Cooperative swarm behavior generation for multi-platform surveillance

Bruno Mendivez, Jack Murley and Jan Carlo Barca

March 3, 2020

Contents

Abstract

This report describes the design and implementation of a cooperative swarm controller for multi-platform surveillance. Agents are trained with the Multi-agent Deep Reinforcement Learning (MADRL) framework, which makes use of custom reward functions to design a particular set of behaviors that agents share. The environment is set to be observed either by transmission of explicit positions or via a vision model that approximates agents' coordinates with a segmented field of view (bullseye model). The surveillance scenario is defined by world entities such as obstacles, light sources and targets with well defined field of views that the agents must avoid. The final objective is to avoid collisions with peer agents and obstacles while covering all targets. Light sources constitute convenient obstacles that can be crossed if there is no other option to navigate towards a target. The intensity of the light indicates how often it should be treated as an obstacle. The higher the intensity, the more it resembles a regular obstacle, so the likelihood of collision should decrease. Tests were carried out to evaluate the feasibility of the scenario, finding a proper validation and evidence for the correct application of the desired behavioral features in successful missions. Physical testing was also attempted, finding that a more accurate positioning and higher update rate for the Marvelmind hardware is required in order to guarantee a correct operation of the MADRL trainer.

1 Framework

The Multi-Agent Deep Reinforcement Learning (MADRL) framework was introduced in [1]. A scenario definition holds the description of the agents and other entities that share a particular environment. Agents move through the environment so they achieve a certain global goal. For example, there are some targets that the agents must cover in order to succeed. Initially, agents do not hold any type of predetermined knowledge of how to move through the environment and how to interact with other entities. Individual behavior is "learned" through several attempts to reach the goal. Actions are chosen depending on the observation of the environment and later penalized or rewarded, depending on the mission's progress, overall probability of succeeding and how the desired behavioral policies should be learned (*e.g.* tactical policies such as collision avoidance, route planning, etc.)

When we consider multiple agents that interact with each other and the environment, individual actions have collective consequences. Under a cooperative scheme, actions of other agents are optimized in order to improve a collective reward that moves the mission forward.

MADRL defines a training scheme where local learned policies are globally evaluated by a centralized omniscient critic. Once the policies have been optimized after episodic training, the execution can be decentralized and agents autonomous in their decision-making process.

Algorithm 1 MADRL trainer

```
    Input: Agents $A$, world $\omega$
    Output: Saved state for learned policy $\pi$
 1: procedure TRAIN($A, \omega$)
 2:     $X_i \leftarrow \{\}, \forall a_i \in A$
 3:     for all episodes do
 4:         RESETWORLD($\omega$)
 5:         $b_i \leftarrow$ OBSERVEWORLD($a_i, \omega$), $\forall a_i \in A$
 6:         for all steps in episode do
 7:             $c_i \leftarrow$ ACTION($a_i, b_i, X_i$), $\forall a_i \in A$
 8:             WORLDSTEP($\omega$, $\{c_i, \forall a_i \in A\}$)
 9:             $\phi_i \leftarrow$ REWARD($a_i, c_i, \omega$)
10:             $b'_i \leftarrow$ OBSERVEWORLD($a_i, \omega$)
11:             $X_i \leftarrow$ EXPERIENCE($X_i, b_i, b'_i, c_i, \phi_i$)
12:             $b_i \leftarrow b'_i$
13:             if GOALREACHED($A, \omega$) then
14:                 break
15:             end if
16:         end for
17:     end for
18:     $\pi \leftarrow$ SAVESTATE()
19: end procedure
```

2 Training algorithm

MADRL uses episodic training to optimize strategic policies. These policies align with the current scenario and ultimate goal of the environment. The main MADRL training algorithm is defined in Algorithm 1.

Every agent is initialized with an empty experience X_i. As the episodes are executed, individual experiences are refined based on current and previous observations, actions and rewards. The centralized critic discerns between failed policies in an agent's experience and dictates which actions are more suitable. The critic takes into account the aggregated reward of all agents in order to guide the global behavior towards the main goal. An episode ends when all steps were covered or when the goal was reached. At the end of the training routine, the last state is saved as the learned policy π which can be loaded for autonomous operation of the agents. The resulting policy is a neural network for state/action pairs encoded as a collection of tensors.

3 Scenario

Scenarios are particular world arrangements that consist of different entities, environment characteristics and wanted behavioral features. World functions are defined for any particular scenario, depending on how agents observe the environment, how they interact with other agents, how rewards are given and how the world is built.

3.1 Observation

Agents observe the world and collect information from it. They measure distances to peers and other entities, but the quality of the observation depends on the "vision" capabilities of the agent. An observation is also an opportunity to collect the agent's own state (*e.g.* position, velocity). Such local information is usually more precise than measurements that regard other entities. Observations rely on the type of communication used by the agents. For instance, if there is support for a two-way communication where any type of data can be exchanged then such communication link is *explicit* and therefore the observation, since positions and velocities can be shared. In contrast, observations that rely on *implicit* communications do not have access to exact positional information. Agents do not "talk" to each other, so the alternative is to figure out the world state via vision, or be aided by external sources like stationary sensors.

3.1.1 Target assignment

Under a scenario with *implicit* communication, agents use vision models to approximate the world state. The resulting model of the world defines the entities' states in terms of probabilities or through a more simplified coordinate system. In general, observations are the main input for the routines that select a particular action. Therefore, the chosen model that describes the world must encourage a proper navigability. A common choice for an observation structure is some sort of positional representation of the surrounding entities.

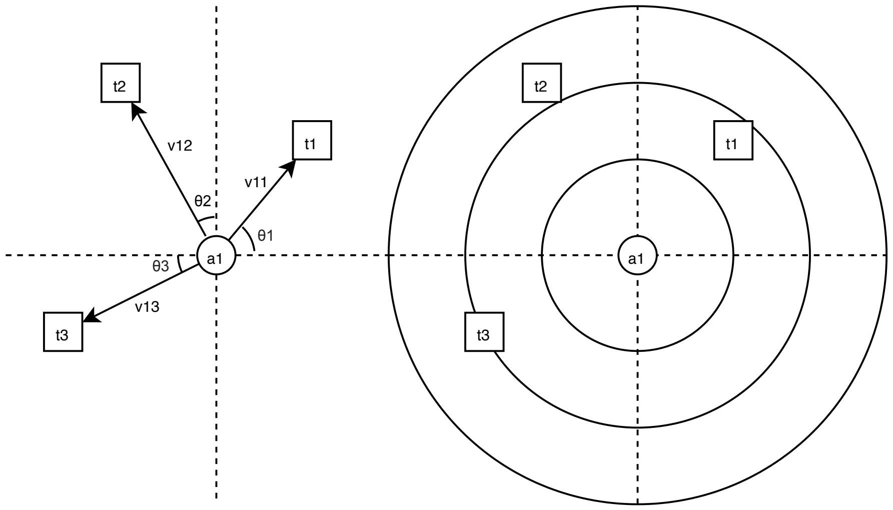


Figure 1: Bullseye vision model for an agent. Four sections and three bins were used to segment the field of view. The resulting view is a vector of entity occurrences count per concentric section.

On the other hand, *explicit* communications enable perfect knowledge of positions and velocities of surrounding entities. Under a real implementation case, this can be achieved by a two-way communication setup where local information is exchanged between agents. Nonetheless, different levels of noise can interfere with a perfect communication scheme so, in some cases, incoming data can be just as unreliable as vision models.

Observation algorithms depend on how the world is constructed, the nature of the entities and the overall goal design. For this project, one particular scenario will be described up next. It will be improved progressively by adding new types of entities and behavioral features.

Consider *n* agents inside a two-dimensional arena together with *m* targets. The coordinates indicating the positions of the targets are known by the agents at all times. Agents have two types of coverage that define the interaction with the environment. First, a *communication range* that involves the use of vision in order to infer peer positions in an agent's field of view. Second, a *sensor range* used to detect if an agent has reached another entity (*e.g.* obstacle, peer, target, light, etc.) by proximity detection. A set of stationary external sensors (beacons) aid the agents with positional information. There is an ultrasonic RF link between each agent and the beacons so the positions can be inferred by the measurement of the RF signal's propagation delay and trilateration [2]. The goal is to have all targets covered by one or more agents.

An implicit communication procedure is used in this scenario. Positions of surrounding entities are collected as occurrences in the observer's segmented field of view (*i.e.* bullseye vision model). Figure 1 shows a random arrangement of one agent and three entities (*e.g.* targets or peer agents). The agent's vision model collects the dis-

Algorithm 2 Bullseye view

```
    Input: Agent $a_i$, entities $T$, sections $n$, bins $m$
    Output: Histogram by section $h_j$
 1: procedure HistogramBySection($a_i, T, \omega$)
 2:     $d_j \leftarrow \{\}, \forall j \in [1,s]$
 3:     for all $t \in T$ do
 4:         $\mathbf{v} \leftarrow \mathbf{t} - \mathbf{a_i}$
 5:         $j \leftarrow \lfloor \text{atan2}(v_y, v_x) n/360 \rfloor + 1$
 6:         $d_j \leftarrow d_j \cup \{\|\mathbf{v}\|\}$
 7:     end for
 8:     for all $j \in [1,s]$ do
 9:         if $d_j \neq \{\}$ then
10:             $h_j \leftarrow$ Histogram($d_j, m$)
11:         else
12:             $h_j \leftarrow \{\}$
13:         end if
14:     end for
15: end procedure
```

tances to entities that fall into the agent's field of view sections. In this case, the segmentation is done in four sections of 90 degrees. The angle of the vector from an agent to an entity determines in which section the entity is. Then, for each section, distances from agent to entities are collected. Finally, to increase the agent's observability, a histogram is calculated from the set of distances in each section. The number of bins can be changed in order to further increase the granularity of the field of view. Bins group the number of distance occurrences for each section. At the end, the vision model returns the number of entities that are inside individual sections of a bullseye-like field of view.

Algorithm 2 describes the procedure to obtain a histogram of distances for each section of the bullseye view model. For every entity, the vector $\mathbf{v}$ from agent a_i to entity t is calculated. To determine the entity's section, the angle of $\mathbf{v}$ is used through the equation in line five, together with the number of sections s. Distances $\|\mathbf{v}\|$ are then stored in the set d_j for any section j. Finally, histograms per section are stored in h_j by taking the distance set d_j as input, together with the number of bins b. A generalization for this procedure would take an agent's communication radius and limit the maximum distance considered to be inside a section.

The output from the vision model is just one aspect of the final observation vector. The full observation procedure is described in Algorithm 3. The procedure begins by gathering all vectors from agent to targets $(\mathbf{s}_j)$, for all targets t_j. The observation vector is preliminary composed by the agent's position $(\mathbf{a}_i)$, velocity $(\mathbf{v}_i)$ and vectors to targets $(\mathbf{s}_j)$. Peer agents of a_i are used for the bullseye model if the communication type is implicit. In such case, all sections/bins are returned and stored in the observation vector. Otherwise, if the communication type is explicit, vectors from agent to peers $(\mathbf{s}_i)$ are returned.

Algorithm 3 Observation

```
    Input: Agent $a_i$, targets $T$, world $\omega$
    Output: Observation $b_i$
 1: procedure OBSERVATION($a_i, T, \omega$)
 2:     $n \leftarrow$ SECTIONS($\omega$)
 3:     $m \leftarrow$ BINS($\omega$)
 4:     for all $t_j \in T$ do
 5:         $\mathbf{s}_j \leftarrow \mathbf{t} - \mathbf{a}_i$
 6:     end for
 7:     $b_i \leftarrow \{\mathbf{a}_i, \mathbf{v}_i, \mathbf{s}_j\}, \forall j \in [1, |T|]$
 8:     $p_i \leftarrow$ PEERS($a_i$)
 9:     if COMMUNICATION($\omega$) is implicit then
10:         $h_k \leftarrow$ HISTOGRAMBYSECTION($a_i, p_i, n, m$), $\forall k \in [1, n]$
11:         $b_i \leftarrow b_i \cup \{h_k\}, \forall k \in [1, n]$
12:     else
13:         $\mathbf{s_i} \leftarrow \mathbf{p} - \mathbf{a}_i, \forall p \in p_i$
14:         $b_i \leftarrow b_i \cup \{s_i\}$
15:     end if
16:     return $b_i$
17: end procedure
```

3.2 Reward

Besides the introduction of additional entities through the world definition and vision model, behavioral features can be learned progressively by designing a custom reward function. Let us first define a simple reward function that will be used for our target assignment scenario. The entities are just agents and targets and the goal is to reach all of them. However, in order to illustrate how behavior can be taught trough this function, let us introduce a collision avoidance strategy by penalizing them. In contrast, a reward (or lower penalization) will be given when an agent gets closer to a target, which is the desired behavior.

Algorithm 4 describes a simple reward function that implements collision avoidance strategies. Each agent calculates a collective reward first and then individual penalties for collisions. Moving towards the goal is encouraged since the reward function sums all minimum distances from all agents to all targets. The negative value is taken since longer distances can be seen as a penalty, while shorter ones will increase the overall reward. At the end, the trainer will maximize the total reward in the system so higher values are desirable. Individual penalties are given when a collision is detected so such behavior is not repeated in subsequent episodes.

3.2.1 Target field of view

Reward functions can be modified to encourage agents to learn certain behaviors. They are also useful for introducing new entities or properties for existing ones. For example, when agents must approach targets from a particular direction only. This is the target's

Algorithm 4 Simple reward function

```
    Input: Current agent $a_i$, agents $A$, targets $T$, penalty $\sigma$
    Output: Current agent's reward $\phi_i$
 1: procedure REWARD($a_i, A, T, \sigma$)
 2:     $\phi \leftarrow 0$
 3:     for all $t \in T$ do
 4:         $\phi \leftarrow \phi - \min(\|\mathbf{t} - \mathbf{a}\|, \forall a \in A)$
 5:     end for
 6:     if COLLIDES($a_i$) then
 7:         for all $a \in A - \{a_i\}$ do
 8:             if ISCOLLISION($a, a_i$) then
 9:                 $\phi \leftarrow \phi - \sigma$
10:             end if
11:         end for
12:     end if
13:     return $\phi_i$
14: end procedure
```

field of view, which constitutes a rejecting region for incoming agents. Covering the target without being noticed is the encouraged behavior in this case. Figure 2 shows the field of view representation where the circular section from θ_1 to θ_2 defines the region from which agents are allowed to reach the target.

In order for agents to learn this behavior, any contact with the FOV region will be penalized. Anytime an agent a_i reaches a target t, it is true that $\|\mathbf{t} - \mathbf{a}_i\| \leq r_s$ where r_s is the agent's sensor radius. The position of the agent relative to the target is described by the angle $\theta = \text{atan2}(a_{i_y} - t_y, a_{i_x} - t_x)$.

Formally, $\phi_i = \phi_i - \sigma$ if $\theta < \alpha_i^t \vee \theta > \alpha_j^t$ where α_i^t is the starting angle of the acceptance region and α_j^t the end angle for the acceptance region of target t.

3.2.2 Obstacles

The same collision avoidance strategy that was applied to peer agents can be applied to obstacles. However, we can introduce a positive reinforcement by rewarding agents that keep their distance from obstacles. The orientation of the agent can be considered as well. If an agent is keeping a direct trajectory towards an obstacle, the angle between the two entities would be small. Hence, moving towards an obstacle would be penalized (small angle) and moving away from it (large angle) would be rewarded. The angle between two entities e_i, e_j that want to avoid a collision is

$$\theta_{ij} = |\,\text{atan2}(e_{i_y}, e_{i_x}) - \text{atan2}(e_{j_y}, e_{j_x})| \tag{1}$$

In terms of distance between the two entities, if it is shorter than the sensor radius (*i.e.* $\|\mathbf{e}_j - \mathbf{e}_i\| < r_s$), then the agent is rewarded if it moves away from the obstacle, hence

$$\phi_i = \phi_i + \|\mathbf{e}_j - \mathbf{e}_i\| + \theta_{ij} \tag{2}$$

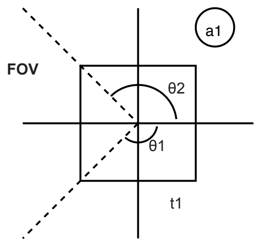


Figure 2: A target's field of view defines the region to be avoided by the agents. The circular section from θ_1 to θ_2 defines the region from which an agent can approach the target.

3.2.3 Lights

Another type of entities are lights. They basically behave like obstacles but it is alright going through them depending on the intensity. For high intensity lights the priority is to avoid them; however, they can be passed through as a last resort. Lower intensities are more lenient in terms of navigability.

For a light source l, if the distance from agent to light is less than the sensor radius, the reward for staying away from the light is higher if the intensity δ_l is high as well. Formally

$$\phi_i = \phi_i + \delta_l \|\mathbf{a}_i - \mathbf{l}\| \tag{3}$$

Algorithm 5 implements the reward function for the final target assignment scenario. An additional penalty was included so agents do not move outside of the world boundaries.

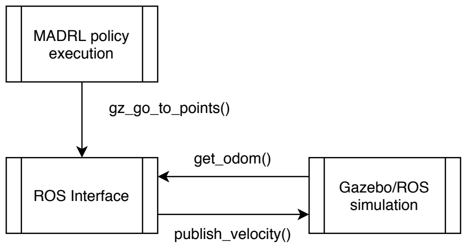
4 Experimental design

The target assignment scenario was tested with TurtleBot simulations in Gazebo. Each robot was controlled through ROS (Robot Operating System) nodes and the rest of the entities were positioned in the Gazebo environment at runtime. The MDRL trainer defined a set of useful policies and then all agents could run autonomously. The execution of the learned behaviors was exclusively commanded by MADRL, in the sense that positions were calculated by MADRL and then forwarded to the Gazebo simulation environment. In order to achieve this, an interface between Gazebo/ROS and MADRL was needed.

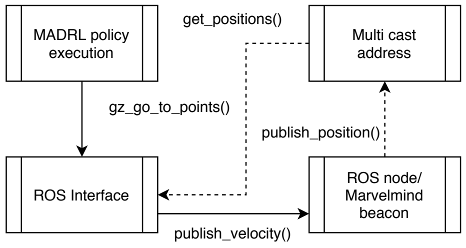
Figure 3(a) shows the communication workflow for the simulations. When the MADRL program executes one step, new positions for all agents are calculated. These positions are transmitted to the ROS interface program, which takes care of each of the robot's movement. To do that, the interface publishes linear and angular velocity vectors and retrieves odometry data in order to track each robot's movements and make sure they reached the desired positions. Once the interface determines that the robots

Algorithm 5 Reward function for the target assignment scenario

```
    **Input:** current agent $a_i$, targets $T$, agents $A$, obstacles $B$,
    lights $C$, sensor radius $r_s$, penalty $\sigma$
    **Output:** current agent's reward $\phi_i$
1:  procedure GETREWARD($a_i, T, A, B, C, r_s, \sigma$)
2:      $\phi_i \leftarrow 0$
3:      for all $t \in T$ do
4:          $\phi_i \leftarrow \phi_i - \min(\|\mathbf{t} - \mathbf{a}\|, \forall a \in A)$
5:          if ISCOLLISION($t, a_i$) then
6:              $\theta \leftarrow \text{atan2}(a_{iy} - t_y, a_{ix} - t_x)(180/\pi)$
7:              if $\theta > \alpha_i^t \vee \theta < \alpha_j^t$ then
8:                  $\phi_i \leftarrow \phi_i - \sigma$
9:              end if
10:         end if
11:     end for
12:     if OUTOFBOUNDS($a_i$) then
13:         $\phi_i \leftarrow \phi_i - \sigma$
14:     end if
15:     if COLLIDES($a_i$) then
16:         for all $a \in A - \{a_i\}$ do
17:             if ISCOLLISION($a, a_i$) then
18:                 $\phi_i \leftarrow \phi_i - \sigma_m$
19:             end if
20:         end for
21:         for all $b \in B$ do
22:             if ISCOLLISION($b, a_i$) then
23:                 $\phi_i \leftarrow \phi_i - \sigma_m$
24:             else
25:                 $do \leftarrow \|\mathbf{a_i} - \mathbf{b}\|$
26:                 $\theta \leftarrow \text{atan2}(\mathbf{b}_y, \mathbf{b}_x) - \text{atan2}(\mathbf{a_i}_y, \mathbf{a_i}_x)$
27:                 if $do < r_s$ then
28:                     $\phi_i \leftarrow \phi_i + do + \theta$
29:                 end if
30:             end if
31:         end for
32:         for all $c \in C$ do
33:             $dl \leftarrow \|\mathbf{a_i} - \mathbf{c}\|$
34:             if $dl < r_s$ then
35:                 $\phi_i \leftarrow \phi_i + dl\delta_l$
36:             end if
37:         end for
38:     end if
39: end procedure
```



(a) Workload for simulations.



(b) Workload for physical testing.

Figure 3: Workflow for the experimental setup.

went to their corresponding positions, the MADRL program can execute the next state and so on.

The velocity controller implemented in the ROS interface is described in Algorithm 6. The robot moves towards the goal point **g** until the distance that separates them is at least γ. The current position **r** and orientation θ of the robot i is retrieved via the odometry module in ROS. A turning angle α is calculated from the current position and the goal point. The turning angle is then refined based on the current orientation of the robot, yielding β, which would later become the angular velocity vector **w**. The linear velocity **v** depends on the current distance to the goal and it is capped at 0.1 to avoid oscillatory movements. Similarly, the angular velocity **w** is corrected as well and limited to the $[-1.5, 1.5]$ range.

When goal points are transmitted from MADRL to the ROS interface program, each robot runs a thread executing Algorithm 6. The ROS interface program waits until all threads return *true*, before MADRL continues with another step.

4.1 Physical testing

Physical tests were conducted in a similar manner as the simulations. The hardware used was the following

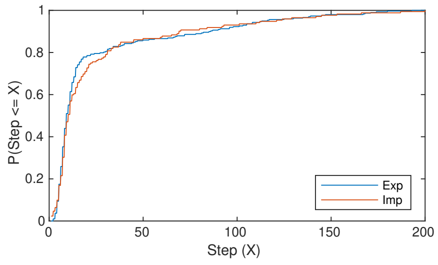
- Turtlebot3 Waffle Pi robot.
- Intel NUC (to run MADRL trainer and ROS).
- TP-Link TL722N wireless dongle for communication.

- Marvelmind indoor GPS system (stationary beacons, mobile beacons and modem).

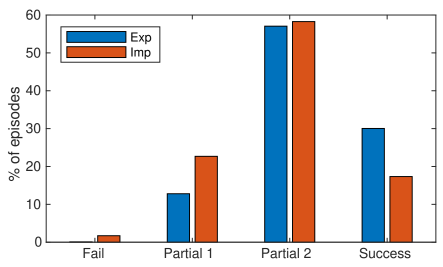
Algorithm 6 Robot's velocity controller for the ROS interface

```
    Input: Goal point $\mathbf{g}$, robot ID $i$, tolerance $\gamma$
    Output: True if robot reached goal point
 1: procedure GoToPoint($\mathbf{g}, i, \gamma$)
 2:     $\mathbf{r}, \theta \leftarrow$ GetOdometry($i$)
 3:     $gd \leftarrow \|\mathbf{g} - \mathbf{r}\|$
 4:     $d \leftarrow gd$
 5:     while $d > \gamma$ do
 6:         $\mathbf{r}, \theta \leftarrow$ GetOdometry($i$)
 7:         $\alpha \leftarrow \text{atan2}(g_y - r_y, g_x - r_x)$
 8:         if $\alpha < 0$ then
 9:             $\alpha \leftarrow 2\pi + \alpha$
10:         end if
11:         $\beta \leftarrow \alpha - \theta$
12:         if $\beta > \pi$ then
13:             $\beta \leftarrow \beta - 2\pi$
14:         end if
15:         $\mathbf{w} \leftarrow (0, 0, \beta)$
16:         $d \leftarrow \|\mathbf{g} - \mathbf{r}\|$
17:         $\mathbf{v} \leftarrow (\min(\{d, 0.1\}), 0, 0)$
18:         if $w_z > 0$ then
19:             $\mathbf{w} \leftarrow (0, 0, \min(\{w_z, 1.5\}))$
20:         else
21:             $\mathbf{w} \leftarrow (0, 0, \max(\{w_z, -1.5\}))$
22:         end if
23:         PublishVelocity($i, \mathbf{v}, \mathbf{w}$)
24:     end while
25:     PublishVelocity($i, (0,0,0), (0,0,0)$)
26:     return true
27: end procedure
```

More information about the physical setup can be found in Appendix 3. Due to the use of the Marvelmind indoor GPS system and the wireless dongle, the communication type was explicit throughout all the experiments. Positional information was retrieved locally by each robot and then transmitted to a multi cast address in the local network so it was available to all agents. The MADRL trainer was executed on each robot and the position outputs sent to the ROS interface program. From there, control is done locally by updating the robot's velocity; however, positional data from other agents was retrieved from the local network. Specifically from a multi cast address where all robots published their Marvelmind positions via wireless communication. Figure 3(b) depicts this workflow. Dashed lines indicate a wireless data flow and a solid line indicates a local one (*i.e.* within each robot's Intel NUC). A new step is executed by



(a) Probability of mission completion time.



(b) Mission status histogram.

Figure 4: Mission status and completion time probability for the obstacle avoidance scenario.

MADRL once goal points have been reached by all agents. This is verified locally based on the shared positions that are published by all peers to the multi cast address.

4.2 Scenarios

The following scenarios were considered for the simulations and physical trials. They constitute modifications of the original target assignment scenario.

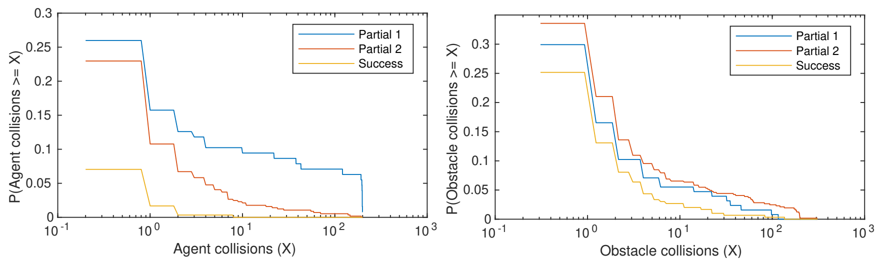
- *Obstacle avoidance*. 3 agents, 3 targets and 4 obstacles participate in this scenario. The communication type can be implicit (4 bins, 4 sections) or explicit. The world is a $10m$ square and agents have a sensor radius of $1m$. Note that physical testings only consider the explicit version of this scenario.

- *Full scenario*. The entities considered in this scenario are the following: 3 agents, 3 targets, 3 obstacles, and 2 light sources. Both communication types were supported as well. As for the target's field of view, the valid region (outside of the FOV) is delimited by the angle range $[-135°, 135°]$. The world dimensions and agents' properties are the same as the obstacle avoidance scenario. Note that for this scenario, only simulations were performed.

Both scenarios were trained considering 50000 episodes with 200 steps each. A checkpoint file was saved at the end of the training process in order to save the state of the neural network that represents the learned policy. Finally, the evaluation set consists of 10000 episodes that were used for benchmarking purposes.

5 Results

We will discuss the simulation results for both scenarios. The episodic benchmark data was used to analyze the resulting behaviors.



(a) Agent collision probability. (b) Obstacle collision probability.

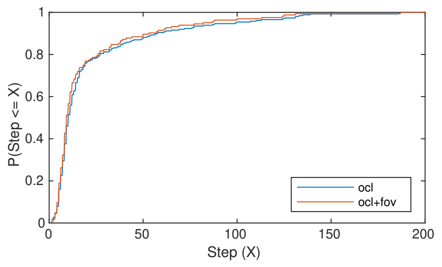
Figure 5: Agent and obstacle collision probabilities for the *obstacle avoidance scenario* under different mission outcomes.

5.1 Obstacle avoidance scenario

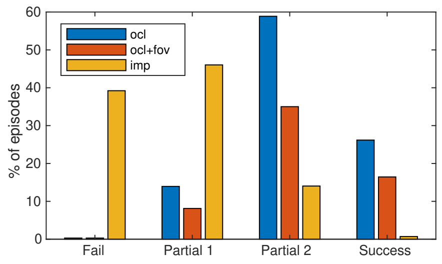
Figure 4(a) shows the probability of mission completion time for different step values. We intend to analyze how fast the agents converge into a successful mission state (*i.e.* all three targets being covered). Both communication types are compared, showing a small advantage of the explicit case over the bullseye vision model. Although it is fair to say that the implicit approximation does a good job at providing agents with an accurate world representation, which is reflected on little to no delay in mission achievements. Similar conclusions can be extracted from Figure 4(b) where different mission outcomes can be seen in terms of percentage of the total benchmark episodes.

It is important to point out that partial completions might consists of targets being covered by more than one agent. However, our explicit definition of successful mission requires all three targets to be covered by one agent. We can see in Figure 4(b) that the overall successful rate is higher under explicit communication, which is expected. Partial completions seem to go either way although there are more total failures (*i.e.* no targets covered) under implicit communication schemes. Overall, observations that rely on implicit communication through the bullseye model show a slight disadvantage for this fairly simple scenario. It would be interesting to observe another comparison where the number of bins/sections changes in order to improve accuracy.

As for the collision avoidance strategies, this simple scenario uses Algorithm 4 to penalize agents that collide with each other or obstacles. The performance of this method can be seen in Figure 5(a) for the case of agent collisions. Successful runs have a lower probability of collisions, which shows that the strategies are being enforced and the fact that agents cover all targets is not a product of random behavior that comes up from several episodic executions. The highest collision probability is found for the partial completions where only one target is covered. This shows that failed attempts to reach a target are caused by agents trying to overcome a challenging path and therefore performing erratic actions that translate to collisions. Similarly, Figure 5(b) shows lower collision probabilities for successful runs when compared to partial completions. In both plots, failed missions were rare and not enough data was available to show significant results.

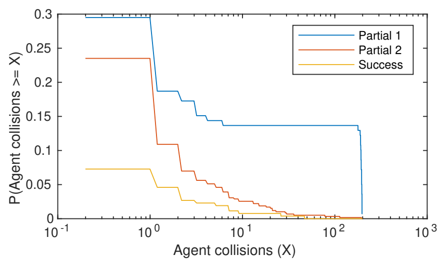


(a) Probability of mission completion time.

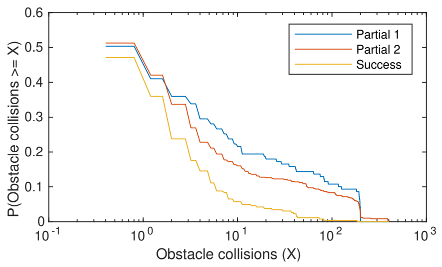


(b) Mission status histogram.

Figure 6: Mission status and completion time probability for the *full scenario*.



(a) Agent collision probability.

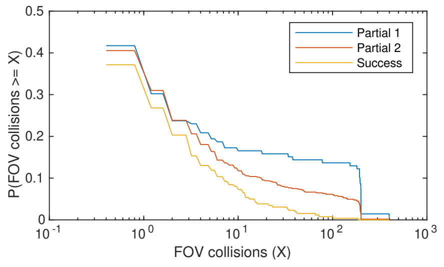


(b) Obstacle collision probability.

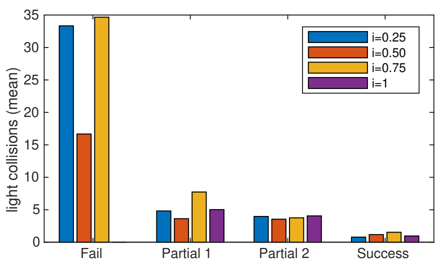
Figure 7: Agent and obstacle collision probabilities for the *full scenario* under different mission outcomes.

5.2 Full scenario

Under a full scenario, new entities were introduced such as light sources and fields of view for targets. As a result, a successful mission is not only when all targets are covered. In addition, a mission is successful when a target was covered without the agent being noticed; in other words, without a collision with the target's FOV. This new mission outcome definition can be analyzed in Figure 6(a), which shows the CDF of the mission completion times under two outcomes: all targets covered (ocl) and valid covering outside of FOV (ocl+fov). In terms of how fast agents converge into any of these states, there is no noticeable difference between them. The probability is slightly higher in the FOV validation case, perhaps by the limited availability of trajectories. Nonetheless, the difference is too small for a definitive explanation. As for the mission outcomes, Figure 6(b) shows a noticeable decrease in "real" successful missions when we take into account the target's field of view. Another important observation is the sharp decrease of successful missions for an implicit version of the full scenario. It shows that for more complex scenarios where several behaviors are to be learned, a limited vision model must heavily rely on a good accuracy of positional information in order to derive a good global performance. Subsequent plots will only make use of the explicit communication version of the full scenario.



(a) FOV collision probability.



(b) Lights collision histogram.

Figure 8: FOV and light collision measurements for the *full scenario*.

As with previous results regarding the correctness of the collision avoidance strategies, Figure 7 confirms the intended behaviors derived from the reward function in terms of agent and obstacle collisions. Successful missions have a lower probability of colliding with other agents and obstacles than partial completion missions. Figure 8(a) shows a similar pattern for the probability of entering a target's FOV.

Finally, agents demonstrate a similar behavior to obstacles and other agents when it comes to approaching lights. The only difference is that collision avoidance is less enforced depending on the intensity of the light source. The result of this strategy can be seen in Figure 8(b). As with other "avoidable" entities, collisions are much lower on successful missions when compared to other outcomes. However, by looking at the intensity of the light sources, there is no evidence that high intensity lights received less collisions, or even the opposite. Clearly, modulating the reward depending on the light intensity seems like a good incentive for obstacles to avoid/pass. However, the true nature of this entity is to provide agents with alternative paths to reach the targets. Rather than defining a reward/penalty for collisions, a much more detailed scenario must be constructed so it is seen as an opportunity by the agent to reach a target. The reward is given for taking the opportunity if some conditions hold (*i.e.* shortcuts).

5.3 Physical testing results

On February the 14th we conducted our physical testing on our modified TurtleBot3 platform which can be seen below in Appendix 3. We ran a total of 11 tests with a variety of obstacle configurations as well as target configurations. Out of the 11 tests only 2 were successful. It needs to be noted that we only considered a mission successful if all 3 targets were covered by 1 or more agents. Also, an explicit communication was considered. Ultimately resulting in the agents having a 18% successful mission rate where they covered all three targets. All 11 tests can be viewed from the following link.

Furthermore, it also needs to be noted that we ended the mission and declared it as unsuccessful when any of the following happened:

- Collision between agents.
- An agent leaving the testing arena.

- An agent hitting an obstacle.
- An agent hitting a target.
- An agent failed to start/move initially.

5.3.1 Physical testing videos explained

For the physical tests that were conducted we decided to only include obstacles as well as the targets in the testing environment. This was primarily due to the issue's described below. Nevertheless, in the physical testing videos you will be able to see 3 cardboard boxes as well as 3 groups of different colored blocks on the ground.

In the physical testing videos, each cardboard box represents an obstacle that the agents have to avoid as they would in either the particle or gazebo simulated environments. Whereas each small group of colored blocks on the ground represent 1 target for the agents to reach and for the mission to be successful all 3 targets must be covered by 1 or more agents.

Within the physical testing, we assumed that each agent had a sensor radius of 0.3 meters, the minimum distance to reach a target. However, we also need to take into consideration that the agents at all times knew where both obstacles and targets were in the environment.

5.3.2 Why the successful mission rate was so low?

As you can see above from the physical testing videos, we can see a very distinctive zig zagging type movement from each of the robots in every single test. The main reason we can see this action is that they are not receiving timely updates about their current location. This is in due fact caused by the localization system that we were using as it was found out while testing that the Marvelmind localization system was highly unreliable in the closed lab environment that we were testing in.

As a result of the poor update rate the robots assumed, they were in a position when in fact they were in another. Ultimately resulting in the following behaviors:

- Leaving the testing arena.
- Crashing into another agent.
- Crashing into obstacles.
- Crashing into targets.

However, we also need to consider that the Marvelmind localization system has distinct downsides and limitations. One direct limitation that affected the physical testing is that the agents needed direct line of sight to the localization system and at times due to the obstacles they lost their line of sight resulting in the agent's location not being updated. Ultimately, resulting in the agents running or moving into a one or more of the five reasons that causes a test to fail. Nevertheless, in the successful tests we can clearly see how the agents lose the zig zagging type movement due to the agents receiving timely location updates from the Marvelmind system.

6 Conclusions and future work

A cooperative behavior designed for surveillance was constructed. The surveillance scenario consists of entities such as agents, targets, obstacles and light sources. Agents are required to reach the targets while avoiding collisions with other agents and obstacles. In addition, an agent can only reach a target by avoiding its field of view.

The MADRL trainer was used to "teach" agents the desired behaviors by modifying a reward function that evaluates the actions taken by the agents at every step in an episodic run. After the training process, agents were put to the test to validate their behavior. Two types of communication were used, implicit which produces observations conducted via a bullseye vision model, and explicit, where positions and velocities were constantly exchanged.

Observations that rely on implicit communication and the bullseye model show similar performance to explicit communication schemes only when the scenario is simple and not many desired behaviors are in place. Increasing the accuracy of the bullseye model by increasing the granularity of the agent's field of view (bins and sections) could improve performance when tested in more complex scenarios like surveillance. The vision model can also benefit from inferring positional information as a set of coordinates, rather than just indicating the occurrence of peer agents in the different sections of the bullseye. An observation or communication radius can also be considered in order to provide a more narrow and thus more precise focus for the bullseye, in terms of computational costs when increasing the segmentation of the field of view.

Reward functions were used to guide the agents into the desired behaviors. The use of conditional rewards or penalties seems to cause sharp changes in the actions taken by the agents. During benchmarks, it was observed that agents tend to get stuck on a "decision making loop", when challenged by situations that would trigger more than one of these conditional penalties/rewards. When the decision making mechanics grow in complexity, the desired behaviors seem to compete for the current action that maximizes the reward, creating an oscillating movement pattern as a result. A more smooth definition of a reward function rather than a piecewise one would be considered to mitigate getting stuck on local minimum/maximum. This would benefit the performance of all aspects of the behavioral design and, for example, improve the mission successful rate for when the agents need to approach a target outside of its field of view.

Additionally, the reward definition for the light sources needs to take another approach since the intended behavior is to consider lights as opportunities to reach a target, rather than something to be avoided on a particular level (*i.e.* as defined by the light intensity). Of course, this would depend on the current conditions and whether or not a light source is useful or not. Such utilitarian approach can define a magnitude for the reward, if we consider lights as shortcuts with a fixed difficulty, directly proportional to the their intensity.

For future iterations and in order to test the robustness of the reward algorithms, noise will be added to both explicit and implicit communication data.

6.1 Future work for physical testing

We plan to implement the full set of features described in the algorithm design sections. For the physical testing, an implicit scenario will be deployed, together with light sources and the target's FOV.

As for the state of the current set of testings, we encountered a variety of issues with the Marvelmind system however, we also found out an error in the ROS code which merged the locations of two robots into one. For example, robot 1 was receiving robots 2 position rather than its own. While this rarely occurred it still caused the robots to not receive their correct location which was the main and only issue we encountered in physical testing aside from the issues with the Marvelmind localization system.

The core issues that we encountered with Marvelmind system was the following:

- Echo in the closed lab environment, which distorted and greatly decreased the update rate.
- Line of sight was often lost when the robots went near an obstacle, which decreased the update rate drastically.
- Firmware stability, we found throughout the project that the stability and reliability of the Marvelmind firmware was far too volatile and as a result caused changing firmware and operating platforms frequently requiring new code to often be written due to the changes.
- Update rate differences between agents and the desktop, the update rate that the desktop displays compared to the robot's update rate is significantly different. One example is the desktop will show the robots have an average update rate of 12 Hz when in reality the robots only have roughly a 1 or 2 Hz update rate.
- Stationary beacon location changes, due to the inverse architecture that we used it required us to manually map and set a $(0,x)$ and $(0,y)$ point for the robots to use and often spawn at. However, we found out early on that the Stationary beacons change their location in the Marvelmind software which then requires a brand new $(0,x)$ and $(0,y)$ point to be calibrated and set. It also needs to be noted that the stationary beacons would sometimes change their location during operation which would then affect the operating perimeter measurements and cause a reason for the test to fail such as leaving the testing arena.

As stated above we also encountered an issue where the robots would merge their own location data and as a result receive the wrong location. While this rarely happened, we noticed that this occurred due to how the Multi-master FKIE package is built as it subscribes and publishes all topics that can be sent out over the multi cast IP. Furthermore, due to how the differences in the topic names were so minimal such as `Robot1/Location/X`, `/Robot2/Location/X` it sometimes sent out the wrong subscribed topic to an agent resulting in the merged location data.

6.1.1 Fixing the Marvelmind issues

Fixing the variety of issues listed above with the Marvelmind system is rather simply and not overly complicated. Fixing the echo in the closed lab environment can easily be fixed by running the physical tests in an open environment where the ultrasonic frequencies would not be able to bounce off any walls and interfere with each other or the receiving beacons on the ground.

Fixing the line of sight issues, we encountered when using the cardboard boxes can be fixed by placing paper of the ground as the obstacle rather than a cardboard box which would not interfere with the line of sight. Another possible solution could be that we have an additional Marvelmind beacon on the ceiling facing directly downwards which should compensate and still allow for line of sight between agents.

Fixing the firmware stability is an issue that is outside of our control as we do not have access or the ability to modify the dashboard in any shape or form. However, we can move over to Marvelmind's NIA (Non-Inverse Architecture) which the company supports and develops as the main architecture that they use which in turn should provide a higher level of stability.

Fixing the transparency of the update rate between the agents and the dashboard is another issue which we cannot directly control due to not being able to modify the dashboard. However, what we can do instead is track and record the update rate of each agent and publish it to a cloud server which then shows the accurate update rate of each individual robot.

Fixing the stationary beacon location changes is another issue we cannot directly fix due to it being a firmware issue. However, we can work around this issue by changing over to the NIA in which the beacons automatically measure the distance from each other and if we place a beacon on each corner of the operating perimeter we can easily set a $(0,x,0,y)$ that way and write a simple script for the code to adopt this change.

6.1.2 Fixing the merged location issue

Fixing the merged location data that we encountered while testing can very easily be fixed. This can simply be done by ensuring that every robot has a unique subscribed published name. For example:

```
Robot1/Robot1_Location/Robot1_X
Robot2/Robot2_Location/Robot2_X
```

With this simple change in how we name the subscribed topics, we can remove the issue of the robots merging their locations with each other.

6.1.3 Possible major changes to remove other issues

A variety of major changes could be implemented to remove the need for a localization system like the Marvelmind system. One significant change would be to use IMU data collected from the OpenCR board on the TurtleBot3 in conjunction with the LDS (Laser Distance Sensor) on top of the TurtleBot3 for both location and collision avoidance tracking abilities.

By offloading the work to the built in IMU data and LDS sensor for location and collision avoidance it would allow us to have a significantly higher location update rate due to how the OpenCR board is directly connected to the Intel NUC. Which in turn should theoretically allow us to receive an agent's location as fast as the network permits it as well as by providing location via IMU we do not need to worry about our operating arena/area, allowing for more accurate real-world tests to be conducted.

The change over to IMU data rather than the Marvelmind localization system should be relatively simple as there is already a large source of robotic applications that base location off of IMU data. Furthermore, there is open source code written specifically for the TurtleBot3 platform to conduct both SLAM and collaborative SLAM in ROS which could be used as a basis to implement IMU location tracking abilities. Nevertheless, there is also code written for obstacle avoidance with the LDS sensor that the TurtleBot3 uses making the process into integrating over to IMU data and using the LDS for collision avoidance faster and more streamlined.

Another major change that could be implemented to potentially increase the performance of the agents during operation could be swapping over to ROS 2 rather than ROS. ROS 2's implementation of middle ware is based on the DDS (Data Distribution Service) standard whereas ROS relies on a central discovery mechanism. The main benefit of ROS 2 over ROS is that we can create multiple notes in a process which is something that ROS is unable to do as well as through ROS 2's DDS standard it will allow for an improved transmission rate between agents.

Ultimately ROS 2 will allow us to have an increased transmission rate due to the DDS as well as an increased node management. This unique feature will allow us to have greater control over the state of the ROS system and indeed allow `roslaunch` to ensure that all components have been instantiated correctly, thus allowing nodes to be restarted or replaced online which increases both the robot's and operator's safety.

References

[1] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, pages 6379–6390, 2017.

[2] M. Robotics. Precise (2 cm) indoor gps: For autonomous robots, copters and vr, 2017.

A Installing the MADRL trainer

When installing and setting up the particle environment on a new machine we heavily recommend that you initialise a standalone Python environment with pyenv due to the ease of use. Therefore, execute the following in a terminal:

```
$ sudo apt-get update
$ sudo apt-get install -y curl git make build-essential libssl-dev
zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev wget curl
llvm libncurses5-dev libncursesw5-dev xz-utils tk-dev libffi-dev
liblzma-dev python-openssl
$ curl https://pyenv.run | bash
```

Then replace `hfolder` with your home folder and execute

```
$ echo 'export PATH="/home/hfolder/.pyenv/bin:$PATH"' >> .bashrc
$ echo 'eval "$(pyenv init -)"' >> .bashrc
$ echo 'eval "$(pyenv virtualenv-init -)"' >> .bashrc
```

You can query the Python versions that are available to install

```
$ pyenv install --list | grep " 3\.[5]"
```

Then select version 3.5.4

```
$ pyenv install -v 3.5.4
```

Now create a folder that will contain the MADRL source

```
$ mkdir mal
$ cd mal
```

Attach a virtual environment to it

```
$ pyenv virtualenv 3.5.4 malvenv
$ pyenv local malvenv
```

Verify Python version

```
$ python --version
```

Install dependencies

```
$ pip install ‘numpy==1.14.5’
$ pip install ‘gym==0.10.5’
$ pip install ‘tensorflow==1.8.0’
$ pip install scipy
$ pip install matplotlib
```

Download the assfiles.tar from the link supplied below and copy the assfiles.tar into the mal folder

```
https://github.com/JTMurley/aasIntialCodeFile
```

Then execute

```
$ tar xvzf aasfiles.tar.gz
```

Go to `maddpg` and execute

```
$ cd aasfiles/maddpg
$ pip install -e .
```

Then go to `multiagent-particle-envs` and execute

```
$ cd ../multiagent-particle-envs
$ pip install -e .
```

Finally, go to `tools/other_src`

```
$ cd ../tools/other_src
```

Execute

```
$ tar zxvf zeromq-4.3.1.tar.gz
$ cd zeromq-4.3.1/
```

Make sure you install these dependencies `libpgm-dev` and `libnorm-dev`. The package `libnorm-dev` will require the `stretch-backports`

```
$ sudo nano /etc/apt/sources.list
```

Add the following at the end of the file

```
deb http://mirror.aarnet.edu.au/debian stretch-backports main
```

Now install the dependencies

```
$ sudo apt-get install libpgm-dev libnorm-dev
```

While in the `zeromq-4.3.1` directory execute

```
$ ./configure --with-pgm --with-norm
$ make
$ sudo make install
```

Then install `pyzmq`

```
$ cd ..
$ tar zxvf pyzmq-18.0.2.tar.gz
$ cd pyzmq-18.0.2
$ python setup.py install --zmq=/usr/local/
```

That should complete the installation of all dependencies. The main file that we use to start the training is `mal/aasfiles/maddpg/experiments/run.sh`
You need to change this file so the `PYTHONPATH` variable is pointing to the correct directories.

Workflow

Changes in the scenarios are done in the `multiagent-particle-envs` project. Here is where the code for designing the reward function lives. Here is also where the design of entities (agents, obstacles, world in general) is implemented.

The `maddpg` project contains code for training. The main entry point is `mal/aasfiles/maddpg/experiments/train.py`. You can see the full list of parameters at the beginning of the file. I will try to indicate the most important ones. The main training algorithm works as follows

1. Train: train() in maddpg/experiments/train.py
    a. Make environment: make_env() in maddpg/experiments/train.py
        i. We consider an internal environment `--environment-type INTERNAL` (simulation)
        ii. External is used when it is time to deploy to real agents
        iii. Make world: `make_world()` in `multiagent-particle-envs/multiagent/scenario.py`
            1. Defines dimensions of the world
            2. Defines propagation type `--propagation-type` (`IDEAL` if transmission of current position is done to all agents, `DISC` if we consider a communication radius `--comms-radius-m`)
            3. Make world: `_make_world()` in `multiagent-particle-envs/multiagent/scenarios/target_assignment/target_assignment.py`
                a. Creates agents, landmarks (targets) and obstacles
                b. Define properties for each (id, name, can_collide, etc.)
            4. Last time to update agent's parameters (location noise, if agent can broadcast)
            5. Reset world: `reset_world()` in `multiagent-particle-envs/multiagent/scenarios/target_assignment/target_assignment.py`

                a. Set random initial states for agents, targets and obstacles.
        iv. Get environment. If benchmarking, environment supports measurements.
            1. Make environment: callback to `__init__()` in `multiagent-particle-envs/multiagent/environment/environment_base.py`
                a. Defines parameters for the environment (agents, world, callbacks for reward functions, benchmark functions, etc.)
                b. Indicates whether the reward is shared (collaborative scenario) or not
                c. Defines a discrete action space
                d. Defines action and observation spaces (the dimension of the data structures that will contain the set of actions and observations).
                e. The action space is a vector of 2*world_dimensions + 1 (5 elements).
                f. The observation space depends on the type of observation (**implicit** or **explicit**). More on that later.
                g. Action and observation spaces depend on the number of agents.
    b. Get the size of the observation space for the environment
    c. Initialise trainers: get_trainers() in `maddpg/experiments/train.py`
        i. Since no adversaries are considered, for each agent get one trainer.
        ii. A trainer is an agent that executes the MADRL model. It needs to know the size of the observation space, and the action space.
        iii. Trainers calculate actions based on observations.
    d. If benchmarking, execute episodes based on the already trained agents.
    e. Reset the environment. Make first a first observation for each agent.
        i. Agent makes an observation: callback to `observation()` in `multiagent-particle-envs/multiagent/scenarios/target_assignment/target_assignment.py`
            1. If observation type `--observation-type` is `IMPLICIT` then get the following for the observation space
                a. Field of view of agent is divided into sections and each section into bins. Get histogram of peer occurrences for the entire field of view.
                b. Histogram but now for landmarks.
                c. Agent's velocity.
                d. Agent's position.
                e. Vectors from agent to landmarks.
            2. If observation type is `EXPLICIT`, the observation space is conformed by
                a. Agent's velocity.
                b. Agent's position.

        c. Vectors from agents to landmarks.
        d. Vectors from agents to peers (if DISC propagation is selected, consider only peers inside communication radius)

f. While true
    i. Get actions from trainers and their observations.
    ii. Apply actions to the environment: `MultiAgentEnvInternal._step()` in `multiagent-particle-envs/multiagent/environment/environment.py`
        1. For each agent, set action: `_set_action()` in `multiagent-particle-envs/multiagent/environment/environment_base.py`
            a. Update the internal action of each agent with the one obtained from the trainer.
        2. Advance the world one step: `step()` in `multiagent-particle-envs/multiagent/core.py`
            a. Apply action force for each agent (action plus noise).
            b. Integrate the state of the world. For each movable entity (agent)
                i. Update agent's velocity = (action force / mass) * time_step
                ii. Update agent's position = agent's velocity * time_step
        3. For each agent, update agent peers: `update_agent_peers()` in `multiagent-particle-envs/multiagent/agents.py`
            a. Add noise to agent's location.
            b. Broadcast location: `_broadcast_location()` in `multiagent-particle-envs/multiagent/agents.py`
                i. Put agent's position in peers that are inside of communication radius.
                ii. This would not be used if using the implicit communication but it is still executed.
        4. For each agent
            a. Get a new observation.
            b. Get an individual reward: `reward()` in `multiagent-particle-envs/multiagent/scenarios/target_assignment/target_assignment.py`
                i. Reward is the sum of all distances to targets, times -1.
                ii. Penalties (reward - 1) are granted if a collision is detected (with either another agent or obstacle).

            c. See if the agent has reached the target (mission status).
            d. Get data for benchmarking (if applicable).
        5. Total reward is the sum of all individual rewards. If shared reward, all rewards are the total reward.
        6. Return new observations, rewards, mission status and benchmarking data (if applicable).
    iii. Episode step = episode step + 1
    iv. Get global mission status
    v. For each agent, gain experience. Experience is the history of individual observations, actions, and rewards.
    vi. If mission status is successful then finish the episode and reset the environment.
    vii. If benchmarking, write benchmark data.
    viii. If displaying, render the world.
    ix. Break if all episodes were recorded.

Experiments are specified in `run.sh` as

```
python train.py [params]
```

Then

```
$ bash run.sh
```

B Using the MADRL trainer

Using the Particle Environment

Within the particle environment we need to know that we use the run.sh file to declare the parameters of the experiments that we run. The run.sh file can be found under `aasfiles/maddpg/experiments`.

The parameters that are available for us to use can all be found within the train.py file which is found in the same directory as the run.sh file.

1. Train.py Parameters Explanation

Within the Train.py file there are a variety of parameters that you can add to the particle environment. The parameters are broken up into 8 categories:

- Scenario
- Learning
- Propagation
- Experiment
- Environment
- Core Training
- Checkpointing
- Evaluation

1.1. Scenario Parameters

Within the Scenario parameters we have 9 key parameters we need to consider in the particle environment:

1. Exp-name, this is the name of the experiment and takes a string input. Note you will want to change this name for each new experiment as if it is the same it will overwrite the file, and it saves the data under `aasfiles/maddpg/data`
2. Num-agents, this is how many agents are in the experiment and takes an int input. By default, it is 3 agents in the environment however you can change this to as many you want.
3. Num-targets, this is how many targets the agents will seek out and takes an int input. Note that these are also referred to as landmarks in the code. This is by default set to 3.
4. Num-obstacles, this is how many obstacles are present in the environment and takes an int input. By default, it is 3 however it has gone as high as 30 in testing
5. Num-lights_100, this is how many lights at intensity 1 are present in the scenario and takes an int input. This is the highest intensity light and the most penalised as it is considered as direct sunlight.
6. Num-lights-75, this is how many lights at intensity 0.75 are present in the scenario and takes an int input. This is the second highest intensity light and the second most penalised as it is considered as a lightly shaded area
7. Num-lights-50, this is how many lights at intensity 0.5 are present in the scenario and takes an int input. This is the third highest intensity light and the third most penalised as it is considered as a very poorly lit shaded area
8. Num-lights-25, this is how many lights at intensity 0.25 are present in the scenario and takes an int input. This is the lowest intensity light and the least penalised and is considered as a barely visible lit area.
9. Target-fov, whether or not to consider targets' FOV.
10. Out-fov-start, start angle for the region outside of the FOV. Counterclockwise direction.
11. Out-fov-end, end angle for the region outside of the FOV. Counterclockwise direction.

1.2. Learning Parameters

Within the learning parameters we have 8 key parameters we need to consider:

1. Peer-data-threshold-timeout, this is the peer data timeout (steps) and takes an int input. By default, this is 10.
2. Num-sectors, this is the number of observation sectors that get sent, this takes an int input. For example, if 4 sectors are given it provides 90-degree quadrants
3. Num-bins, this is the number of histogram bins, this takes an int as the input.
4. Bin-type, this is the type of bin that is present, by default it is Linear however it can also be changed to power. The main difference is the way in which the histogram rings are spaced
5. Transmit-history-size, this is the size of the transmission history size, it takes an int as the input and by default is 10.
6. World-size, this is the how big the world is, it takes a float as the input. By default, it is set to 100.
7. Observation-type, by default the observation type is explicit however it can also be changed to implicit. In testing we achieved similar results with both explicit and implicit communication types
8. Environment-type, by default it is set to internal and this is to indicate whether we are using the particle environment or an external environment. We would change this to external when we are testing outside of simulation.

1.3. Propagation Parameters

Within the propagation learning parameters, we have 8 key parameters we need to consider when using the particle environment:

1. Propagation-type, this is the propagation type that is currently being used. By default, it is set to ideal, however we also have the choices between disc, log_distance and log_normals. Note that it is all measured in meters as well.
2. Comms-radium-m, this is the distance of the communication radius, it takes a float input and is set to 20 meters by default.
3. Path-loss-exponent, this is the path loss exponent for log distance/normal path loss and takes a float input by type and is set to 2.4 by default.
4. Ref-dist-m, this is the reference distance for the log distance/normal path loss (meters). It takes a float input and is set to 1 by default.
5. Frequency-hz, this is the communications frequency (hz) and by default is set to 802.11 channel 1. It takes a float input.
6. Shadowing-std-dev, this is the shadowing variable for the log normal propagation (db). It takes a float input and is set to 3 by default.
7. Tx-power, this is the transmission power in dBM. It takes a float input and is set to 20 by default
8. Snr-receive-threshold, this is the signal to noise ratio receive threshold (dB) and takes a float input and it is to 3 by default.

1.4. Experiment Parameters

We have 3 main parameters that we need to consider in the experiment parameters:

1. Sensor-radius, this is the radius of the circular sensor. By default, it is set to 10 and takes a float as the input
2. Position-error-std-dev, this is the standard deviation of the normal position error measured in meters. By default, it is set to 0 and takes a float as the input.

3. Always-broadcast, this is type bool and is set to True by default. When set to true, the agent broadcasts every step, however when it is set to false the transmission is part of the agent action decision and this is all measured in meters.

1.5. Environment Parameters
We have 4 main parameters that we need to consider in the environment parameters:

1. Scenario, this is the scenario script that is present on the agents, by default it is set to target_assignment as this is the only current script that has been made.
2. Max-episode-len, this is the maximum episode length, by default it is set 200 and takes an int as the input.
3. Num-episodes, this is the total amount of episodes, by default it is set to 25,000 and takes an int as the input.
4. Good-policy, this is the policy that is present for good agents, it is set to maddpg by default and takes a string as the input.

1.6. Core Training Parameters
We have 4 core training parameters that we need to consider when using the particle environment in both internal and external environments:

1. Lr, this is the learning rate for the Adam optimizer, by default it is set to 1e-2 and takes a float as the input.
2. Gamme, this is the discount factor that we use, it is set to 0.95 by default and takes a float as the input.
3. Batch-size, this is the number of episodes to optimize at the same time, it is set to 1024 by default and it takes an int as the input
4. Num-units, this is the number of units in the mlp and it set to 64 by default, it takes an int as the input.

1.7. Checkpointing Parameter
We only have 1 key checkpointing parameter that we need to consider:

1. Save-rate, the save rate is how often it saves the model after every time a set number of episodes are completed. By default, it is set to 100 episodes and it takes an int as the input.

1.8. Evaluation Parameters
Within the evaluation parameters we have 2 key parameters that we need to consider that show the learning process and benefit us with benchmark data:

1. Display, this is whether we want to display the actual learning environment or not. This is set to false however when we set it to true, it shows the current environment in matplotlib.
2. Benchmark-episodes, this is how many episodes are run for benchmarking, this is set to 1000 by default and takes an int as the input.
3. Benchmark-iters, this is the number of iterations run for benchmarking, this is set to 10,000 by default and takes an int as the input

C Setting up the Gazebo environment

Setup of the simulation environment

FIrst we need to indicate the ROS repository to the apt sources system.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
```

Adding the trusted keys for the repo.

```
curl -sSL
'http://keyserver.ubuntu.com/pks/lookup?op=get&search=0xC1CF6E31E6
BADE8868B172B4F42ED6FBAB17C654' | sudo apt-key add -
```

Update the package list.

```
sudo apt-get update
```

Installing ROS Melodic.

```
sudo apt-get install ros-melodic-desktop-full
```

Initialise the new installation.

```
sudo rosdep init
rosdep update
```

Set ROS parameters on shell startup.

```
echo "source /opt/ros/melodic/setup.bash" >> .bashrc
```

Perform changes now

```
source .bashrc
```

Install additional dependencies.

```
sudo apt-get install python-rosinstall python-rosinstall-generator
python-wstool build-essential
```

Create the ROS library root in the home directory.

```
mkdir -p ~/catkin_ws/src
```

Initialise workspace.

```
cd catkin_ws/src/
catkin_init_workspace
```

Gather the ROS libraries for the turtlebot3 simulator.

```
git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
git clone
https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
```

Go back to the catkin root directory.

```
cd ..
```

Compile new libraries.

```
catkin_make
```

Test the new ROS installation.

```
roscore
rosrun turtlesim turtlesim_node
```

Initialise additional parameters on shell startup.

```
echo 'source ~/catkin_ws/devel/setup.bash' >> .bashrc
```

Test the turlebot3 simulation.

```
export TURTLEBOT3_MODEL=waffle_pi
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

Running simulation of one robot in an empty world.

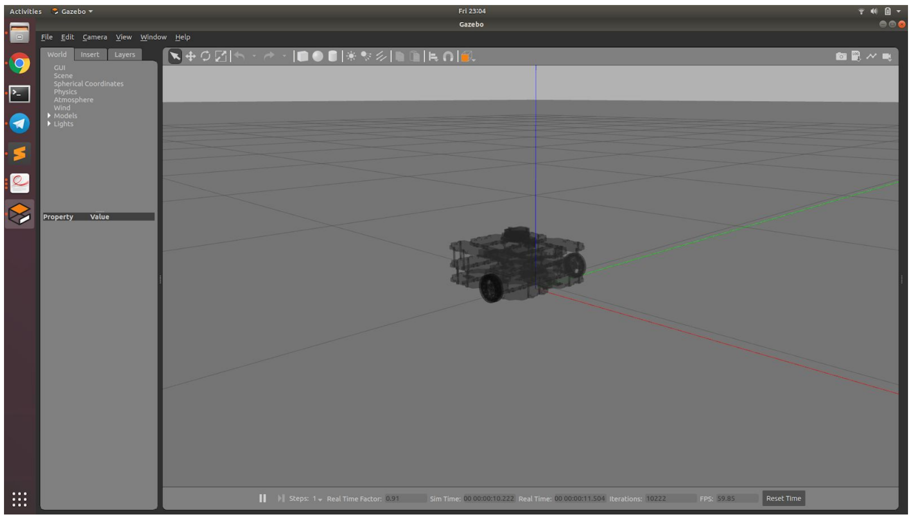


Figure 1: A Turtlebot3 simulation in Gazebo

Creating new ROS packages for a custom simulation

Go to the main `catkin_ws` entry point and place the file `madrl_ros.tar.gz` inside. Then extract it.

```
cd ~/catkin_ws/src
tar -xzvf madrl_ros.tar.gz
```

The main packages and their file structures are

1. `multi_robot/launch`
    a. `main.launch`: Starts the world, the Gazebo GUI and calls `robots.launch`.
    b. `one_obstacle.launch`: instructions to spawn one obstacle.
    c. `one_robot.launch`: instructions to spawn one TurtleBot.
    d. `one_target.launch`: instructions to spawn one landmark/target.
    e. `robots.launch`: spawns all entities (3 robots, 3 targets and 4 obstacles). Quantities can be changed through this file.
2. `obstacle_urdf/urdf`
    a. `obstacle.urdf`: definition for the physical properties of one obstacle object.
3. `target_urdf/urdf`
    a. `target.urdf`: definition for the physical properties of one target object.

Go back to the ROS library root and run the following to compile the new packages

```
cd ~/catkin_ws && catkin_make
```

The final simulation can now be launched with

```
export TURTLEBOT3_MODEL=waffle_pi
roslaunch multi_robot main.launch
```

Running the custom environment.

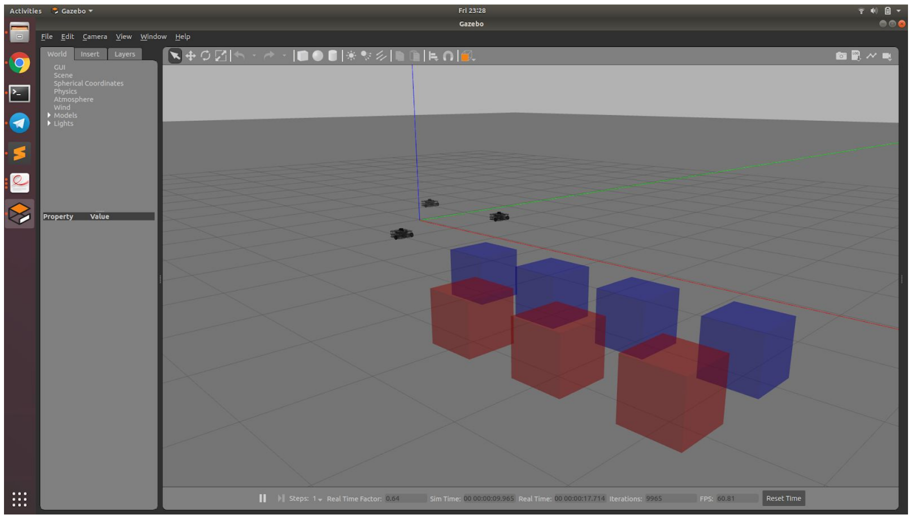


Figure 2: Multiple entities in the Gazebo world

Positions are arbitrary since they will be changed later through the MADRL trainer execution. The physical properties of these elements can be changed from the MADRL code. In order to do that, a server was used to listen to certain commands and redirect them to the ROS instances that controls the simulation.

Switching between simulation and physical testing

The file train.py was modified to support the execution of learned policies under the Gazebo environment or a real testing scenario. Positional data is dumped directly to stdout so it must be redirected to a file for storing purposes.

Additional parameters considered for the train.py entry point are:

`--gazebo-run` (runs the experiment under the Gazebo virtual environment)
`--real-run` (runs a physical test)

Other evaluation parameters are not recommended to be used in conjunction with these two. A typical run command looks like this:

```
python train.py --propagation-type DISC --observation-type
EXPLICIT --exp-name 4-obst_disc_explicit --num-obstacles 4
--world-size 10 --gazebo-run
```

Steps to run simulations in Gazebo

1. Follow the installation guide for the MADRL trainer (with `pyenv`). Use the latest `aasfiles.tar.gz` as the main source code.
2. Verify that under the `aasfiles/maddpg/data` there are experiments stored.
3. Follow the installation guide for ROS/Gazebo. Use the latest `madrl_ros.tar.gz` for the ROS packages.
4. In one terminal window, run `gaz_run.sh`. Make sure the experiment name and other parameters are consistent with the stored experiment.
5. In one terminal window, run `python ros_server_gz.py`. Make sure you are running Python 2.
6. In one terminal window, go to `aasfiles/maddpg/experiments` and run `bash run.sh`. This requires Python 3 but you should be fine if you follow the installation of the MADRL trainer with `pyenv`.
7. You should see robots moving.

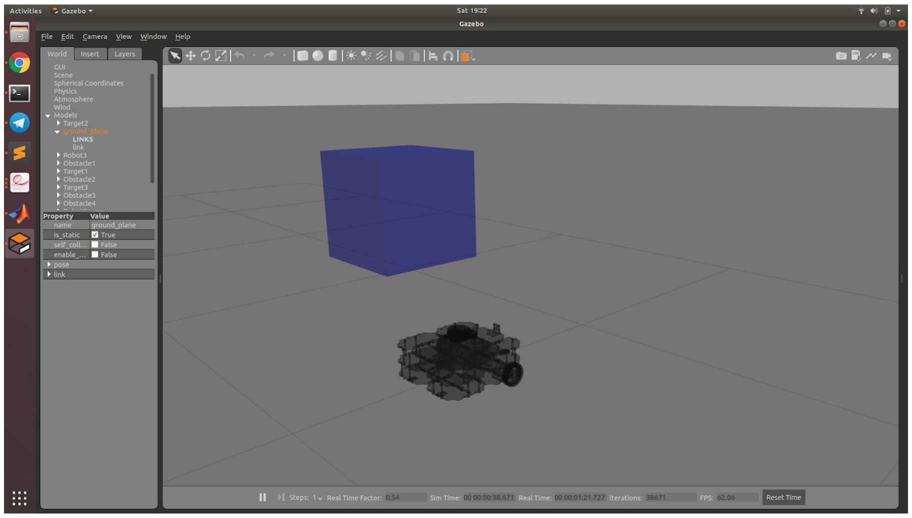


Figure 3: Simulation running in Gazebo

D Setup for physical experiments

1. Overview of Resources Used

Throughout the duration of the project we primarily used the TurtleBot3 Waffle Pi platform in conjunction with Marvelmind's precise indoor localization system to test the project outside of simulated conditions.

1.1. Hardware Used

The hardware used in the project to implement it outside of the simulated environment was the following:

- TurtleBot3 Waffle Pi Kit
- Intel NUC (On board computer)
- Power Bank (VINSIC Power Bank 30,000mAh model VSPB401)
- TP-Link TL722N wireless dongle
- Generic USB Extender
- Beacon HW v4.9-IMU (Marvelmind Robotics)
- Beacon Mini-RX (Marvelmind Robotics)
- Modem HW v4.9 (Marvelmind Robotics)

1.2. Software Used

The software used in the project to implement it outside of simulation was the following:

- VirtualBox, Version 6.0.14
- Linux Operating System, Bionic Beaver 18.04
- ROS Melodic, Version 18.04
- Marvelmind Dashboard, Version 6.160 Ultimate
- Marvelmind Beacon Inverse System Firmware, Version 2019_08_21_beacon_h259_sw6_160i_915-d9da8e7
- Marvelmind Mini RX Beacon Inverse System Firmware, Version 2019_10_08_beacon_mini_rx_sw6_170i_915
- OpenCR, Version 1.4.13
- Multimaster_fkie, Version 0.8.12

2. Hardware Layout

The physical components of the modified TurtleBot3 are broken up into four distinct sections that include each layer of the TurtleBot3 as well as the desktop/environment setup that was used to test the code base. This is important as the environment has a specific setup to ensure that the indoor localization system broadcasts accurate data to all nodes/agents.

2.1. TurtleBot3 Layer 1

The first layer of the modified TurtleBot3 consists of the following hardware components:

Original TurtleBot3 Parts

- 1x Li-Po Battery
- 2x Wheel
- 2x Tire
- 2x Dynamixel (XM430) motors
- 8x Waffle-Plate
- 2x Dynamixel to OpenCr Cable
- 12x Plate_Sipport_M3x35mm
- 6x Bracket
- 2x Ball Caster
- 32x Nut_M3
- 12x Rivet
- 44x PH_M3x8mm_K
- 8x PH_m2.5x12mm_K
- 8x PH_T2x6mm_K
- 8x PH_M2x4mm_K

Additional Parts

- Power Bank (VINSIC Power Bank 30,000mAh model VSPB401)

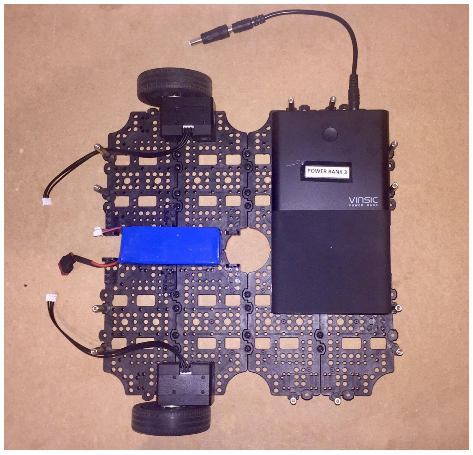


Figure 4: First layer of the modified TurtleBot3

The reasoning behind adding the much larger power bank (as shown above) in addition with the Li-Po battery is to ensure that both our on-board computer (intel NUC) and Dynamixel motors on the TurtleBot3 have sufficient power and the ability to run for longer durations. For Specific information on any of the parts listed above refer to the following:

TurtleBot3 - http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/

Power Bank – https://www.gearbest.com/power-banks/pp_1426586.html

2.2. TurtleBot3 Layer 2

The second layer of our modified TurtleBot3 consists of the following hardware components:

Original TurtleBot3 Parts

- 1x OpenCR1.0 Board
- 1x USB2LDS
- 1x Li-Po Battery Extension Cable
- 2x USB Cable
- 4x PCB Support
- 8x Waffle-Plate
- 10x Plate_Support_M3x45mm
- 32x NUT_M3
- 12x NUT_M2.5
- 4x Rivet
- 54x PH_M3-8mm_K
- 8x PH_M2.5x12mm_K
- 12x PH_M2.5x8mm_K

Additional Parts

- Intel NUC
- Wi-Fi Dongle (TP-LINK 722N)
- Generic 1M USB to Micro USB Cable

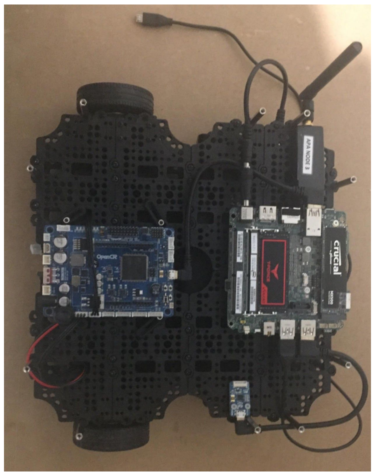


Figure 5: Second layer of the modified TurtleBot3

The reasoning behind swapping out the Raspberry Pi 3B+ with the Intel NUC is primarily the extra computing power that the intel NUC provides. Additionally, the Wi-Fi Dongle is to boost the signal strength on the intel NUC as well the extension cable is to allow us to connect to the Marvelmind beacon on the 3rd level. For specific information on any of the parts listed above refer to the following:

TurtleBot3 – http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/

Intel NUC – https://www.mouser.com/new/intel/intel-nuc-7-boards/

Wi-Fi Dongle – https://www.tp-link.com/au/home-networking/adapter/tl-wn722n/

USB to Micro USB cable - https://www.officeworks.com.au/shop/officeworks/p/1m-micro-usb-cable-cou2mb01

2.3. TurtleBot3 Layer 3

The third layer of our modified TurtleBot3 consists of the following hardware components:

Original TurtleBot3 Parts

- 1x 360 Laser Distance Sensor LDS-01
- 4x PCB Support
- 4x Spacer
- 8x Waffle-Plate
- 32x NUT_M3
- 12x NUT_M2.5
- 42x PH_M3x8mm_K
- 4x PH_M2.5x16mm_K

Additional Parts

- 1x Beacon Mini-RX

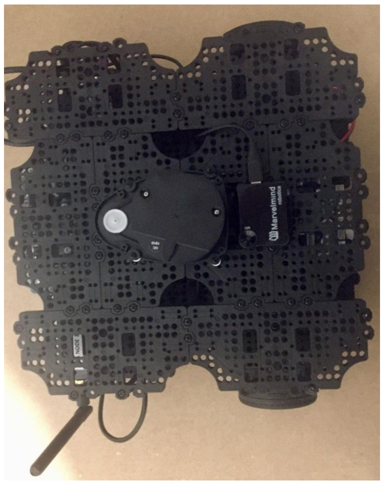


Figure 6: Third layer of the modified TurtleBot3

The reasoning behind having the Mini-RX beacon on the top level of the TurtleBot3 is to allow the beacon to have an uninterrupted line of sight to the stationary beacons. Furthermore, the reason we have chosen to go with the Mini-RX rather than the counterpart which is the Beacon HW v4.9-IMU, is the size comparison. As the Mini-RX doesn't interfere with the LDS sensor due to how small it is. For specific information on any of the parts listed above refer to the following:

TurtleBot3 - http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/

Beacon Mini-RX - https://marvelmind.com/product/mini-rx/

Beacon HW v4.9-IMU - https://marvelmind.com/product/beacon-hw-v4-9-imu/

2.4. Desktop/Environment Setup

Due to the Marvelmind real time positioning system in place, we require that the environment the robots are operating in be set up specifically for the real time positioning system to ensure that the robots are receiving timely and accurate data. Therefore, we had 4 stationary beacons placed 2 metres high and 5 metres apart in a square formation.

In conjunction with the stationary beacons we had the Marvelmind Modem plugged into a laptop placed at least 2 metres away to ensure that the Modems frequency does not interfere with the stationary beacons.

As a result, the desktop/environment setup has the following **minimum** hardware/software requirements:

Hardware

- Marvelmind Modem HW v4.9
- Beacon HW v4.9-IMU
- 1M USB to Micro USB Cable

Software

- Marvelmind Dashboard
- Openssh-server

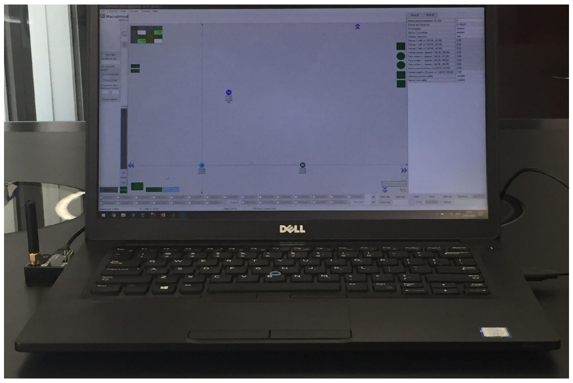


Figure 7: Marvelmind desktop setup

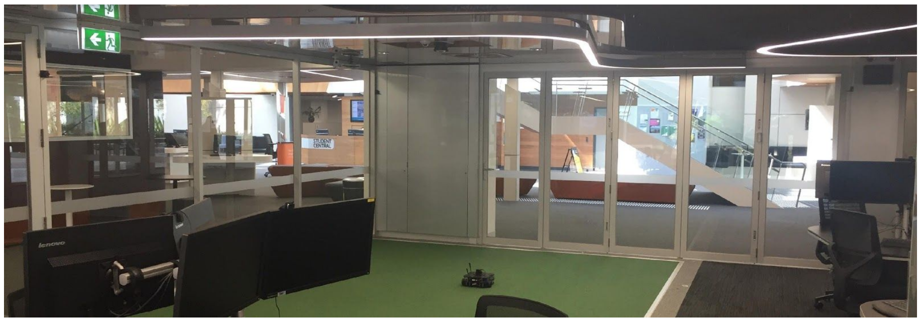


Figure 8: Physical testing environment

The reasoning behind having the stationary beacons roughly 5 metres apart in the square formation is that each beacon is one corner of the operating perimeter as shown above where you can see 3 on the beacons on the wall up against the glass.

2.5. Modified TurtleBot3 Layer 1 To Layer 2 Interconnections

The main interconnections between Layer 1 and Layer 2 on the modified TurtleBot3 is the connection between the:

1. Dynamixel motors and OpenCR board
2. Li-Po Battery and OpenCR board
3. Power bank and Intel NUC

Interconnections 1 and 2 can be seen within the TurtleBot3 Waffle Pi quick start manual found here:

http://emanual.robotis.com/docs/en/platform/turtlebot3/hardware_setup/

Interconnection 3 can be seen below:

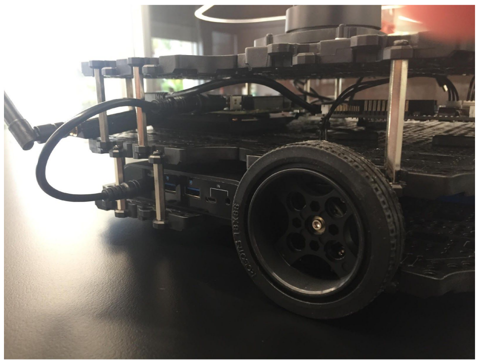


Figure 9: Interconnection 3 Between the Battery Pack and Intel NUC

2.6. Modified TurtleBot3 Layer 2 To Layer 3 Interconnections

The only interconnections between Layer 2 and Layer 3 on the modified TurtleBot3 are the following:

1. USB2LDS and 360 Laser Distance Sensor LDS-01
2. Intel NUC and Marvelmind

Interconnection 1 can be found above in the link provided whereas interconnection 2 can be seen below (it needs to be noted that when not using the LDS sensor, the hedgehog should be placed in front of the sensor):

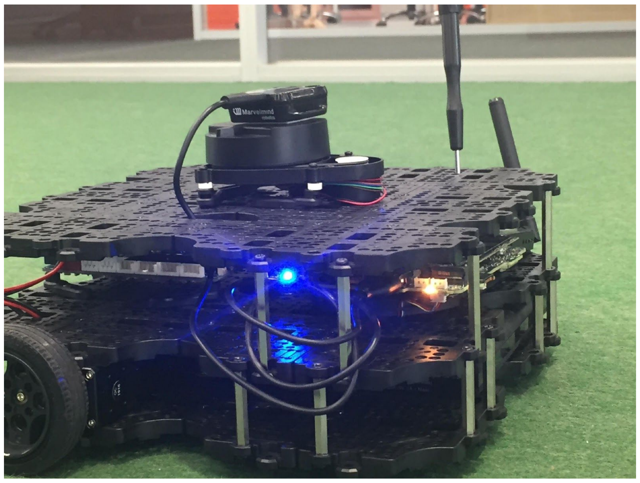


Figure 10: Interconnection 2 Between the Marvelmind Hedgehog and Intel NUC

3. Understanding/Checking Key Components

In order to recreate the project, we first need to understand and check key components in the project. This section is broken up into distinct categories to ensure key major components in the project are working and setup correctly.

3.1. Setting Up the Marvelmind System

In order to recreate the project, you will need the following physical hardware to allow the agents to broadcast their exact positions to each other (excluding the modem):

- At least 2 Beacon HW v4.9-IMU, Stationary Beacons (to measure Z values you will need at least 3 Beacons with different frequencies)
- At least 1 Beacon Mini-RX, Mobile Beacon (Placed on the TurtleBot3)

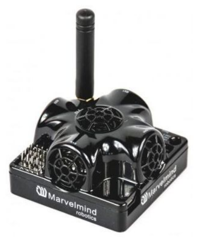


Figure 11: Beacon HW v4.9 IMU

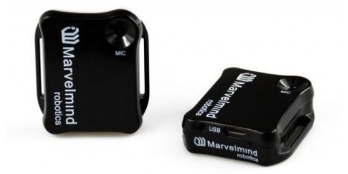


Figure 12: Beacon Mini-RX

3.1.1. Installing the Marvelmind Dashboard

In order to run the real time positioning system, the dashboard has to be up and running with the modem plugged into the computer. This is particularly important as the system won't work if the dashboard isn't up and running due to the Inverse Architecture that we used.

The Marvelmind dashboard comes in the following operating systems:

- Windows
    - 32 Bit Operating System
    - 64 Bit Operating System
- Linux
    - Arm (for less powerful SOC's like Raspberry Pi's)
    - X86

They can all be installed from the following link - https://marvelmind.com/download/

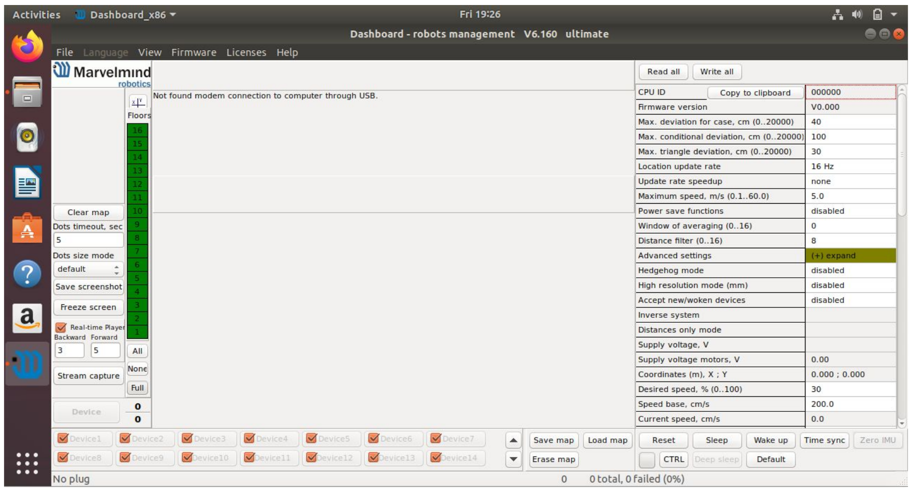
The exact software used in the project can be downloaded and installed from the following repository - https://github.com/JTMurley/MarvelMindFirmWare

Both Windows and Linux dashboards have been used and tested and they both work however the Windows Marvelmind dashboard provides more functionality and flexibility with firmware versions as it is the targeted operating system made by Marvelmind Robotics.

Note that if you cannot find the Beacon's via USB cable to upload the firmware for the first time and if you are using a Linux Dashboard try the windows dashboard and see if they appear on that dashboard as the Linux Dashboard has issues recognising later firmware versions as seen by our own trial and error testing.

In order for the Marvelmind Localisation System to work correctly, the modem needs to be plugged into a computer with the dashboard running in order to have the Mini RX beacons transmit their current position in ROS with the Inverse Architecture.

Once you have downloaded the dashboard and installed it, the dashboard should look like the following if you don't have the modem plugged in:



Not found modem connection to computer through USB.

CPU ID	000000
Firmware version	V0.000
Max. deviation for case, cm (0..20000)	40
Max. conditional deviation, cm (0..20000)	100
Max. triangle deviation, cm (0..20000)	30
Location update rate	16 Hz
Update rate speedup	none
Maximum speed, m/s (0.1..60.0)	5.0
Power save functions	disabled
Window of averaging (0..16)	0
Distance filter (0..16)	8
Advanced settings	(+) expand
Hedgehog mode	disabled
High resolution mode (mm)	disabled
Accept new/woken devices	disabled
Inverse system	
Distances only mode	
Supply voltage, V	
Supply voltage motors, V	0.00
Coordinates (m), X ; Y	0.000 ; 0.000
Desired speed, % (0..100)	30
Speed base, cm/s	200.0
Current speed, cm/s	0.0

0 0 total, 0 failed (0%)

Figure 13: Marvelmind dashboard

If you are having issues with the dashboard or want more information on the dashboard, refer to the following system manual - https://marvelmind.com/pics/marvelmind_navigation_system_manual.pdf

3.1.2. Setting Up the Beacons

Before manually placing the beacons on the wall, we need to ensure that they have the latest firmware uploaded onto them. In order to do this, we need to turn them on and connect them up to the dashboard to check the firmware version as well as to know/set their unique node ID (note they have to be in working mode to update firmware).

Note that the DIP (manual electronic) switch for the Mini RX is located inside the shell, the easiest way to remove the shell on the Mini RX Beacon is by pulling up on the USB port gap.

Refer to the following video for a full tutorial for the aforementioned -https://www.youtube.com/watch?v=sOce7B2_6Sk

Now that both the Stationary and Mobile beacons are up to date or have the same firmware versions, we need to flip the DIP switch into working mode as shown below:

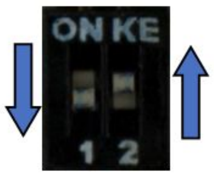


Figure 14: Working DIP Switch Mode Configuration

Note that the DIP switch is in working mode when you see LED 2 blinking blue or bright green. If LED 2 is not blinking the beacon may be dead and need to be charged, leave it to charge for 2 or more hours and then try again. If the problem persists refer to the system manual - https://marvelmind.com/pics/marvelmind_navigation_system_manual.pdf

When setting up the beacons on the wall you need to ensure that they are at least a meter away from the modem plugged into the laptop as if they are within a meter of the modem they won’t work as optimally as they could due to the strong frequency that the modem emits.

If you only have two stationary beacons you need to note that the Z value won’t be calculated and as a result the X and Y will be slightly off (roughly 5-15 CM from testing) due to the height that the beacons are placed at (if the height of each beacon is not manually entered). However, you need to note that the X and Y are only off with respect to the map and not actual distance between nodes/agents.

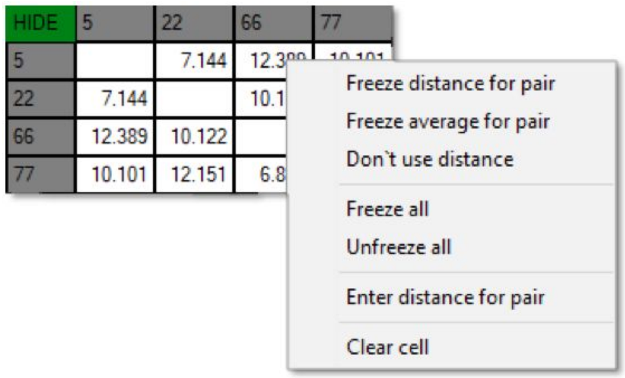
A video displaying how to best place the stationary beacons can be seen from the following video - https://www.youtube.com/watch?v=WY0HkLzmjys

Note that if you don’t hear a buzzing sound from the stationary beacons on the wall, they might not emitting or emitting at a low HZ rate such as 1 or 2 and as a result the system might not transmit any data or transmit data at a low update rate which will affect the performance of the modified TurtleBot3’s.

3.1.3. Configuring the Dashboard

Due to the inverse architecture that we are using with the Marvelmind positioning system, we have to manually input the distances between each beacon. To do this we simply need to enter the distances between beacons into the table of distances. We can do this in the following steps:

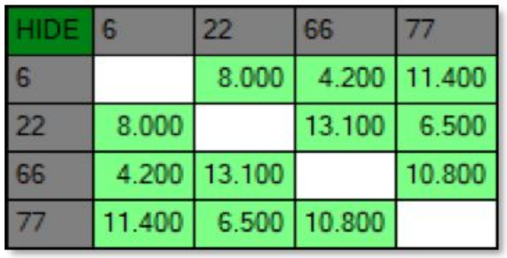
1. Open the dashboard. You should see the table of distances in the submap you have selected. If you don't, simply use the drop-down menu at the top of the dashboard and then select table of distances
2. Right click on the cell you want to enter the distance. An additional menu will open up. In this additional menu you can control the table of distances. Select "Enter distance for pair" to enter the distance between that pair of beacons as shown below:



HIDE	5	22	66	77
5		7.144	12.3[illegible]	[illegible]
22	7.144		10.1	
66	12.389	10.122		
77	10.101	12.151	6.8	

Figure 15: Manually Entering Distances for Beacons

3. Now, enter the measured (measure it with a laser distance meter or equivalent) distance value. Note that values won't change until you unfreeze or clear cells. Even if beacons get moved, the distance will stay the same. Therefore, as a result you need to be careful with frozen cells and beacons as a small mistake can cause a huge impact on your tracking.
4. Repeat for all cells, as shown below:



HIDE	6	22	66	77
6		8.000	4.200	11.400
22	8.000		13.100	6.500
66	4.200	13.100		10.800
77	11.400	6.500	10.800	

Figure 16: All Distances Entered in the Distance Table

5. Now freeze the submap and by doing so the beacons will stop measuring relative distance and now measure the distance from the mobile beacons. Note that you also need to freeze the main map and not just the submap (click on the modem button in the bottom left of the dashboard to bring up the main map).

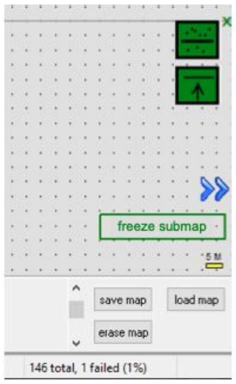


Figure 17: Freezing the submap

6. Turn on and wake up the mobile beacons (ensure the DIP switch is in working mode for all beacons).
7. Now you should see your mobile beacons appear on the dashboards map. If you see on the devices' panel in the dashboard that the beacon is coloured orange, it means that there are some differences in some of the settings between beacons. For example, some sensors may be off or some ultrasonic or radio settings may be different. You can change the settings for sensors manually by clicking on the panel on the upper right corner of the dashboard to change the cells from grey to green to turn on the sensor. The following hedgehog colours represent:

   Blue - normal mode and confident tracking

   Orange - system provides the best location data possible, but confidence is lower than blue

   Transparent Blue - lost radio packets

   Transparent Orange – weak ultrasonic coverage

8. Furthermore, to ensure that you have the most accurate tracking it is heavily recommended that you directly enter the height at which the beacon is placed at. Directions about how to enter the height as well as other useful settings available on the beacon can be found here - https://www.youtube.com/watch?v=sm0R5QPLWoE

As well because the Stationary beacons v4.9 have to have different ultrasonic frequencies, the following colours represent such frequencies:

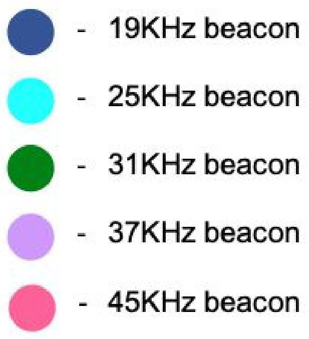


Figure 18: Stationary beacons Frequency Colour Representation

Now you should be able to move around with the mobile beacons and see them being tracked on the dashboard. However, if you are unable to do this refer to section 5.1 (pg. 69) in the following system manual - https://marvelmind.com/pics/marvelmind_navigation_system_manual.pdf

3.1.4. Receiving Data in ROS

Before receiving data in ROS, you will need the following software installed and set up:

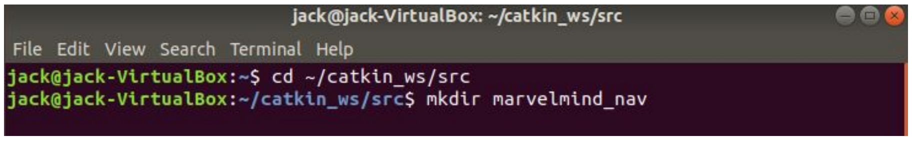
- A distribution of ROS (we used ROS Melodic)
- Catkin workspace setup

If you don't have a distribution of ROS or your catkin workspace setup you can do so by following these links:

- ROS (Melodic), http://wiki.ros.org/melodic/Installation
- Catkin Workspace, http://wiki.ros.org/catkin/Tutorials/create_a_workspace

Once you have the following software installed follow these steps:

1. Create a directory to store our test code, as shown below:


jack@jack-VirtualBox: ~/catkin_ws/src
File Edit View Search Terminal Help

```
jack@jack-VirtualBox:~$ cd ~/catkin_ws/src
jack@jack-VirtualBox:~/catkin_ws/src$ mkdir marvelmind_nav
```

Figure 19: Creating a Marvelmind Directory

2. Download the marvelmind_nav folder from the following repository - https://github.com/JTMurley/MarvelmindROSCode/tree/master/ros
3. Copy the downloaded folder into the directory we just made, it should look like the following:

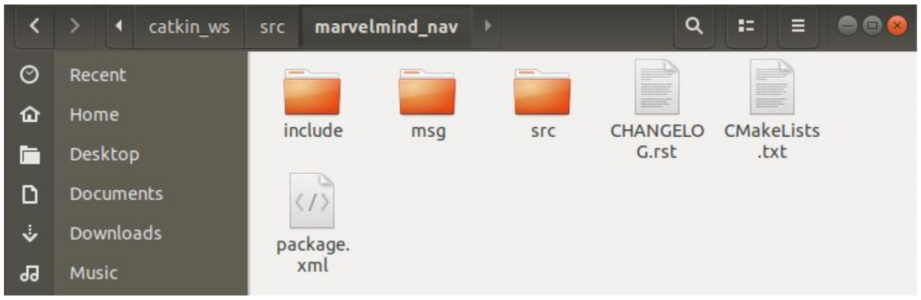
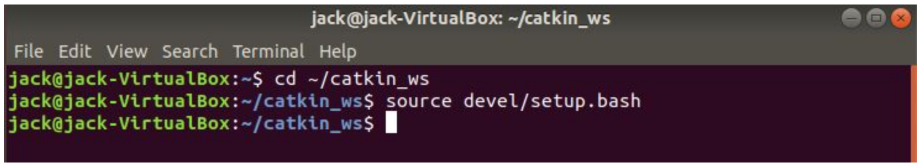


Figure 20: Copying Sample Code into the marvelmind_nav Directory

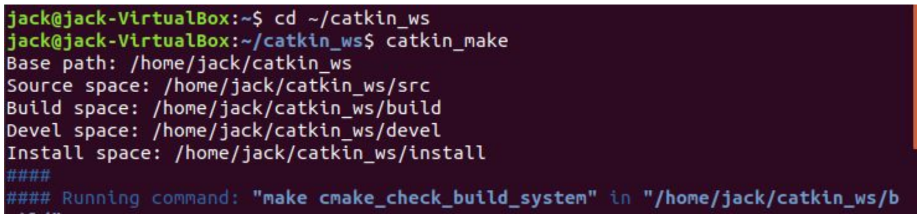
4. Now we need to source the setup.bash as shown below:


jack@jack-VirtualBox: ~/catkin_ws
File Edit View Search Terminal Help

```
jack@jack-VirtualBox:~$ cd ~/catkin_ws
jack@jack-VirtualBox:~/catkin_ws$ source devel/setup.bash
jack@jack-VirtualBox:~/catkin_ws$ 
```

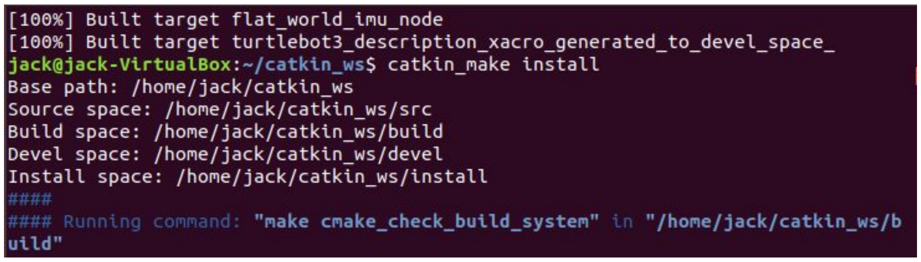
Figure 21: Sourcing the setup.bash

5. After sourcing the setup.bash we can now build and install the Marvelmind ROS packages as shown below:



```
jack@jack-VirtualBox:~$ cd ~/catkin_ws
jack@jack-VirtualBox:~/catkin_ws$ catkin_make
Base path: /home/jack/catkin_ws
Source space: /home/jack/catkin_ws/src
Build space: /home/jack/catkin_ws/build
Devel space: /home/jack/catkin_ws/devel
Install space: /home/jack/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/jack/catkin_ws/b
```

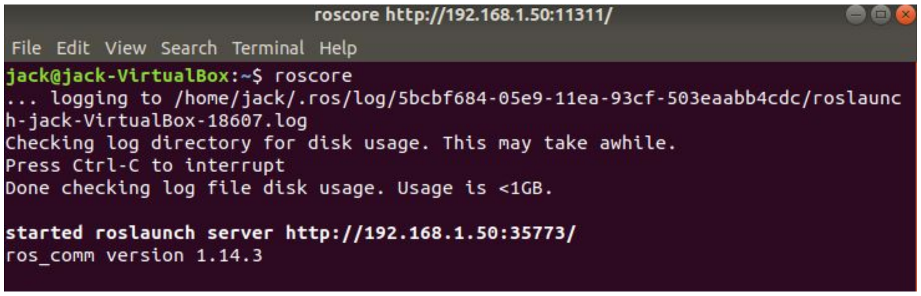
Figure 22: Building the marvelmind_nav Package with catkin_make



```
[100%] Built target flat_world_imu_node
[100%] Built target turtlebot3_description_xacro_generated_to_devel_space_
jack@jack-VirtualBox:~/catkin_ws$ catkin_make install
Base path: /home/jack/catkin_ws
Source space: /home/jack/catkin_ws/src
Build space: /home/jack/catkin_ws/build
Devel space: /home/jack/catkin_ws/devel
Install space: /home/jack/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/jack/catkin_ws/b
uild"
```

Figure 23: Making the marvelmind_nav Package with catkin_make install

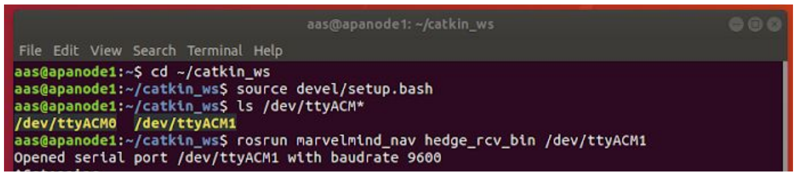
6. Once the Marvelmind ROS package has been successfully built and installed, connect the Mini RX beacon to your ROS machine via USB and run roscore as shown below:



```
roscore http://192.168.1.50:11311/
File Edit View Search Terminal Help
jack@jack-VirtualBox:~$ roscore
... logging to /home/jack/.ros/log/5bcbf684-05e9-11ea-93cf-503eaabb4cdc/roslaunc
h-jack-VirtualBox-18607.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.1.50:35773/
ros_comm version 1.14.3
```

Figure 24: Starting roscore

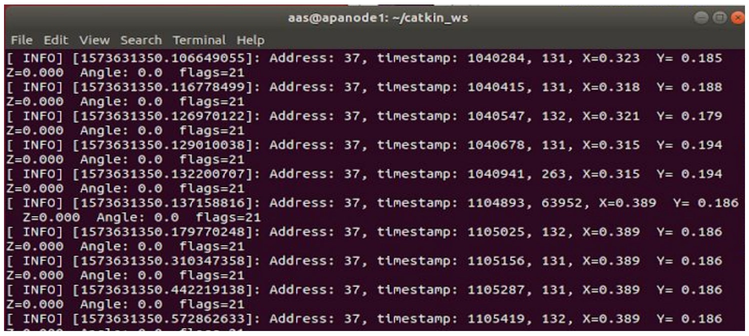
7. Once roscore is up and successfully running, we now need to run the node 'hedge_rcv_bin' to receive data from the hedgehog. This has to be run on the same machine that the mini RX beacon is plugged into. To do this, follow the following steps below as shown in the screenshot (note we do ls/dev/ttyACM* to find the name of the virtual serial port detected by the previous command, if the port is ttyACM0, this parameter can be skipped):



```
aas@apanode1: ~/catkin_ws
File Edit View Search Terminal Help
aas@apanode1:~$ cd ~/catkin_ws
aas@apanode1:~/catkin_ws$ source devel/setup.bash
aas@apanode1:~/catkin_ws$ ls /dev/ttyACM*
/dev/ttyACM0  /dev/ttyACM1
aas@apanode1:~/catkin_ws$ rosrun marvelmind_nav hedge_rcv_bin /dev/ttyACM1
Opened serial port /dev/ttyACM1 with baudrate 9600
```

Figure 25: Running hedge_rcv_bin Node

The output from the hedge_rcv_bin code is in the brackets a ROS timestamp, followed by a hedgehog timestamp in milliseconds, time (in milliseconds) between position samples, coordinates X,Y,Z in metres and bytes of flags.

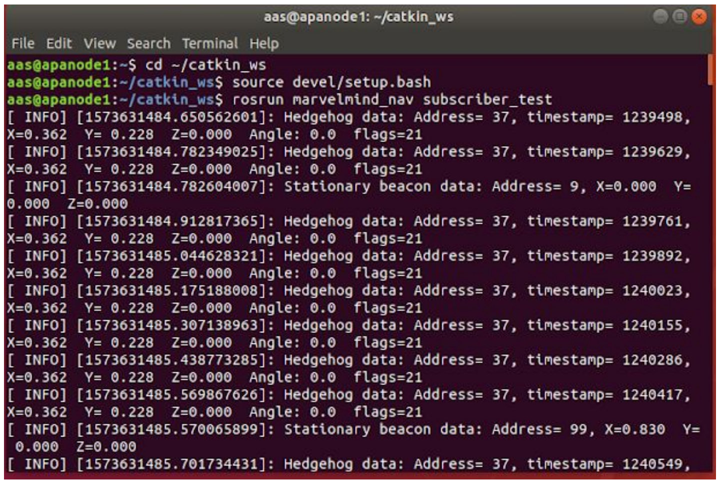

aas@apanode1: ~/catkin_ws

```
[ INFO] [1573631350.106649055]: Address: 37, timestamp: 1040284, 131, X=0.323  Y= 0.185
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.116778499]: Address: 37, timestamp: 1040415, 131, X=0.318  Y= 0.188
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.126970122]: Address: 37, timestamp: 1040547, 132, X=0.321  Y= 0.179
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.129010038]: Address: 37, timestamp: 1040678, 131, X=0.315  Y= 0.194
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.132200707]: Address: 37, timestamp: 1040941, 263, X=0.315  Y= 0.194
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.137158816]: Address: 37, timestamp: 1104893, 63952, X=0.389  Y= 0.186
  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.179770248]: Address: 37, timestamp: 1105025, 132, X=0.389  Y= 0.186
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.310347358]: Address: 37, timestamp: 1105156, 131, X=0.389  Y= 0.186
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.442219138]: Address: 37, timestamp: 1105287, 131, X=0.389  Y= 0.186
Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631350.572862633]: Address: 37, timestamp: 1105419, 132, X=0.389  Y= 0.186
```

Figure 26: Output From the hedge_rcv_bin Node

The node ‘hedge_rcv_bin’ also works as a ROS publisher, it sends the message with location data to the topic names ‘/hedge_pos’. However, the github folder you downloaded contains another node ‘subscriber_test’, which works as a ROS subscriber which receives data from the topic. This node can be used for test purposes and as a basis for user software.

8. To run this subscriber node, use the following commands as shown in the screenshot below:


aas@apanode1: ~/catkin_ws

```
aas@apanode1:~$ cd ~/catkin_ws
aas@apanode1:~/catkin_ws$ source devel/setup.bash
aas@apanode1:~/catkin_ws$ rosrun marvelmind_nav subscriber_test
[ INFO] [1573631484.650562601]: Hedgehog data: Address= 37, timestamp= 1239498,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631484.782349025]: Hedgehog data: Address= 37, timestamp= 1239629,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631484.782604007]: Stationary beacon data: Address= 9, X=0.000  Y=
0.000  Z=0.000
[ INFO] [1573631484.912817365]: Hedgehog data: Address= 37, timestamp= 1239761,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631485.044628321]: Hedgehog data: Address= 37, timestamp= 1239892,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631485.175188008]: Hedgehog data: Address= 37, timestamp= 1240023,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631485.307138963]: Hedgehog data: Address= 37, timestamp= 1240155,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631485.438773285]: Hedgehog data: Address= 37, timestamp= 1240286,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631485.569867626]: Hedgehog data: Address= 37, timestamp= 1240417,
X=0.362  Y= 0.228  Z=0.000  Angle: 0.0  flags=21
[ INFO] [1573631485.570065899]: Stationary beacon data: Address= 99, X=0.830  Y=
 0.000  Z=0.000
[ INFO] [1573631485.701734431]: Hedgehog data: Address= 37, timestamp= 1240549,
```

Figure 27: Subscriber Node Output

This node also works as a publisher and sends data to the topic ‘visualization_marker’. This allows us to view the position in the standard ROS software ‘rviz’.

3.2. Setting Up/Checking the Modified TurtleBot3

This section specifically details how to test the motors on the modified TurtleBot3 as well as check its connection to the master node without any third-party applications.

3.2.1. Checking the Motors

Before running the TurtleBot3 we need to ensure that all baseline hardware and software is up and running correctly. Follow the following guide if you still need to set up the baseline hardware and software on your TurtleBot3 -
http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

Before running any code, we need to ensure that the motors are working, and the lithium battery is charged. To do this we will press SW1 and then SW2 on the OpenCR board to check both the motors. SW1 and SW2 can be found as shown below:

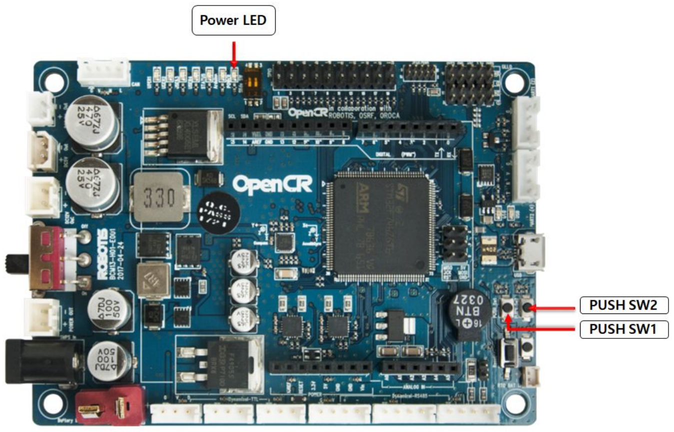


Figure 28: OpenCR Board SW1 and SW2 Locations

Pressing SW1 and SW2 will do the following:

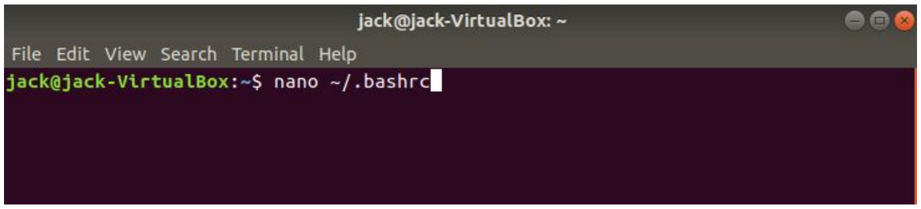
SW1 – Pressing and holding SW1 for a few seconds will command the robot to move roughly 30 centimetres or 12 inches forward

SW2 – Pressing and holding SW2 for a few seconds will command the robot to rotate 180 degrees in place

Note, do not run these tests while the OpenCR board is connected to the intel NUC or if it is connected, ensure that the intel NUC is off. This is to ensure that the board is powered primarily from the lithium battery and not the intel NUC through the battery pack on the bottom level.

3.2.2. Checking the Connection to the Master Node

Once we know that the motors are working, we now need to check and see if all the nodes (TurtleBot3's) have a connection to their own master node. To setup this connection we need to modify the bashrc as shown below:



```
jack@jack-VirtualBox:~$ nano ~/.bashrc
```

Figure 29: Accessing the bashrc

Once in the bashrc press alt+/ to get to the end line of the file.

Once at the end line of the file, you will **want to**:

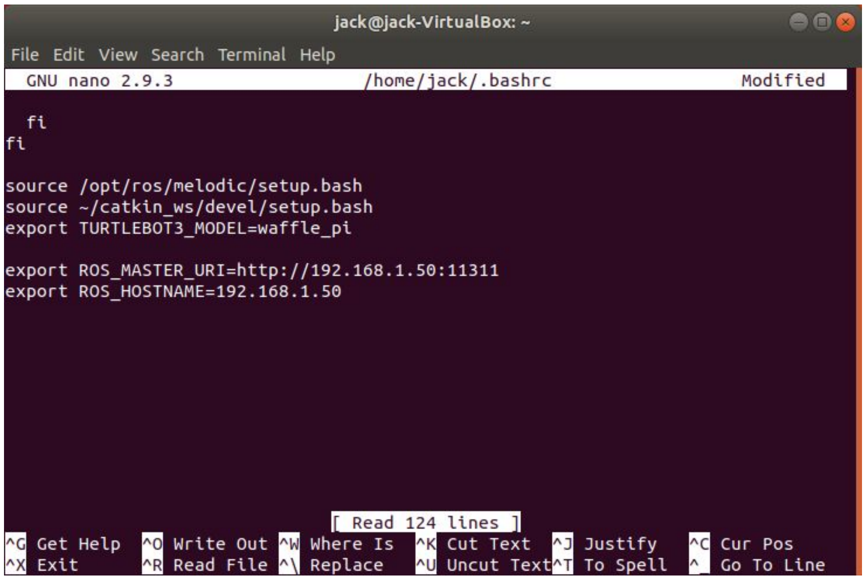
- Source your catkin_ws setup.bash file
- Export the TurtleBot model you are using (TurtleBot3 waffle_pi in our case)

The reason you will want to do this, is you won't have to source the setup file every time you want to run the code and by sourcing the TurtleBot3 model you won't need to declare the model when running the code.

Furthermore, you will **need to**:

- Export the ROS Master URI
- Export the ROS Hostname

This is to ensure the node will have a connection to the master node, a sample declaration of what it should look like is shown below:

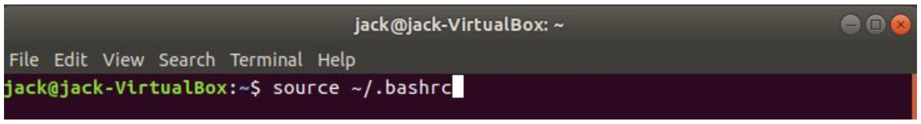


```
  GNU nano 2.9.3                /home/jack/.bashrc                  Modified

  fi
fi

source /opt/ros/melodic/setup.bash
source ~/catkin_ws/devel/setup.bash
export TURTLEBOT3_MODEL=waffle_pi

export ROS_MASTER_URI=http://192.168.1.50:11311
export ROS_HOSTNAME=192.168.1.50

                              [ Read 124 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

Figure 30: Bashrc settings

The ROS Master URI and HostName should be the TurtleBots own IP address with the addition of :113111 at the end of the ROS Master URI. The IP address of the TurtleBot can be found out by typing ifconfig into the terminal. For more information refer to the following guide - http://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/#pc-setup

Once you have saved and properly setup the bashrc don't forget to source the bash.rc as shown below:



```
jack@jack-VirtualBox:~$ source ~/.bashrc
```

Figure 31: Sourcing the bashrc

Note, you will need to repeat this step on all robots as each robot runs it owns roscore in a multi master network.

Once the bashrc is setup correctly on all of the robots, attempt to run roscore on each of the TurtleBots to ensure that they can host their own respective nodes. If any of the TurtleBots are unable to do so, check the bashrc and ensure that the IP addresses are entered correctly as well try to ping their own IP address to see if it is reachable.

3.3. Running Multiple Nodes

This section specifically details how to run multiple of the modified TurtleBot3's and have them run as their own host through the Multimaster_fkie package.

3.3.1. Understanding the Multimaster_fkie Package

Multimaster_fkie is a package which allows a set of nodes to establish and manage a multimaster network with little or no configuration.

The package is comprised of the following nodes:

- Master_discovery node
- Master_sync node
- Node_manger node

The package works by the master_discovery node connecting to the ROS-Master, which gets changes by polling and publishing changes (multicast or/ and unicast) over the network. The received changes of the remote ROS-Master are published to the local ROS topics. The master_sync node then connects to the discovered master_discovery nodes and the master_discovery nodes will request the actual ROS state and register the remote topics/services.

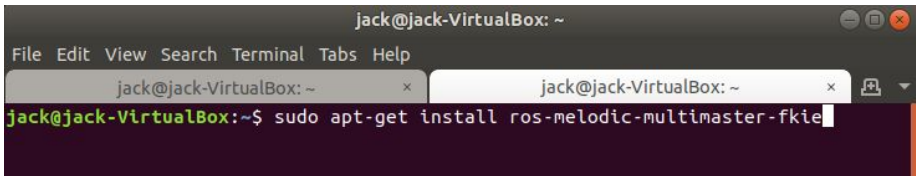
The node_manger then simplifies launching and managing the ROS multi-master system, as well it offers options for managing nodes, topics, services, parameters and launch files.

For specifics refer to the Multimaster_fkie ROS page and Multi-master ROS system documentation as shown below:

- ROS page - http://wiki.ros.org/multimaster_fkie
- Multi-master ROS System document - http://www.iri.upc.edu/files/scidoc/1607-Multi-master-ROS-systems.pdf

3.3.2. Using the Multimaster_fkie Package

To install the package run the following command:

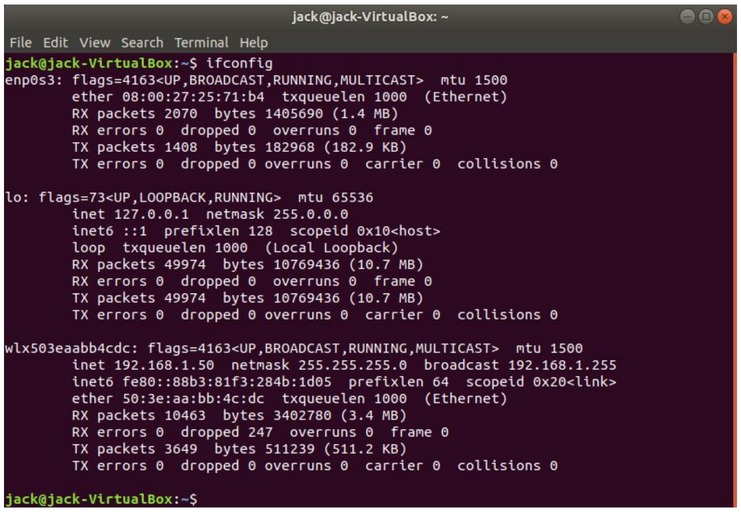


```
jack@jack-VirtualBox:~$ sudo apt-get install ros-melodic-multimaster-fkie
```

Figure 32:Installing the Multimaster-fkie Package

Furthermore, ensure that you are on a network that allows you to multicast or unicast. This can be checked by doing the following:

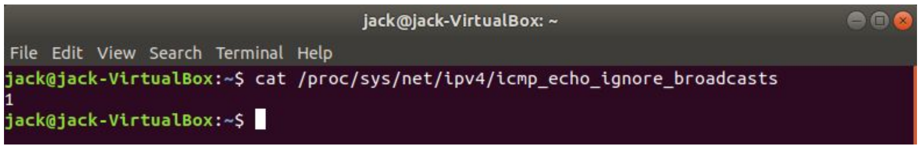
Ifconfig, will display if multicast is enabled on your network



```
jack@jack-VirtualBox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        ether 08:00:27:25:71:b4  txqueuelen 1000  (Ethernet)
        RX packets 2070  bytes 1405690 (1.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1408  bytes 182968 (182.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 49974  bytes 10769436 (10.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 49974  bytes 10769436 (10.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlx503eaabb4cdc: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.50  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::88b3:81f3:284b:1d05  prefixlen 64  scopeid 0x20<link>
        ether 50:3e:aa:bb:4c:dc  txqueuelen 1000  (Ethernet)
        RX packets 10463  bytes 3402780 (3.4 MB)
        RX errors 0  dropped 247  overruns 0  frame 0
        TX packets 3649  bytes 511239 (511.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

jack@jack-VirtualBox:~$
```

Figure 33: Checking to see if Multicast is Enabled

Echo ipv4 broadcasts, by echoing the broadcast you can see if it is enabled. Note that if the network you are on supports multicast you most likely won't have to enable it. It can be checked by using the following command:



```
jack@jack-VirtualBox:~$ cat /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
1
jack@jack-VirtualBox:~$ 
```

Figure 34: Echo ipv4 Broadcast

If the command returns 0, it means that multicast is enabled and if it returns 1, multicast is not enabled. To temporarily enable multicast, you can use the following command:

$ sudo sh -c "echo 0 >/proc/sys/net/ipv4/icmp_echo_ignore_broadcasts"

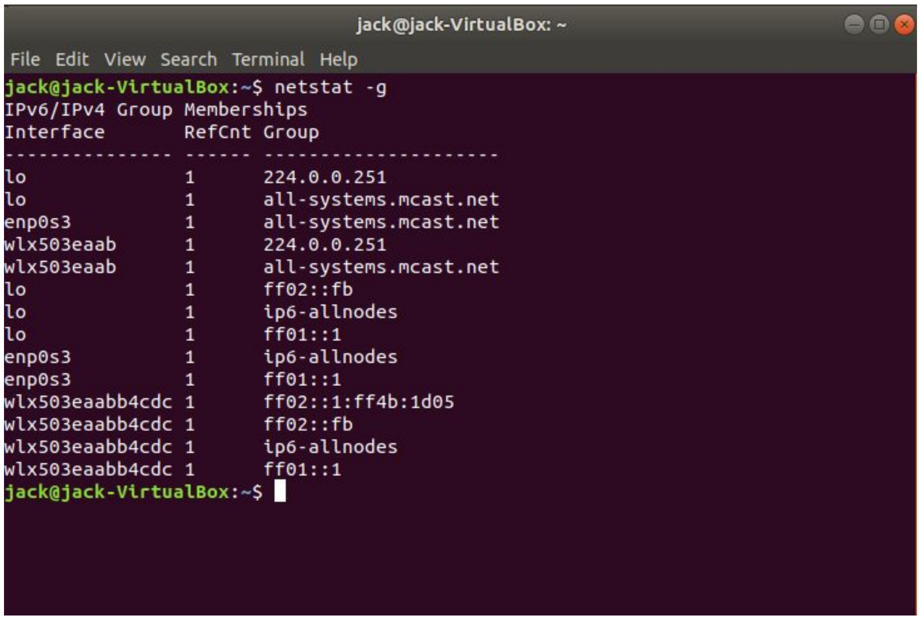
To permanently enable multicast, edit your sysctl.conf file and add the following line or if it exists change the default value to:

$ net.ipv4.icmp_echo_ignore_broadcasts=0

In order for the changes to take effect, execute the following command:

$ sudo service procps restart

Once this is all done, we now need to check which multicast groups are already defined in the computer/network, this can be done by the following command:

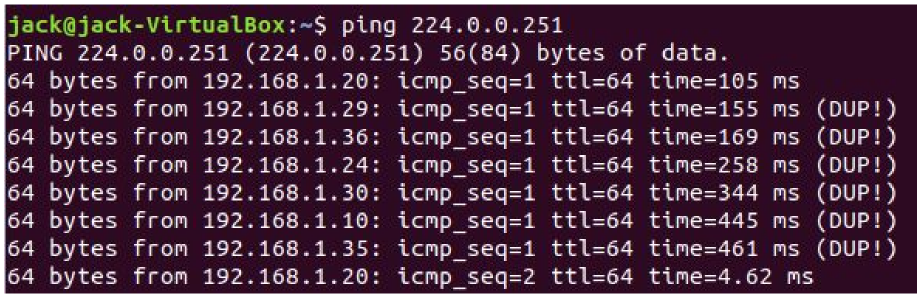


```
jack@jack-VirtualBox:~$ netstat -g
IPv6/IPv4 Group Memberships
Interface       RefCnt Group
--------------- ------ ---------------------
lo              1      224.0.0.251
lo              1      all-systems.mcast.net
enp0s3          1      all-systems.mcast.net
wlx503eaab      1      224.0.0.251
wlx503eaab      1      all-systems.mcast.net
lo              1      ff02::fb
lo              1      ip6-allnodes
lo              1      ff01::1
enp0s3          1      ip6-allnodes
enp0s3          1      ff01::1
wlx503eaabb4cdc 1      ff02::1:ff4b:1d05
wlx503eaabb4cdc 1      ff02::fb
wlx503eaabb4cdc 1      ip6-allnodes
wlx503eaabb4cdc 1      ff01::1
jack@jack-VirtualBox:~$ 
```

Figure 35: Viewing Multicast Groups with netstat -g

We can see that we have a multicast on the IP address – 224.0.0.251 and that is the address that we will use in the following steps.

Before we continue, we will ping that IP address to make sure that it is reachable with minimal delay, as shown below:



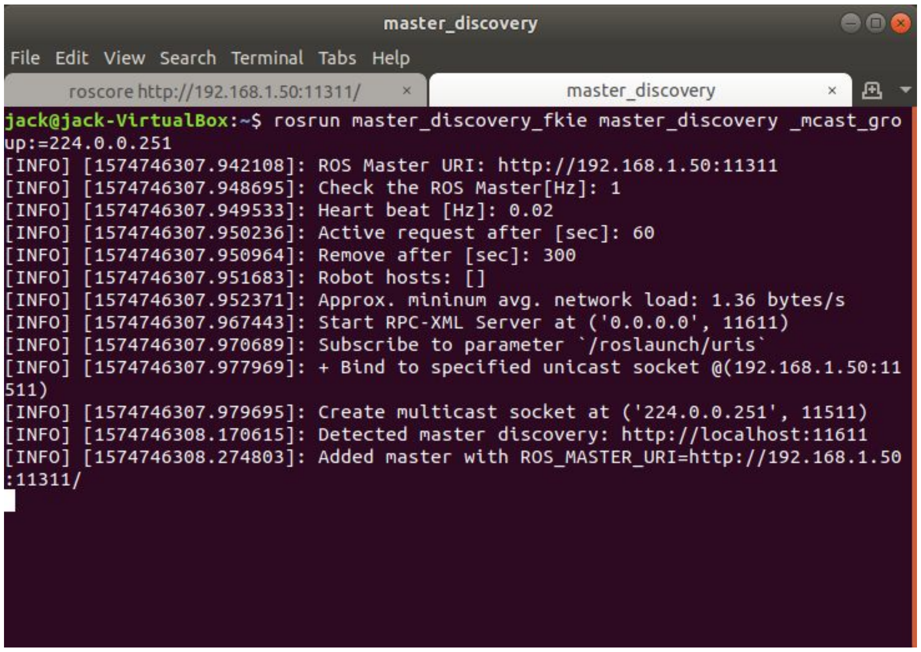
```
jack@jack-VirtualBox:~$ ping 224.0.0.251
PING 224.0.0.251 (224.0.0.251) 56(84) bytes of data.
64 bytes from 192.168.1.20: icmp_seq=1 ttl=64 time=105 ms
64 bytes from 192.168.1.29: icmp_seq=1 ttl=64 time=155 ms (DUP!)
64 bytes from 192.168.1.36: icmp_seq=1 ttl=64 time=169 ms (DUP!)
64 bytes from 192.168.1.24: icmp_seq=1 ttl=64 time=258 ms (DUP!)
64 bytes from 192.168.1.30: icmp_seq=1 ttl=64 time=344 ms (DUP!)
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=445 ms (DUP!)
64 bytes from 192.168.1.35: icmp_seq=1 ttl=64 time=461 ms (DUP!)
64 bytes from 192.168.1.20: icmp_seq=2 ttl=64 time=4.62 ms
```

Figure 36: Pinging the Multicast IP Address

Now that we know we can reach the address; we can now test out the package. Note this should be done on all nodes/robots to ensure that they are correctly configured as well as able to connect to the multicast independently.

Testing the package can be done in the following steps:

1. Launch a local roscore on each computer.
2. Launch the master_discovery node at each computer, passing as an argument the mcast_group parameter to the specific multicast address to be used. As shown below:

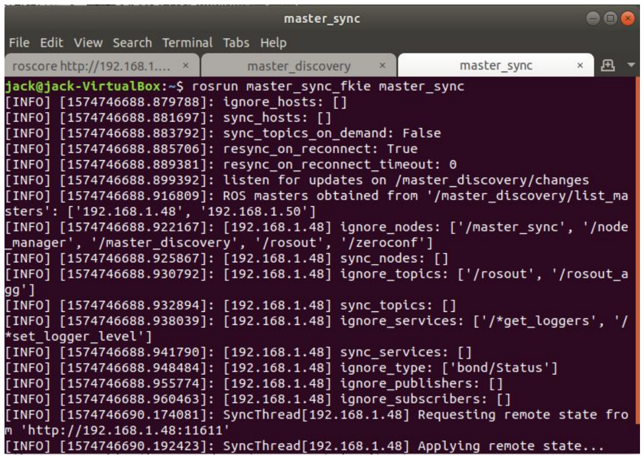


master_discovery
File Edit View Search Terminal Tabs Help
roscore http://192.168.1.50:11311/
master_discovery

```
jack@jack-VirtualBox:~$ rosrun master_discovery_fkie master_discovery _mcast_gro
up:=224.0.0.251
[INFO] [1574746307.942108]: ROS Master URI: http://192.168.1.50:11311
[INFO] [1574746307.948695]: Check the ROS Master[Hz]: 1
[INFO] [1574746307.949533]: Heart beat [Hz]: 0.02
[INFO] [1574746307.950236]: Active request after [sec]: 60
[INFO] [1574746307.950964]: Remove after [sec]: 300
[INFO] [1574746307.951683]: Robot hosts: []
[INFO] [1574746307.952371]: Approx. mininum avg. network load: 1.36 bytes/s
[INFO] [1574746307.967443]: Start RPC-XML Server at ('0.0.0.0', 11611)
[INFO] [1574746307.970689]: Subscribe to parameter `/roslaunch/uris`
[INFO] [1574746307.977969]: + Bind to specified unicast socket @(192.168.1.50:11
511)
[INFO] [1574746307.979695]: Create multicast socket at ('224.0.0.251', 11511)
[INFO] [1574746308.170615]: Detected master discovery: http://localhost:11611
[INFO] [1574746308.274803]: Added master with ROS_MASTER_URI=http://192.168.1.50
:11311/
```

Figure 37: Running the Master Discovery Node

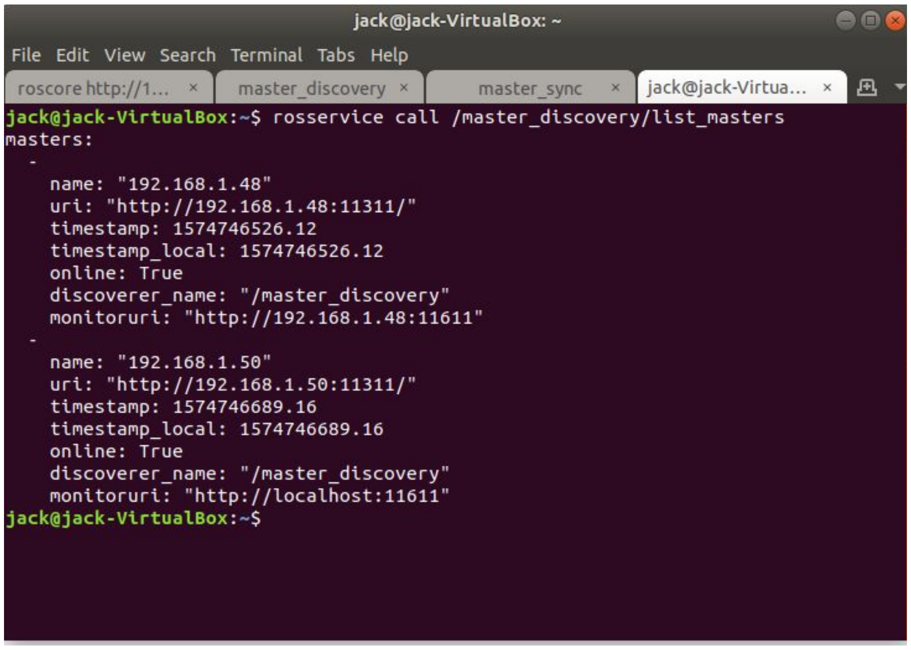
3. Launch the master_sync node on each computer in order to have all topics and services on all computers to be synchronized to each other. As shown below:



```
jack@jack-VirtualBox:~$ rosrun master_sync_fkie master_sync
[INFO] [1574746688.879788]: ignore_hosts: []
[INFO] [1574746688.881697]: sync_hosts: []
[INFO] [1574746688.883792]: sync_topics_on_demand: False
[INFO] [1574746688.885706]: resync_on_reconnect: True
[INFO] [1574746688.889381]: resync_on_reconnect_timeout: 0
[INFO] [1574746688.899392]: listen for updates on /master_discovery/changes
[INFO] [1574746688.916809]: ROS masters obtained from '/master_discovery/list_ma
sters': ['192.168.1.48', '192.168.1.50']
[INFO] [1574746688.922167]: [192.168.1.48] ignore_nodes: ['/master_sync', '/node
_manager', '/master_discovery', '/rosout', '/zeroconf']
[INFO] [1574746688.925867]: [192.168.1.48] sync_nodes: []
[INFO] [1574746688.930792]: [192.168.1.48] ignore_topics: ['/rosout', '/rosout_a
gg']
[INFO] [1574746688.932894]: [192.168.1.48] sync_topics: []
[INFO] [1574746688.938039]: [192.168.1.48] ignore_services: ['/*get_loggers', '/
*set_logger_level']
[INFO] [1574746688.941790]: [192.168.1.48] sync_services: []
[INFO] [1574746688.948484]: [192.168.1.48] ignore_type: ['bond/Status']
[INFO] [1574746688.955774]: [192.168.1.48] ignore_publishers: []
[INFO] [1574746688.960463]: [192.168.1.48] ignore_subscribers: []
[INFO] [1574746690.174081]: SyncThread[192.168.1.48] Requesting remote state fro
m 'http://192.168.1.48:11611'
[INFO] [1574746690.192423]: SyncThread[192.168.1.48] Applying remote state...
```

Figure 38: Launching the Master Sync Node

4. Run a rosservice call to identify all masters connected to the network as shown below:



```
jack@jack-VirtualBox:~$ rosservice call /master_discovery/list_masters
masters:
  -
    name: "192.168.1.48"
    uri: "http://192.168.1.48:11311/"
    timestamp: 1574746526.12
    timestamp_local: 1574746526.12
    online: True
    discoverer_name: "/master_discovery"
    monitoruri: "http://192.168.1.48:11611"
  -
    name: "192.168.1.50"
    uri: "http://192.168.1.50:11311/"
    timestamp: 1574746689.16
    timestamp_local: 1574746689.16
    online: True
    discoverer_name: "/master_discovery"
    monitoruri: "http://localhost:11611"
jack@jack-VirtualBox:~$
```

Figure 39: Checking Connections to the Multicast IP from ROS Masters

As you can see, we now have two computers each running their own roscore connected to the same multicast. You can now subscribe to any published topic from either of the masters.

Some key notes to remember are on each computer, the other computers running a master_discovery node in the same common network are automatically detected and the time stamp difference between them are reported.

To test the connection, you can run the following on one computer:

```
rostopic pub -r 1 /test std_msgs/Int32 1
```

On another computer run:

```
rostopic echo test
```

This will let one computer receive the published message from one master publishing that topic called test. You should see the results in the terminal.

4. Modified TurtleBot3 Start up Procedure/Running the Code

As shown above, multiple components come into play to ensure that the modified TurtleBots run and perform to peak performance. Therefore, as a result the start-up script/procedure is broken up into two sections:

- Pre boot (**before** you turn on the power bank)
- Boot (**after** you turn on the power bank)

Note that this procedure will need to be repeated on all robots you intend to use.

4.1 Pre-Boot Procedure

The key components that you need to ensure that get started up in the Pre-Boot procedure is to turn on the openCR board and ensure the Marvelmind hedgehog is not plugged in. This is to ensure that the openCR board gets assigned to the abstract control model (ACM) port 0. This is important because specific TurtleBot3 bring up actions require the openCR board being assigned on port 0.

Nevertheless, the steps in the pre-boot procedure are the following:

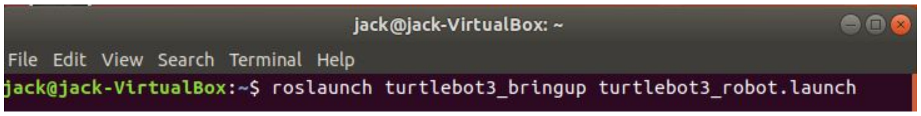
1. Turn on the openCR board
2. Ensure that the Marvelmind hedgehog on top of the modified TurtleBot3 is unplugged and then flip the dip switch into working mode.
3. Open up the Marvelmind dashboard on an external computer
4. Plug in the Marvelmind modem into the same external computer
5. Turn on all stationary beacons
6. Check the Marvelmind dashboard and ensure that the distance table has correct distances entered and that the Marvelmind hedgehog is transmitting its location
7. Turn on the power bank and connect it to the intel NUC

4.2 Boot Procedure

Key procedures that occur in the boot stage are similar to ones displayed above in section 3. Nevertheless, it is important that no warnings or errors occur in the boot procedure as if any arise it could cause errors while testing the code or possibly damage the hardware on the modified TurtleBot3.

Nevertheless, the steps in the boot procedure are the following:

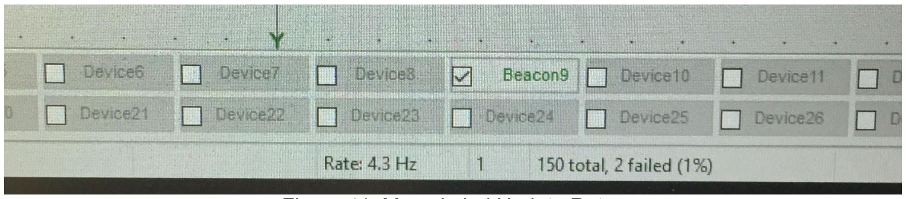
1. Turn on the intel NUC
2. SSH into the intel NUC and ensure you can run commands remotely as all following steps will be run on the robot's intel NUC
3. Run roscore
4. Launch the modified TurtleBot3 as shown below.



```
jack@jack-VirtualBox:~$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Figure 40: Launching the TurtleBot

5. Find the multicast IP address through the netstat -g command.
6. Run the master discovery node with the multicast IP address.
7. Run the master sync node.
8. Check the modified TurtleBot3's connection to the multicast IP address through the mastery_discovery command as shown.
9. Plug in the Marvelmind hedgehog on top of the modified Turtlebot3.
10. Run the Marvelmind hedgehog code.
11. Test the update rate of the Marvelmind hedgehog as shown below.
12. Launch all necessary launch files to start the physical tests



```
Rate: 4.3 Hz    1    150 total, 2 failed (1%)
```

Figure 41: Marvelmind Update Rate