Enhancing the Thymio to perform Search and Exploration

Simone Eandi
eandis@usi.ch

Marco Ferri
ferrima@usi.ch

Nadia Younis
younin@usi.ch

***Abstract*—The object of this project was to extend the capabilities of the Thmyio robot to make it able to explore an unknown indoor environment while searching for a target, under the assumption that an external perfectly accurate localization was given. This was achieved by adding a LIDAR to the robot in order to greatly improve its ability to perceive the environment and implementing a set of ROS nodes to process its readings. A mapping routine was implemented to generate a 3D map of the environment from the point clouds produced by the LIDAR, detect obstacles and create a 2D occupancy grid of the traversable terrain. A template matching algorithm was also designed to detect the presence of a given target from its 3D model and the point cloud mapping of the environment. Then, a set of path planning routines were implemented that make use of the occupancy grid and the target detection module to compute the robot control inputs that allow it to perform map coverage and target reaching. Finally, a Gazebo model of the robot has been created and used to simulate the execution in several different indoor environments where very good results were observed both for the mapping and path planning tasks.**

I. Introduction

The goal of this project was to enhance the capabilities of the very famous Thymio robot [1] in order to make it able to perform search and exploration tasks within indoor environments with flat terrain. The code produced during the development of our project is open source and available on GitHub [2].

The Thymio is a differential drive robot equipped with proximity sensor to detect close obstacles, these sensors have been proven to be insufficient alone to provide enough information to the robot about its environment to make it able to reliably and efficiently explore it; for this reason we have designed an improved version of the Thymio, named Thymar, which adopts a Lidar in order to enhance the perception capabilities of the base robot. The Thymar robot has been implemented as a set of ROS nodes, with each one corresponding to its own ROS package for better modularity, and tested in a simulated environment where it was asked to explore the surroundings in search for a target, defined as a sphere of 15 cm of radius. In particular, the implementation of the Thymar as been subdivided among three ROS nodes as follows: the *thymar_description* node is entirely responsible for the simulation of the robot, the *thymar_lidar* node takes care of mapping the environment and detecting the target while the *thymar* node is the main node responsible for controlling the Thymar robot. For handling the robot simulation and relative visualizations, Gazebo [3] and RViz [4] are being used in combination.

In the following document we will firstly describe the model used for the simulation of the Thymar robot, then we will go over how the mapping of unknown areas is performed, the technique used for identifying the presence of a specific target and the path planning approach used to control the robot; finally, the simulation results will be presented and discussed.

II. Simulating the Thymar

In order to simulate the Thymar robot, a model in Gazebo (figure 1) has been created by using the *thymio_description* ROS package [5] to simulate the Thymio robot and employing the *velodyne_simulator* package [6] to simulate a Velodyne VLP-16 Puck Lidar [7]. In particular, given that the proximity sensors of the Thymio have a maximum range of 15 cm, the LIDAR simulator has been setup to have a range between 14 cm and 3 meters in order to be complementary with them, furthermore, due to the limitations encountered while running the Gazebo simulation on the available hardware, the number of lasers projected by the Lidar within each vertical plane has been set to 64 and the overall reading frequency has been set to 0.5 Hz. The simulation of the Thymar has been completely integrated with ROS by making it publish and subscribe to a set of ROS topics under a common namespace (i.e. */thymar*) that can be used as I/O by any other ROS node. In particular, the model exposes its odometry on the */odom* topic and it continuously publishes the readings from the LIDAR in the form of point cloud messages on the */velodyne_points* topic; finally, the Thymar can be fully controlled by publishing on the */cmd_vel* topic its linear and angular velocities. An important mention deserves the odometry computation: given that the aim of the Thymar was to map the environment under the assumption that its localization is given, the source for the odometry has been set to the Gazebo simulator instead of the differential drive encoder of the Thymar to prevent any error drift and have a perfect localization.

robot.png

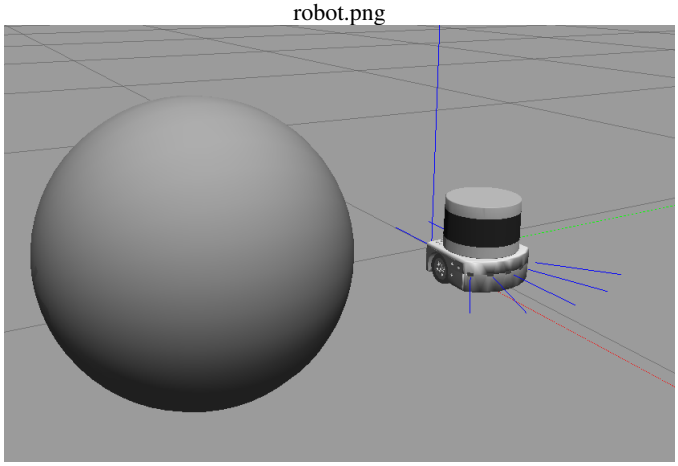


Fig. 1. The Thymar robot and target model in Gazebo

III. Mapping

All the computations related to the mapping of the environment have been assigned to the *thymar_lidar* node situated within the *thymar_lidar* package: it takes care of reading the point cloud messages from the simulation, combining them to create a 3-dimensional map of the environment, separating the obstacles from the traversable terrain and generating the 2D occupancy grid. The node has been implemented fully in C++ in order to make use of the PointCloudLibrary [8] routines for handling point clouds and divided into an interface (Thymar_lidar.cpp) that takes care of initializing the node and managing all the publications and subscriptions with ROS topics and a separate class (PointCloudMapper) which performs all the mapping computations related to point clouds.

A. Mapping the environment

Whenever a new point cloud message is published by the Thymar LIDAR, it is read and represented as a cloud of XYZ points PCL object exploiting the builtin automatic conversion between the ROS PointCloud2 type and the PCL types. Since the received point cloud is represented with respect to the Thymar reference frame, the pose of the robot at the moment of reading is retrieved from the */odom* topic and used to apply a rigid transformation to the point cloud that brings it in the world reference frame defined as the frame associated to the odometry. Then, since the odometry are perfectly accurate, the 3D representation of the environment can be simply achieved by adding the new points to those currently in the world point cloud. In particular, to prevent the world point cloud to become extremely dense and computationally more expensive to analyse, downsampling is performed every time a new point cloud is added through a Voxel filter with leaf size of 5 cm in all directions (**algorithm 1**). Remarkably, to avoid having to synchronize the odometry and point cloud messages, the point cloud produced by the LIDAR is read only when the Thymar velocity is purely linear, in which case no synchronization is necessary due to the low speed.

All the published topics can be visualized in RViz and figure 2 shows how different time steps LIDAR readings can be merged together for creating the 3D map of the environment by taking account the relative position with respect to the initial pose.

merge.png

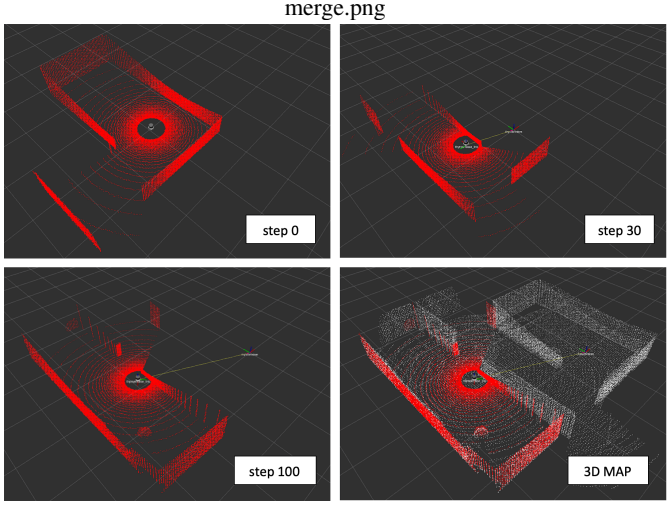


Fig. 2. 3D Map made up by merging point clouds at different time steps

B. Separating obstacles and terrain

The distinction between obstacles and terrain in the point cloud has been performed by simply applying a directional pass through filter which separates points according to their z coordinate with respect to a fixed threshold. Instead of performing this operation on the world point cloud, we chose to do it directly on each new point cloud after it is represented in the odometry reference frame, using an index-based approach to preserve the original cloud, and incrementally build the obstacles and terrain global point clouds in a similar fashion to what was done for the world cloud, including periodic downsampling (**algorithm 1**).

C. Generating the occupancy grid

In order to provide a simple and computationally efficient representation of the environment to be used for path planning, a 2D occupancy grid is generated from the point clouds. The grid has been defined as a matrix sufficiently large to cover all the environment where each cell represents the content of a 5cm by 5cm square region of the environment surface as a numeric value: a -1 value is assigned to unknown areas while a probability between 0 and 100 of being an obstacles is assigned to discovered areas (for simplicity a 0 or 100 binary approach has been used). The occupancy grid is built iteratively in a similar manner to what was done for the world point cloud by updating its content every time a new point cloud message is read. More in detail, the grid update is performed as follows: first, for each point in the new point cloud that has been found to belong to an obstacle, the corresponding cell in the occupancy grid is set to be an obstacle as well as all its neighboring cells to have a more robust representation, then all the points corresponding to terrain are parsed and the corresponding cells in the grid are set to traversable if and only if they were not previously labeled as terrain. Since each scan of the LIDAR acquires points belonging to the terrain arranged in a set of concentric rings, this result in a large groups of cells corresponding to points lying between those rings that remain undiscovered while obviously corresponding to terrain due to the LIDAR acquisition pattern; this issue is mitigated by post-processing the grid after each update in order to set to terrain all the unknown cells having a percentage of neighbors labeled as terrain above some threshold.

Figure 3 shows how the point clouds corresponding to obstacles are used to generate the 2D map of known and unknown areas.

obstacles+map2d crop.png

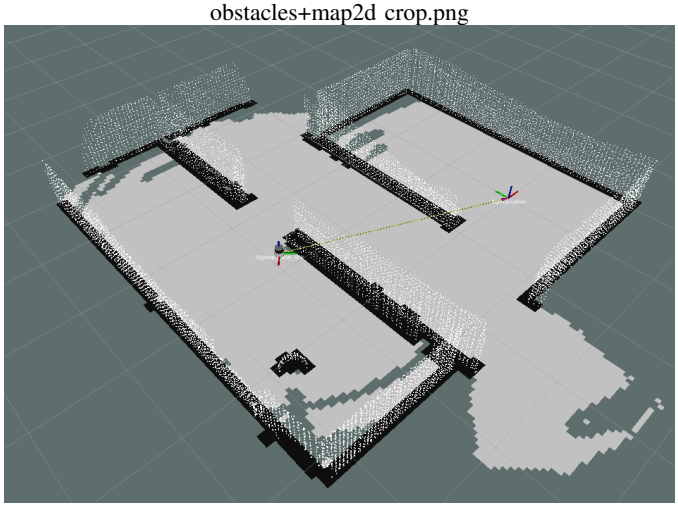


Fig. 3. Obstacles point cloud and 2D map generated from it

IV. Target detection

The *thymar_lidar* node is also responsible for detecting the given target, a sphere of 15cm of radius, whenever it is encountered during the exploration of the environment. The target detection is entirely performed by relying on the 3D point cloud generated by the LIDAR: given that the model of the target is assumed to be known, a template matching algorithm such as RANSAC can be exploited to detect the target. In particular, every time that a new point cloud is processed a clustering algorithm is applied on the global obstacle point cloud and only the clusters having a number of points within some threshold are selected, then, for each one of those, the RANSAC algorithm is applied: if for one of those cluster a fitting of the target model with a number of inliers above some threshold is obtained, then the

Algorithm 1 Mapping

```
init publisher and subscriber
init word,obstacles and terrain point clouds
init occupancy grid
while node running do
  if new point cloud message then
    represent the point cloud in the global frame
    split point cloud into obstacles and terrain
    add full point cloud to the world point cloud
    add obstacles and terrain to the respective point clouds
    downsample word obstacles and terrain clouds
    update occupancy grid
  end if
  publish global point clouds and occupancy grid
end while
```

target is considered to be found and the node publishes a marker in correspondence of the location of the fitted model (**algorithm 2**). Notably, the detection routine is applied on the global obstacles point cloud and not on the individual readings, this was done because the target may actually not be recognizable until multiple point clouds from different views are joint together. Finally, note also that the filtering applied on the clusters of points have been made possible by the fact that the density of the point clouds is kept constant by the continuous downsampling operations and must be fine tuned on the specific model of the target.

Algorithm 2 Target detection

```
init publisher and subscriber
while node running do
  if new point cloud and target not found then
    cluster the global obstacle point cloud
    reject all clouds with a number of samples outside some range
    for cluster in remaining clusters do
      try to fit the target model in the cluster
      if fits then
        set target to found
        publish target location
      end if
    end for
  end if
end while
```

V. Basic exploration

The exploration task is handled by the *thymar* package, which is responsible for driving the Thymar in the selected world for mapping the environment in order to reach the target. This is achieved by defining a ROS controller that subscribes to the topics defined in the previous sections and uses them for computing how to move. Basically, the controller waits that the robot is ready (subscribers are working properly), for then handling the simulation through an infinite loop that evaluates the current situation and computes step-by-step robot's velocities.

A. Sensing the environment

From now on, the robot's perception about the environment will only consider the 2D map computed during the mapping phase, accordingly subscribing to the occupancy grid message published by the *thymar_lidar* node. Since this map is a grid discretization of the world with a specified resolution (5cm, in this case), the odometry information retrieved from the */odom* topic has to be properly converted into the corresponding occupancy grid cells in order to localize the robot within the map. Its position will be taken into account for evaluating what surrounds the robot at a certain moment. It is useful to remember that we use the Gazebo ground truth odometry for this task since we assume that localization is given correctly.

B. Avoiding the obstacles

Given the map of the discovered environment and the robot position, a collision-avoidance policy can be defined for exploring the world without crashing into obstacles. Our basic exploration algorithm drives the robot straight forward since no obstacle is found, otherwise, the best turning direction is chosen for trying to follow a free way. Given the robot pose and orientation, obstacle detection is done by checking for free cells in the occupancy grid, considering the ones that are directly in front the robot within a chosen *visibility* parameter (default 1 meter); for doing so, also robot width is taken into account during the computation.

Once an obstacle is found, an orientation searching phase starts. The algorithm firstly computes the best turning direction (left or right) by considering obstacles distances in the `90`-degrees cone in front of the robot, then a *steering-step-degrees* parameter is used for evaluating possible orientations in that direction, using a doubled robot *visibility* here. The search keeps going until a certain orientation leads to no obstacles in front of the robot or a substantial improvement is found in the distance between currently seen obstacles and the one that is obtained by turning in that orientation. If no way satisfies the condition, a try with decayed *steering-step-degrees* is done and if the problem persists, then a random orientation between `-130` and `130` degrees is chosen. Finally, the robot is guided by the controller for actually rotating until the chosen orientation is reached, so the obstacle avoidance algorithm restarts from the beginning.

This algorithm correctly works in most of the environments for exploring the available space until the target is reached, but easily leads to infinite loops in which the robot keeps traversing the same area over and over. Even if it does not happen, it is clear that this exploration technique is very inefficient for relatively large worlds. We will see in section VII how the problem has been taken out.

VI. Reaching the target

While exploring, the target detection algorithm keeps running in parallel through the *thymar_lidar* node described before. As soon as the target is identified from the point cloud seen by the robot, its pose is published on a proper topic and a marker is published for better visualization in RViz. At this point, it is possible to retrieve both the robot and the target poses inside the map and this information can be used for driving the robot towards the target by using a path planning algorithm.

A. Path planning

Considering the available space as a grid, one of the most intuitive approaches for computing path planning is the A* algorithm [9], which would treat the map cells as weighted graph nodes in order to find the least costing path between two nodes: the robot and the target. A* starts by visiting all the available cell that are reachable from the starting one (i.e., the robot pose) and then keeps iteratively exploring only the best ones accordingly to a given cost, which is computed by the exact number of (minimal) steps needed for reaching that cell plus the cost given by a heuristic function. As soon as the A* search

finds the target cell, the best path for getting there is computed by back-propagating into the explored graph. In order to run A* on our problem, two parts of the algorithm must be defined: the heuristic and the neighborhood function.

The heuristic function is used for evaluating the expected cost of reaching the goal from a certain node of the graph, without actually knowing the real cost the node will have at the end. This is used for choosing nodes from which the graph exploration should continue first, which according to the heuristic function will also be the ones nearest to the target node. In the examined case, this heuristic function can simply be the euclidean distance between the cell we are evaluating and the cell that corresponds to the target pose.

On the other side, the neighborhood function computes the neighbors of a given node in the graph, which means the cells that can be reached from that particular cell we are considering. In an obstacles-free 2D map the neighborhood function will return all the 8 adjacent cells of an input cell, but when dealing with obstacles some directions will be filtered out if they actually correspond to an obstacle. In practice, this is not enough for computing neighborhood in our problem, since the robot is not a single point but instead has s dimension which occupies more than one cell in the map. That is why a parameter is needed for considering how far the robot has to stay away from obstacles, that we will call *safe-distance*; this value has to be compliant with robot width and also allow the robot itself to traverse passages that are wide enough for it to pass through. In real-world applications, this *safe-distance* also has to consider possible noise in the localization task or eventually moving obstacles - which is not the case in our simulation.

B. Path following

Once A* has found the target pose during the graph discovering (which represents the map), the path to reach that cell is reconstructed and finally published as a topic for visualization, as shown in figure 4. This path is then followed pose-by pose by the robot in order to reach the target pose. However, the path has the same grid resolution, so it contains all the cells that connect the robot and the goal which are 5cm wide each. Approaching poses which only differ by 5 centimeters is not easy nor worth it for the robot considering its size, so a defined number of poses are skipped in each step while following the path.

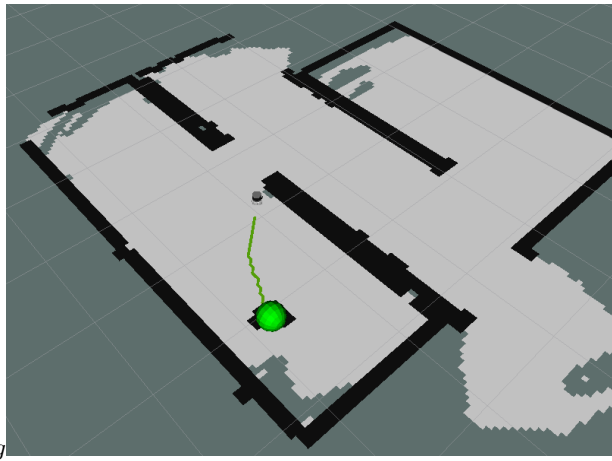
Given a single pose which the path is composed of, the robot uses a standard motion controller for reaching it. As a constraint, we limited the robot to just going straight or rotating, but not both moving forward and steering at the same time. Given the current pose and the goal, the robot initially turns around to face the pose to reach, then proceeds straight ahead until its coordinates in the world are relatively close to the goal pose. If a final orientation is set, the robot rotates again in order to satisfy the requirements.

C. Back to the base

The path planning and tracking algorithms described in this section can actually be used to reach any point in the map as far as its coordinates are known. A common functionality after the target has been reached is to go back to a base (e.g., for returning what has been found). This goal is very simple to achieve by using what we have defined before since it is possible to set the initial pose of the robot as input for the path planning function for easily going back to the position where the robot has been spawned.

VII. Improving the exploration

The basic exploration approach defined in section VII resulted to be very inefficient and not useful at all in certain cases. That is why



target$_{p}lancrop.png$

Fig. 4. Path for reaching the target object

also the exploration phase has been redesigned using path planning.

A. Exploring unknown areas

In order to optimize the exploration of the whole environment, it is useful to prioritize the reaching of those parts of the map which are marked as unknown in the occupancy grid. It is reasonable that always trying to explore unknown areas will certainly lead to also actually explore all the available space, and so, to the target. For achieving this, a searching approach is required and Dijkstra's algorithm [10] has been chosen. It is a generalization of A* in which the weight of each node only accounts for the exact cost computed by the effort required for reaching that specific node from the starting point, without the usage of any heuristic function that points towards a final target node.

Since the only difference with A* is the absence of the heuristic function, Dijkstra practically explores neighbor nodes in every direction such as a classic BFS algorithm [11] would do. Another fundamental difference that has to be defined for our goal is when to stop the search. If when trying to reach a target the algorithm stops when the target is found in the searching graph, this time the stopping criteria is the discovery of an unknown area. If Dijkstra is stopped as soon as an unknown-marked cell is found, then the computed path would be the shortest way for reaching a currently unknown area, which will hopefully lead to the target object in the minimum time.

The computational resources required for computing Dijkstra are greater than A* since no preferred direction is chosen. Our algorithm always points to the nearest unknown cell but, while exploring, this cell will surely become known when approaching. That is why Dijkstra needs to be periodically recomputed in order to update the path the robot has to follow. A trade-off between exploration optimality and computational effort must be found and several experiments led to set the re-computation frequency every time the robot has approximately moved for 0.75 meters.

However, it sometimes happens that while moving the robot discovers new obstacles that lie on the path it is following (e.g., when approaching the corner of a wall). In this case, as soon as those obstacles are facing the robot an early path re-computation is required to avoid crashing against them.

B. Map coverage

The interesting fact behind using Dijkstra for reaching unknown areas is that the same approach can be actually used for handling a

map coverage task, so the mapping of all available traversable terrain. This is particularly useful when a robot is placed in an unknown environment and has to map it, which can have several purposes in real-world applications.

The algorithm used for exploration is exactly the same for map coverage and the only difference relies on failures management. While searching for the target, if Dijkstra is not able to find unknown areas and the target has not been found yet, then it would mean that (assuming there is no error in the mapping task) the target is not present in the environment. However, it can be also stated that the entire environment has been explored. Similarly, if the current objective is to discover the whole map instead of finding the target, a failure in the Dijkstra algorithm means the task has been successfully achieved.

It also has to be noticed that achieving map coverage is actually even more expensive than just exploring the map since the unknown areas could now be very far from the current position, especially if during the exploration some small parts of the environment have been left out. In order to avoid frequent and very long path updates, a workaround has been taken into consideration: if the path is longer than 5 meters, then the re-computation happens just when at least half of it has been traveled.

C. State machine

Given all the features discussed until now, the practical possibilities with such a robot are very powerful. For controlling the robot goals step by step, some states and relative transitions have been defined.

- `exploring_random`: tells the robot to follow the obstacle avoidance policy defined in section V-B
- `exploring_smart`: tells the robot to prioritize the exploration of closest unknown areas such as described in section VII-A
- `exploring_coverage`: tells the robot to achieve map coverage as seen in section VII-B
- `chasing_target`: tells the robot to do path planning in order to reach the target as described in section VI-A
- `chasing_goal`: utility state that is used in combination with other tasks for reaching a given pose in the map, such as simply avoiding an obstacle or following the path with the algorithm briefly discussed in the path following section VI-B
- `returning`: tells the robot to enable path planning towards the initial pose as we have seen in section VI-C
- `end`: final state, used for terminating the program

A combination of all these states can be easily defined for solving different types of problems. An exhaustive simulation will be made by setting the robot for finding the target, reaching it, explore all the maps and the finally return to the base. However, other combinations can be defined such as, for example, the possibility of returning as soon as the target has been found or simply achieving map coverage without anything else.

VIII. Conclusion

Some basic techniques for driving a wheeled robot in search of a target object have been presented. Using a LIDAR for sensing the unknown environment, a 2D map has been created for then applying two different types of path planning for finding a given target and exploring all the available space. Working with the point clouds for defining the map works very well, so it does path planning on the resulting map. Several environments with different characteristics have been tested out and results are very promising. It is cool to notice that our A* implementation also works properly when the target pose is known, but actually resides inside an unknown area; in this case, paths updates are very frequent and the map is created while approaching the target, as shown in figure 5 and video [15].

However, for taking our robot into the real-world some other implementations would be needed, later discussed in section VIII-B.

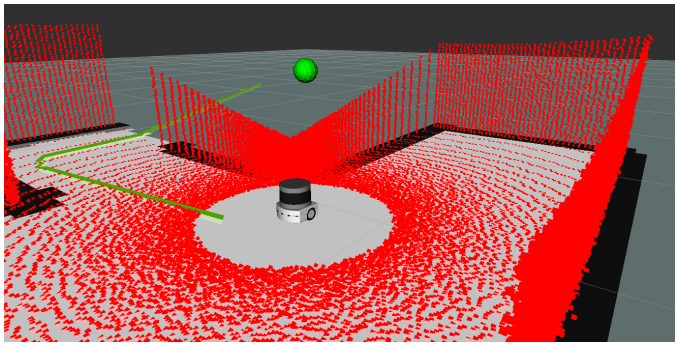


Fig. 5. Path plan for reaching a target placed in unknown area

A. Demos

Some videos [14] showing Gazebo + RViz simulations are available together with an interactive presentation [13] which summarizes what has been done.

B. Future works

Bringing the robot to the real-world requires further tasks:

- Target recognition could be improved for better detecting target also when obstacles that are similar to the target in shape and size are available in the environment (e.g., cylinders with the same radius of our target sphere)
- During development, we assumed that odometry is given and correct for simplifying the environment we work with (we use the information that Gazebo provides us); actually, real-world implementations require both the Mapping and Localization tasks to be solved with proper SLAM techniques, also because the odometry information retrieved by the built-in robot sensors are too noisy for being considered acceptable for the mapping task, as shown in one available video [16]
- A reasonable alternative for reducing noise in the mapping task is to also implement an Iterative Closest Point algorithm [12]

References

[1] Thymio: https://www.thymio.org/it/
[2] Project repository: https://github.com/seandi/thymar
[3] Gazebo simulation environment: http://gazebosim.org/
[4] RViz visualization for ROS: http://wiki.ros.org/rviz
[5] *thymio_description* source repository https://github.com/jeguzzi/ros-aseba
[6] https://bitbucket.org/DataspeedInc/velodyne_simulator/src/master/
[7] http://www.3dtarget.it/eu/it/droni/prodotti-droni/lidar/velodyne-vlp-16-puck-detail.html
[8] PointCloudLibrary C++ https://pointclouds.org/
[9] A* algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm
[10] Dijsktra's algorithm: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[11] BFS algorithm: https://en.wikipedia.org/wiki/Breadth-first_search
[12] ICP algorithm: https://en.wikipedia.org/wiki/Iterative_closest_point
[13] Slides presentation: https://kutt.it/yNJ2NU
[14] All available videos: https://kutt.it/CpBVsQ
[15] Reaching a target placed in unknown area: https://kutt.it/j05GtL
[16] Error in mapping when using simulated odometry: https://kutt.it/J9R0Ep