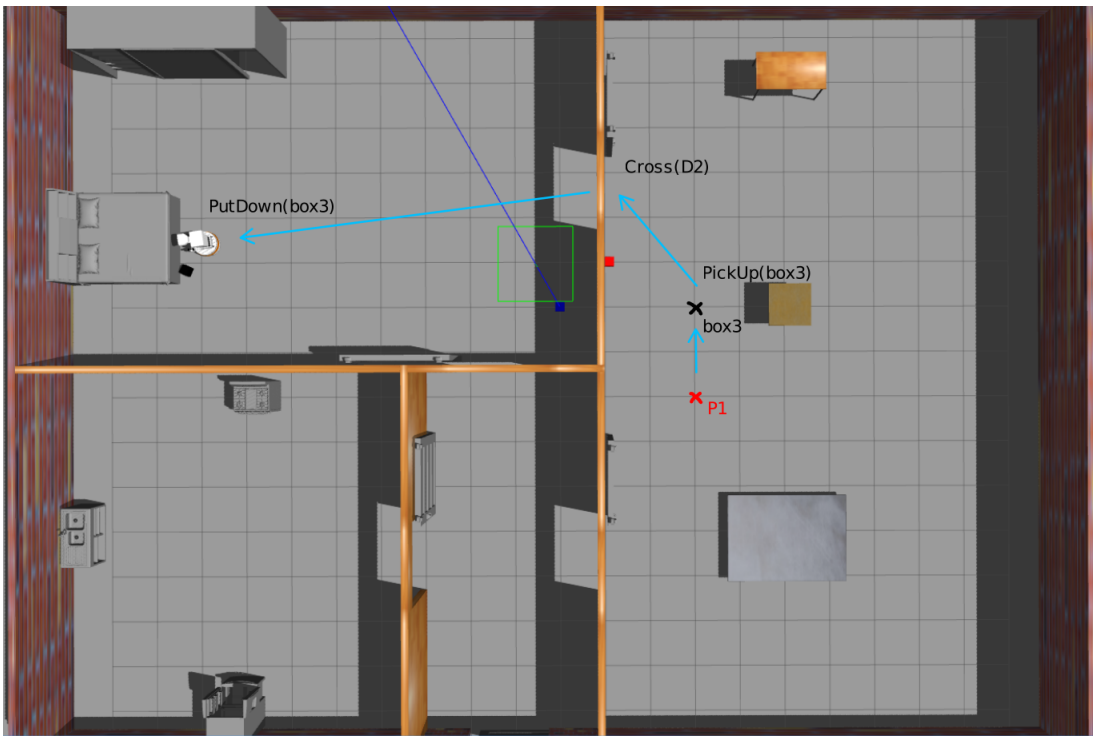

Figure 5: Execution of goal #1

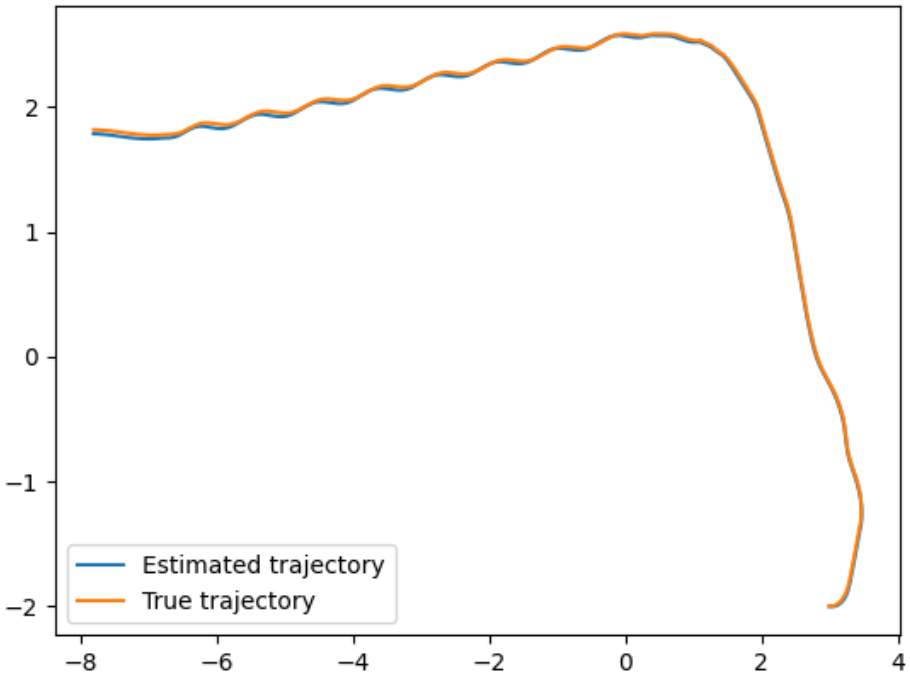

Figure 6: Estimated vs. Real trajectory performed by the robot on goal 1

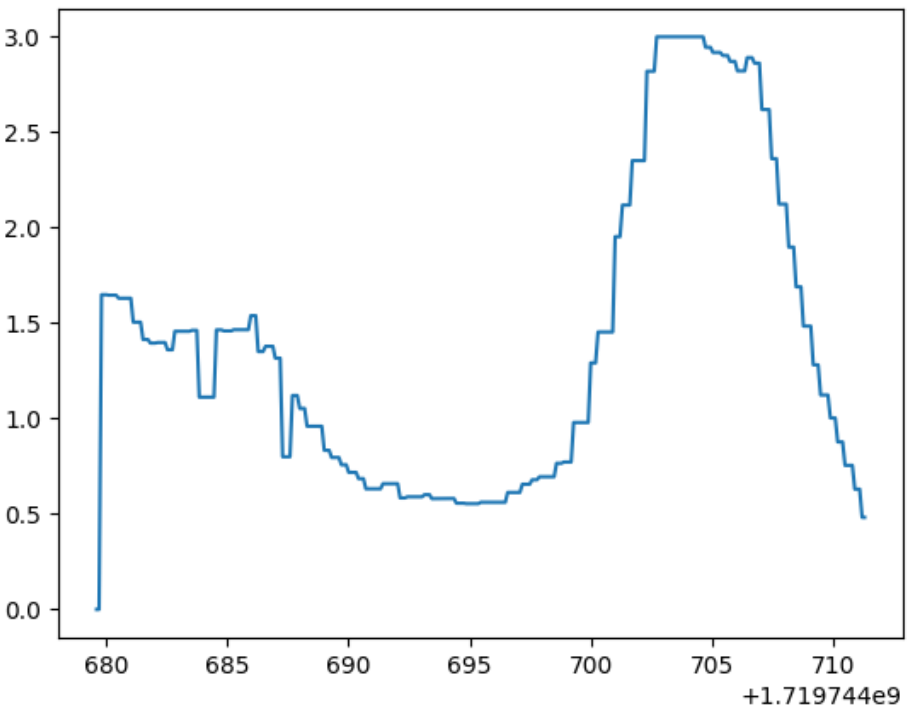

Figure 7: Minimum value of sonar range during the execution of configuration #1

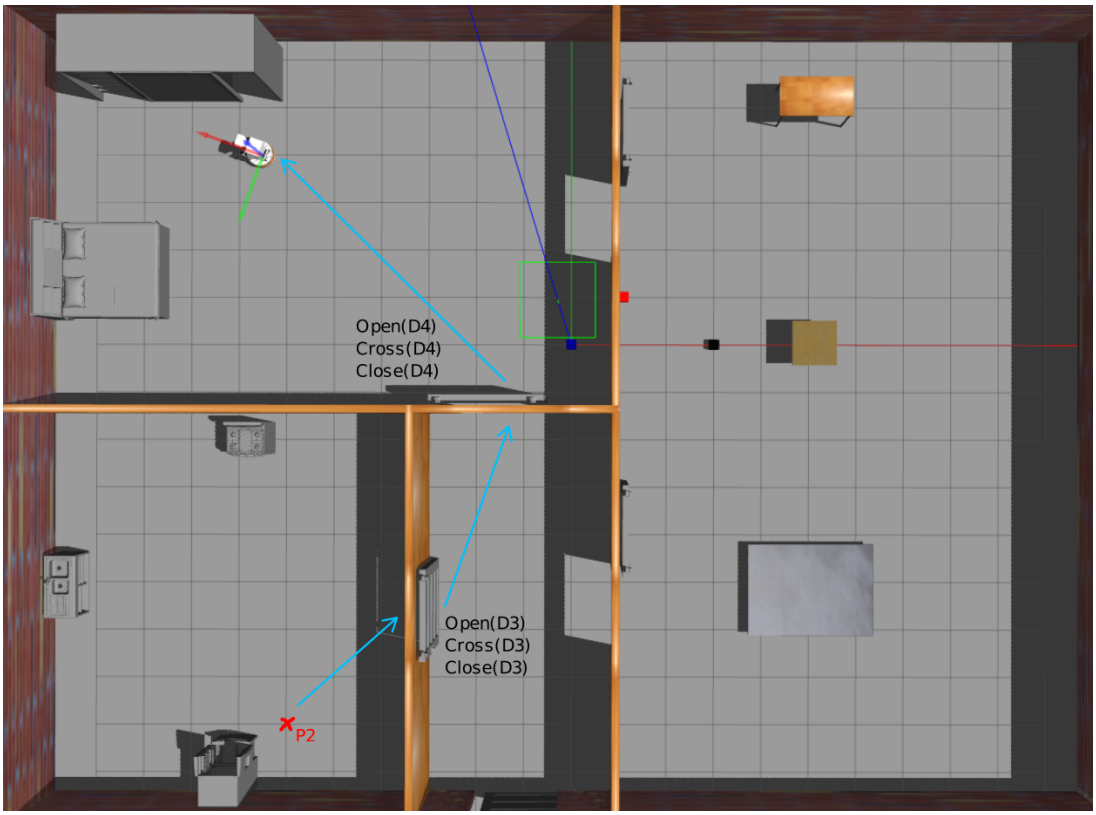

Figure 8: Execution of goal #2

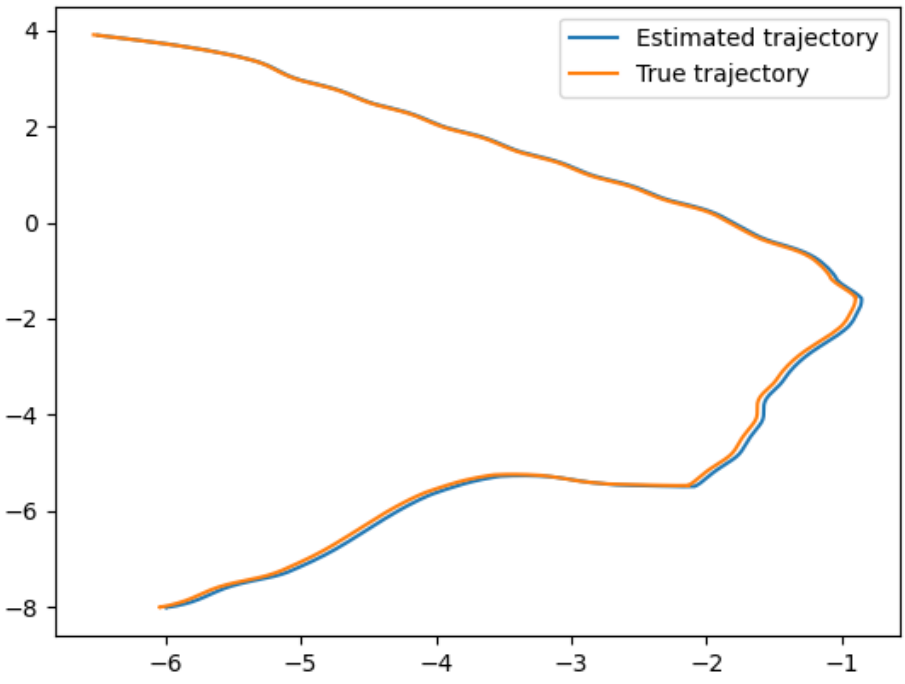

Figure 9: Estimated vs. Real trajectory performed by the robot on goal 2

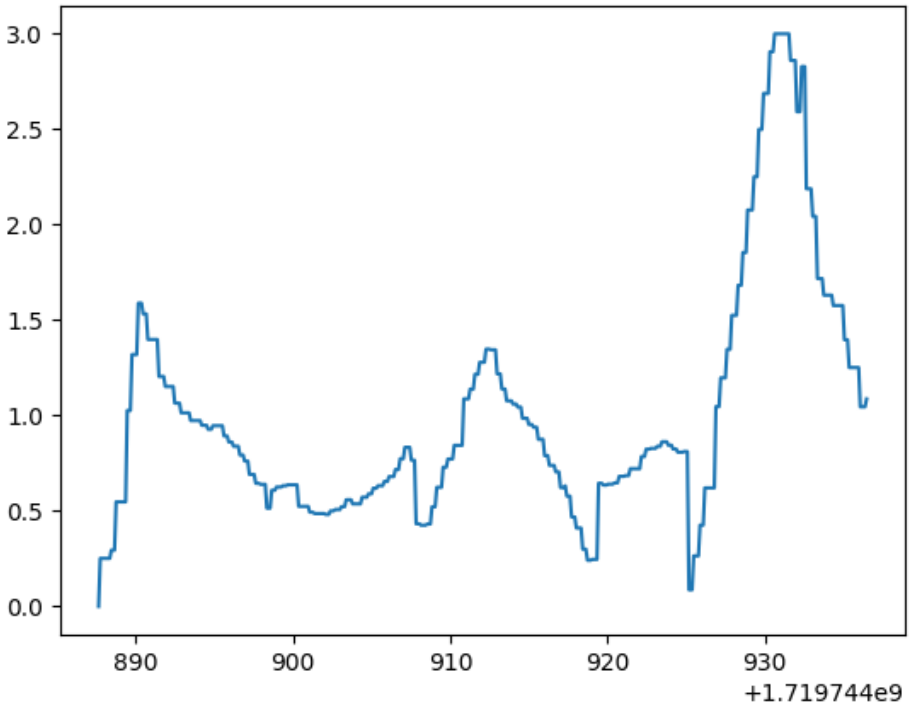

Figure 10: Minimum value of sonar range during the execution of configuration #2

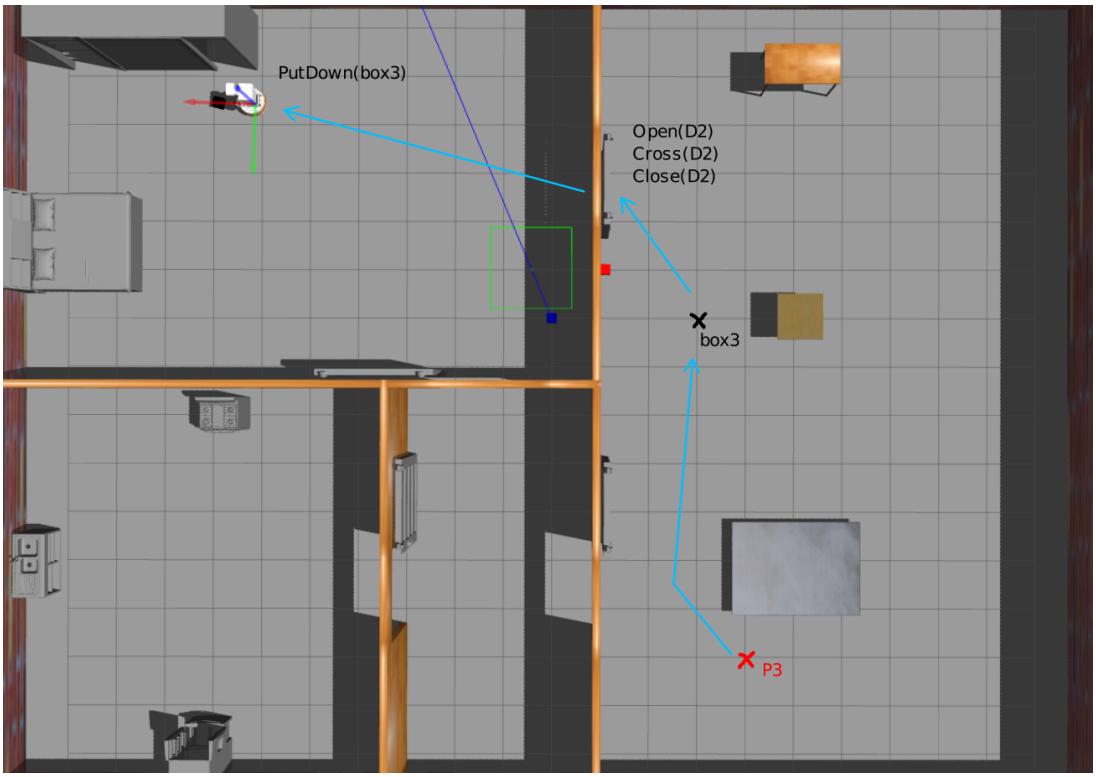

Figure 11: Execution of goal #3

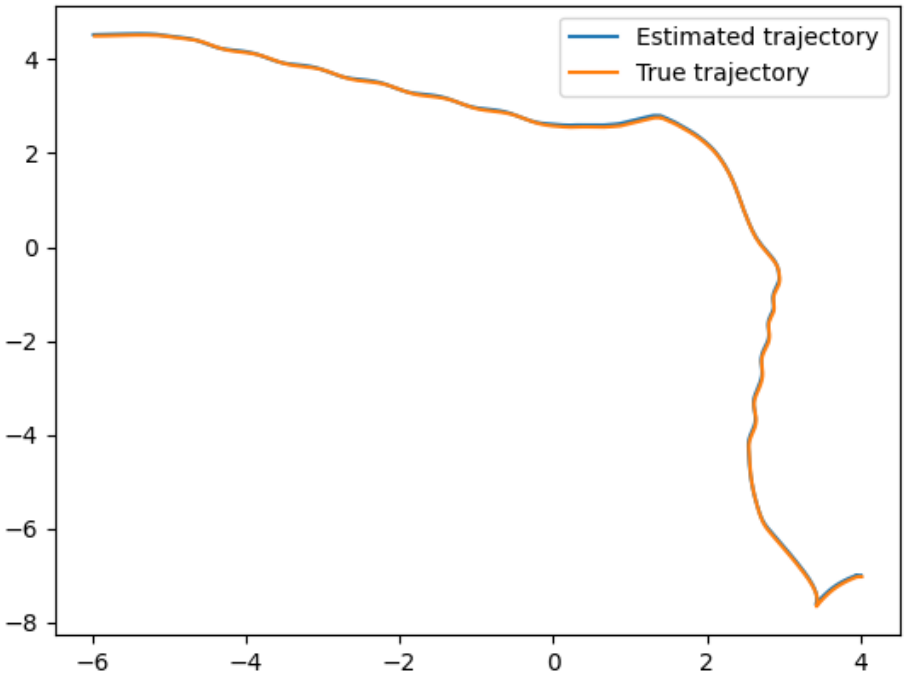

Figure 12: Estimated vs. Real trajectory performed by the robot on goal 3

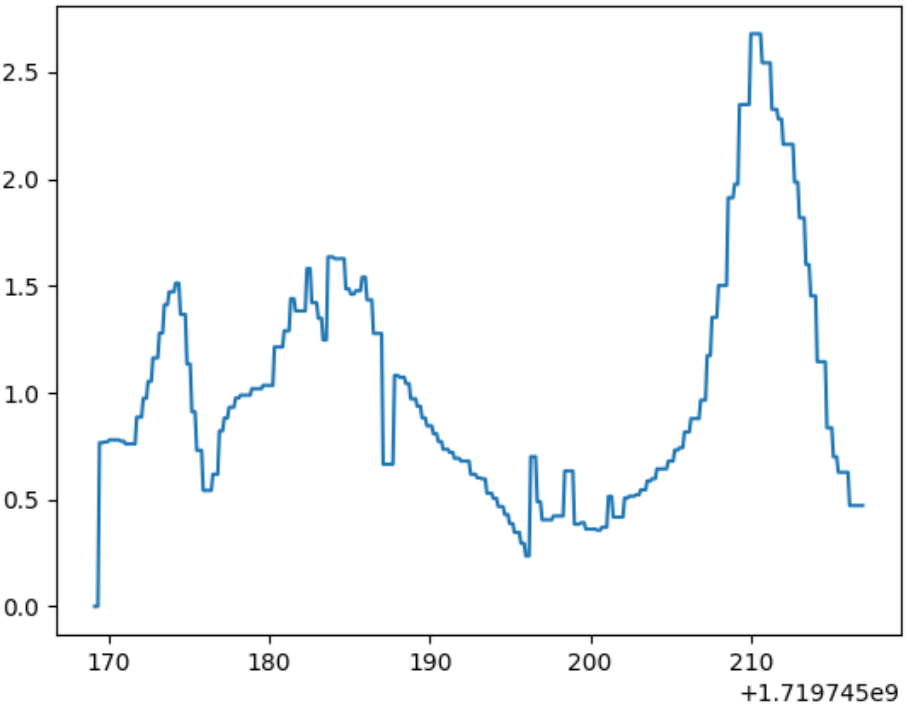

Figure 13: Minimum value of sonar range during the execution of configuration #3

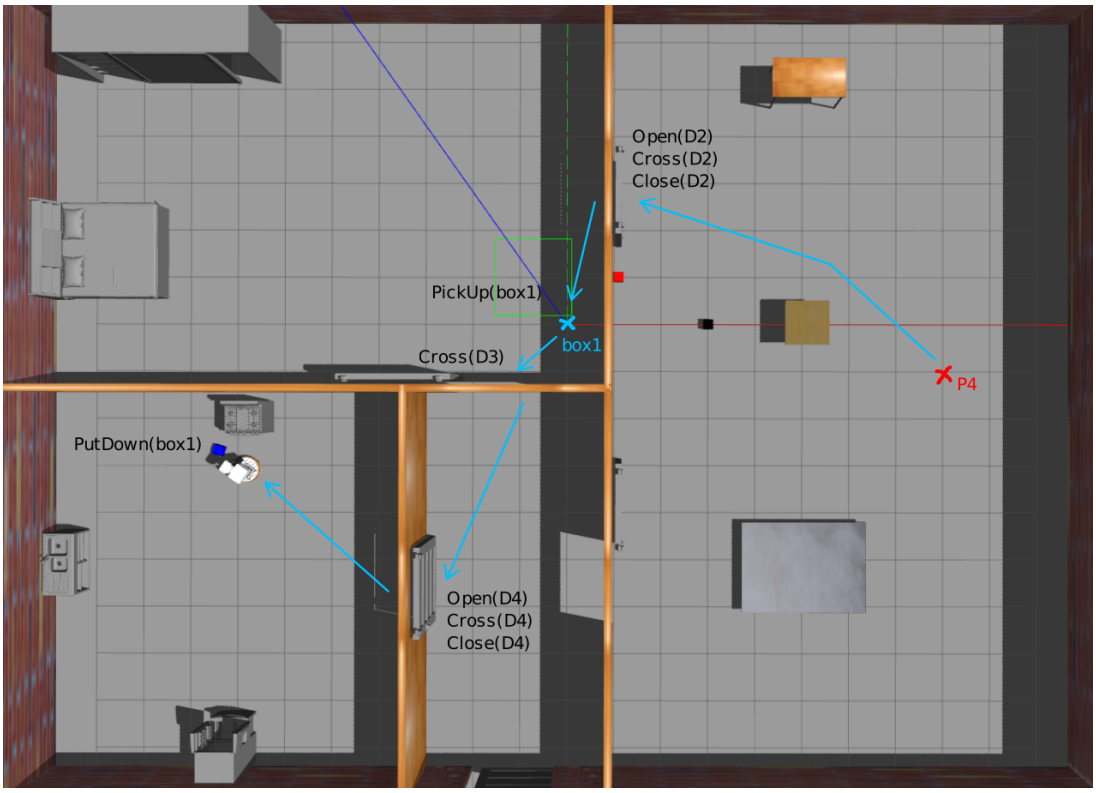

Figure 14: Execution of goal #4

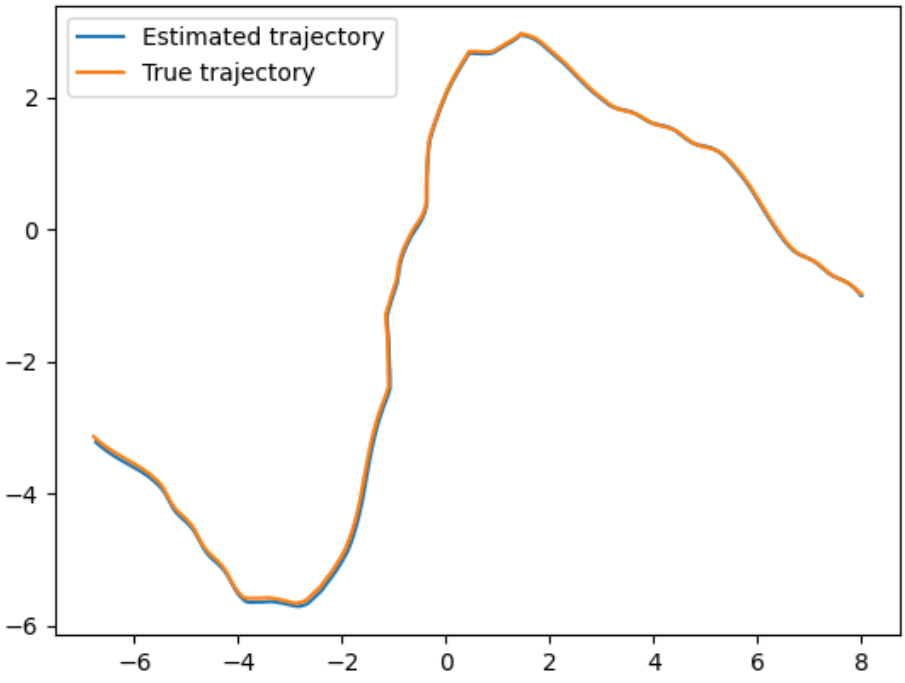

Figure 15: Estimated vs. Real trajectory performed by the robot on goal 4

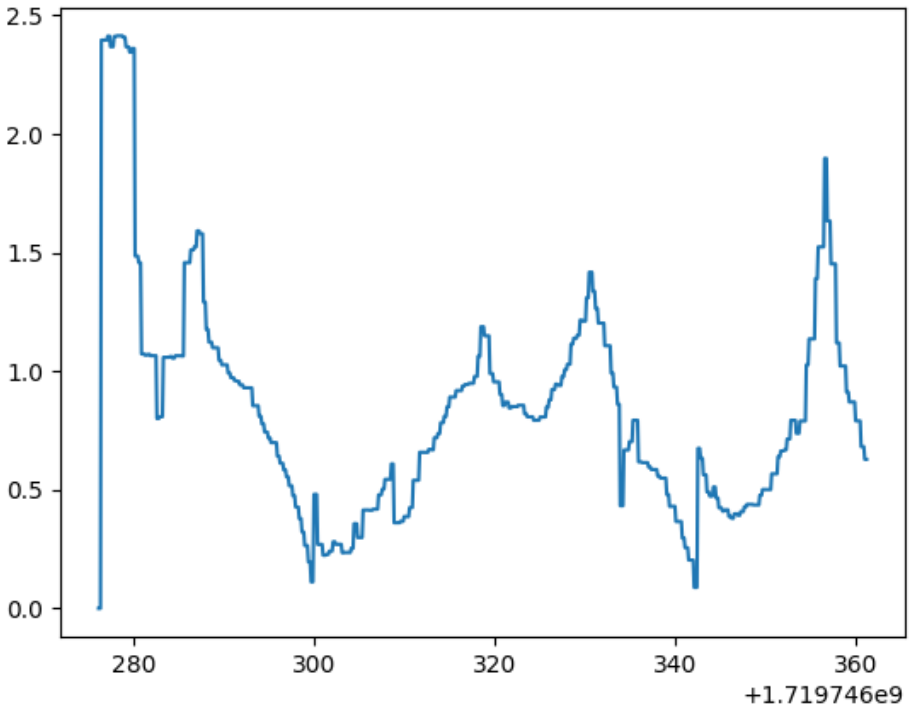

Figure 16: Minimum value of sonar range during the execution of configuration #4

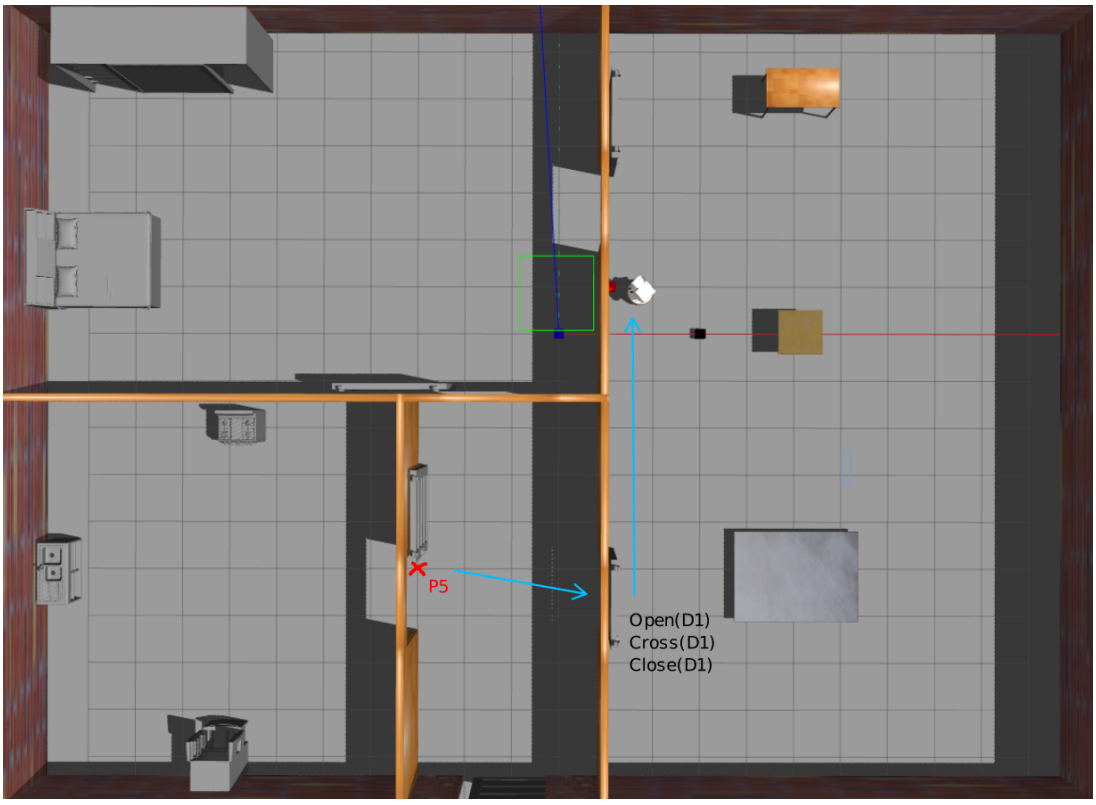

Figure 17: Execution of goal #5

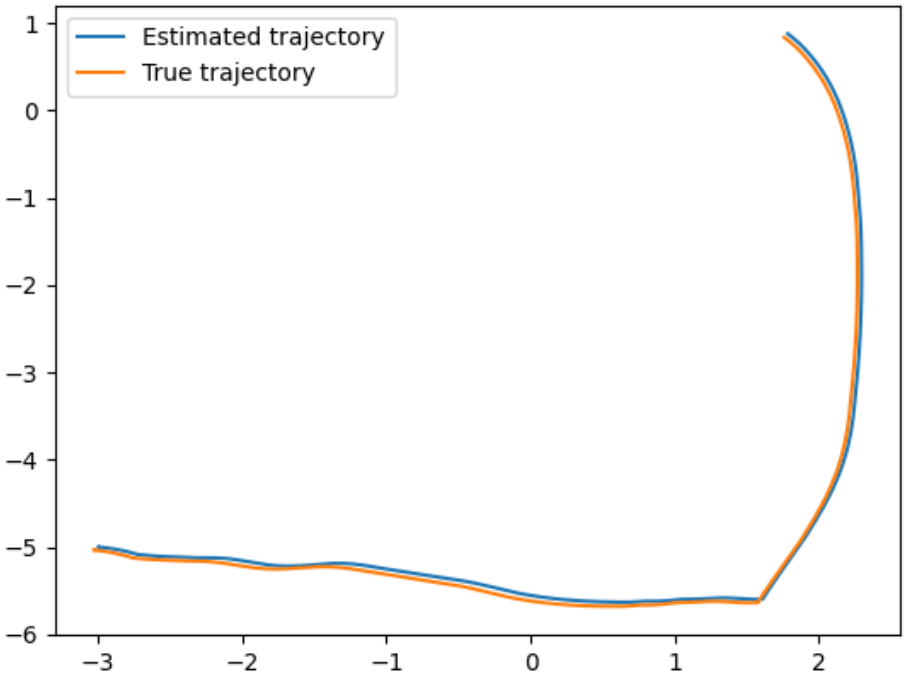

Figure 18: Estimated vs. Real trajectory performed by the robot on goal 5

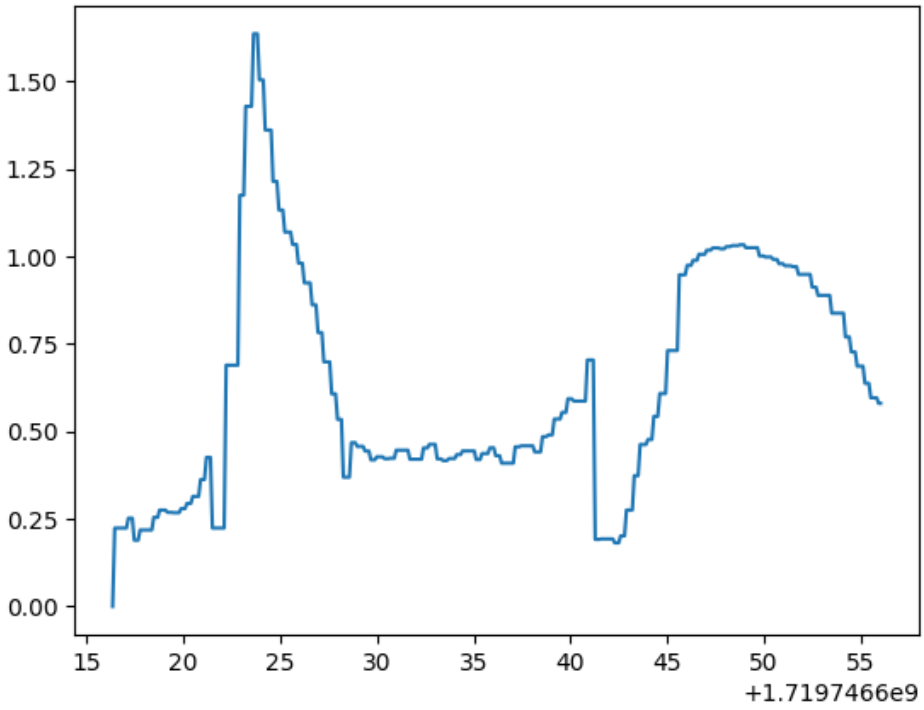

Figure 19: Minimum value of sonar range during the execution of configuration #5

```
Action  ('GoTo', 'box3')  completed
Executing action: ('PickUp', 'box3')
sonars = 1.73 1.61 1.05 1.03 1.18 2.82 3.00 3.00 3.00 2.24 1.87 1.75
Goal achievement: 0.00
(vlin, vrot)) = (0.22, 1.84)
pose = (3.40, -0.51, 98.85)

Action  ('PickUp', 'box3')  failed
Plan execution failed! Replanning...
Planner called from state:
    s1.pos = {'box1': (-4.0, -6.0, 0.0, 0.5), 'box2': (1.6220375299453735, -0.0021450768690556288, 0.0, 0.5), 'box3': (1.198
ble1', 'me_coords': (3.4026052613389504, -0.5114433736219085, 1.72522278816764)}
    s1.room = {'bed1': 'Room1', 'wardrobe1': 'Room1', 'fridge1': 'Room2', 'sink1': 'Room2', 'stove1': 'Room2', 'entrance': '
om4', 'box1': 'Room2', 'box2': 'Room4', 'box3': 'Room4', 'me': 'Room4'}
    s1.door = {'D3': 'Open', 'D2': 'Open', 'D4': 'Open', 'D1': 'Open'}
    s1.connects = {'D3': ('Room3', 'Room1'), 'D2': ('Room4', 'Room1'), 'D4': ('Room3', 'Room2'), 'D1': ('Room4', 'Room3')}
    s1.holding = {'me': None}
** pyhop, verbose=2: **
   state = s1
   tasks = [('transport', 'box3', 'bed1')]
depth 0 tasks [('transport', 'box3', 'bed1')]
depth 1 tasks [('fetch', 'box3'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 2 tasks [('navigate_to', 'box3'), ('PickUp', 'box3'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 3 tasks [('GoTo', 'box3'), ('PickUp', 'box3'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 4 tasks [('PickUp', 'box3'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 5 tasks [('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 6 tasks [('GoTo', 'D2'), ('cross_door', 'D2'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 7 tasks [('cross_door', 'D2'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 8 tasks [('Cross', 'D2'), ('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 9 tasks [('navigate_to', 'bed1'), ('PutDown', 'box3')]
depth 10 tasks [('GoTo', 'bed1'), ('PutDown', 'box3')]
depth 11 tasks [('PutDown', 'box3')]
depth 12 tasks []
** result = [('GoTo', 'box3'), ('PickUp', 'box3'), ('GoTo', 'D2'), ('Cross', 'D2'), ('GoTo', 'bed1'), ('PutDown', 'box3')]
```

Figure 20: Output log after failing a `PickUp` action

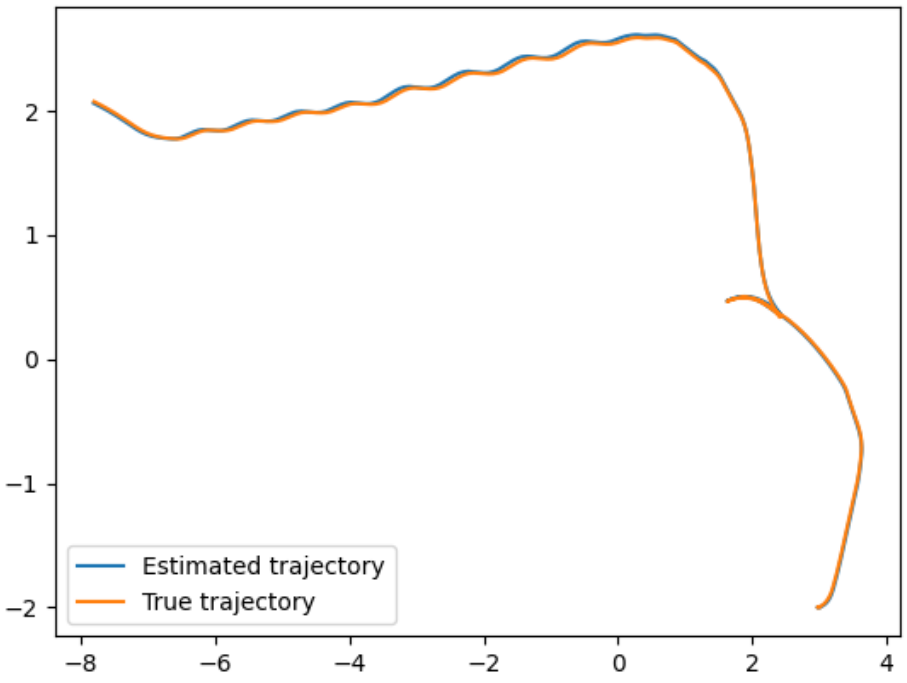

Figure 21: Estimated vs. Real trajectory performed by the robot on goal 1 for optional task #1

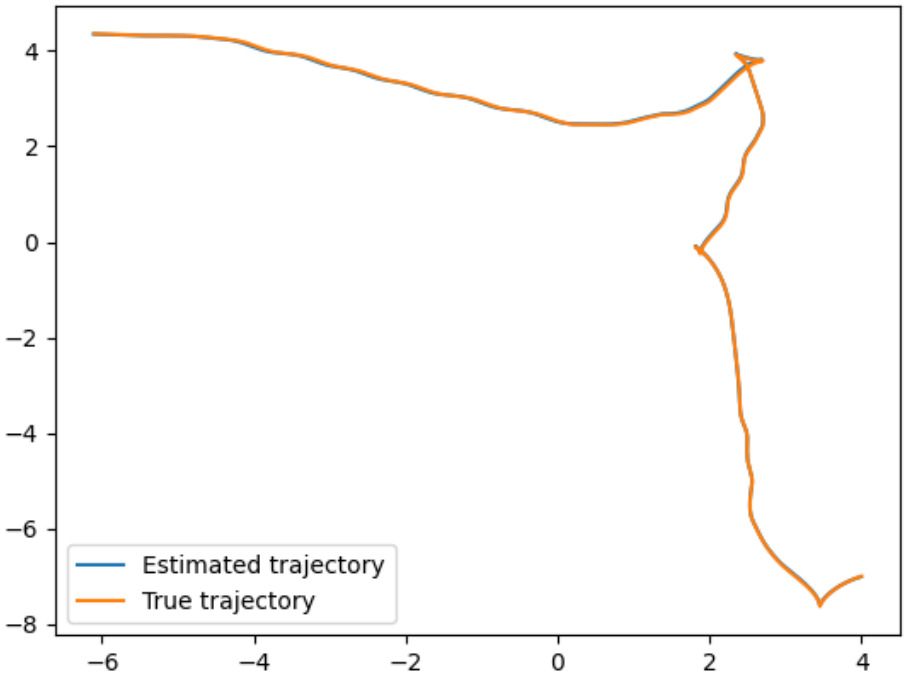

Figure 22: Estimated vs. Real trajectory performed by the robot on goal 1 for optional task #3

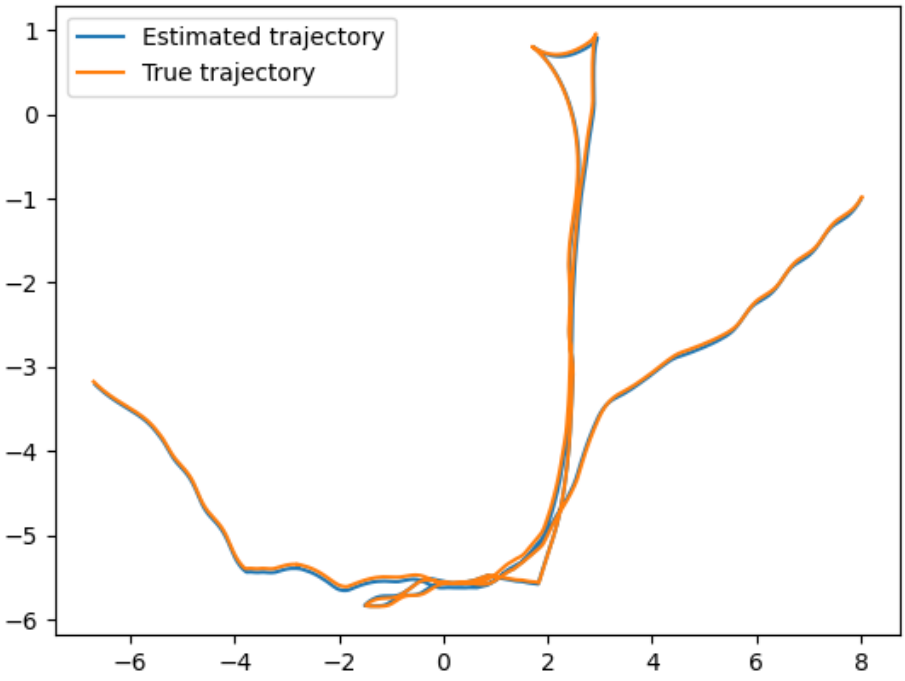

Figure 23: Estimated vs. Real trajectory performed by the robot on goal 4 for optional task #1

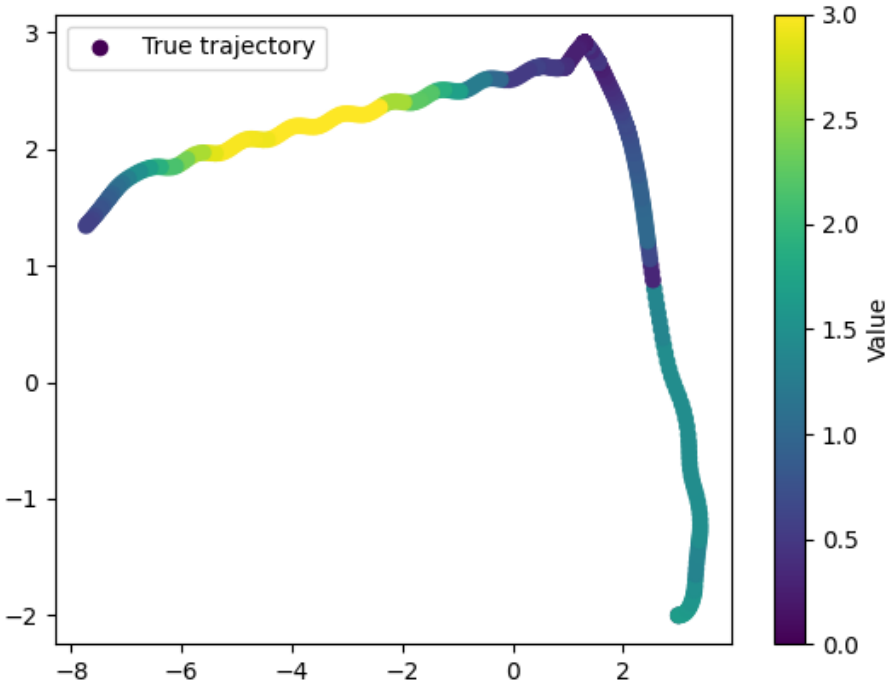

Figure 24: Minimal sonar value depending on robots position