POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

CORSO DI MOBILE ROBOTICS

Prof. Ing. Luca De Cicco
Ing. Carlo Croce
Ing. Gioacchino Manfredi

Agromatic Robot

Students:
Gianluca Barile
Nicola Pace
Francesco Paciolla

ANNO ACCADEMICO 2020-2021

Abstract

The aim of this project is to simulate in the ROS environment the autonomous navigation of a wheeled mobile robot, in particular the Husky A200 in an outdoor environment. For this applications we build a 3D model of an agricultural field in which there are fixed and moving obstacles represented by the tomatoes plants and farmers present in the field. The robot has to preform mainly 2 different tasks: the harvesting of tomatoes, that are put in a box present on the top plate of the robot and then, when it reaches a fixed weight, the robot goes autonomously to the checkpoint to empty the fruit box, avoiding the obstacles. The other task emulates the watering or fertilization of the field using a random counter and a checkpoint, when the counting ends the robot comes back to the checkpoint avoiding obstacles. In addition, we created customized functions which allow the saving of some intermediate waypoints in a .json file and the tele-operation of the robot.

Contents

1 Introduction

Today a lot of mobile robots work only in indoor environments, but outdoor navigation operations tackle more challenges: fewer data resources are available, the environment features like the terrain and the lighting conditions are harder to control and the operating conditions are more volatile. Usually, outdoor robots are used in fields like demining or exploration, but they offer the potential for new application areas such as agriculture, as showed in this project that we call Agromatic Robot. In particular, the goal of the project is to implement the autonomous navigation in ROS of a wheeled mobile robot, in particular the Husky A200. The robot has to cover the surface of an agricultural field in order to perform two different tasks, managing between parallel rows of tomato plants, the barriers of the field and moving obstacles. Two different use cases have been developed:

- **Harvesting**: The robot receives and stocks the tomatoes at each predefined waypoint previously saved during the navigation, carrying them to a checkpoint, different from the start point, where they are unloaded. This operation is carried out only when the weight of the fruit box present on the robot's base link is at his max capacity (4.5 Kg). After the unloading the robot comes back to the successive saved waypoint and continue the tomatoes collection.
- **Irrigation/Fertilization**: The robot follows the predefined waypoints previously saved and comes back to the checkpoint, different from the start point, after a random time, represented by the counter, whatever is its position in the field. The random time simulates different needs of irrigation/fertilization during a day. Ideally, at the checkpoint the robot fills the tank for the irrigation/fertilization of the field. After this operation the robot comes back to the successive waypoint and continue the operations.

For this application the robot that we choose to use is a Husky A200 [1], a differential drive robot which is equipped, in this case, only with a LIDAR sensor to detect the obstacles and helps the localization of the robot in the field. For the implementation of the navigation stack it has been used the Dynamic Window Approach (DWA) local planner[2], which is a great algorithm with static obstacles and the Timed Elastic Band (TEB) local planner[3], in presence of mobile obstacles. The static map of the environment is known after having been generated for the first time using a SLAM (Simultaneous Localization and Mapping) algorithm called Gmapping.

1.1 The robot

The Husky A200 robot, developed by the company Clearpath Robotics Inc.[1], is a differential drive wheeled mobile robot with four wheels that weighs approximately 50 kg with payload and has a top forward speed of 1.0 m/s. It is engineered to thrive in harsh outdoor environments thanks to its 4x4 zero-maintenance drivetrain, furthermore its rugged all-terrain tires and an important ground clearance allows the robot to tackle challenging real-world terrain. Its large payload capacity and its power system accommodate an extensive variety of payloads and there is also the possibility to customize the robot to meet the research needs depending on the different fields in which the robot is employed. To expand the capabilities of the robot it can be equipped with different sensors as LIDAR, IMUs, GPS and cameras but for the purpose of this project the only sensor utilized is a LIDAR.

Figure 1: Views of the robot.

Dimensions	Weight	Maximum Payload	Operating Time
$990x670x390$ mm	50 kg	75kg	3 h

Husky has very-high resolution encoders that deliver improved state estimation and dead reckoning capabilities. A finely tuned, yet user adjustable controller, offers smooth motion profiles even at slow speeds ($<$ 1 cm/s) with excellent disturbance rejection. When commanded with a positive translational velocity (forward), wheels travel in the positive x-direction.

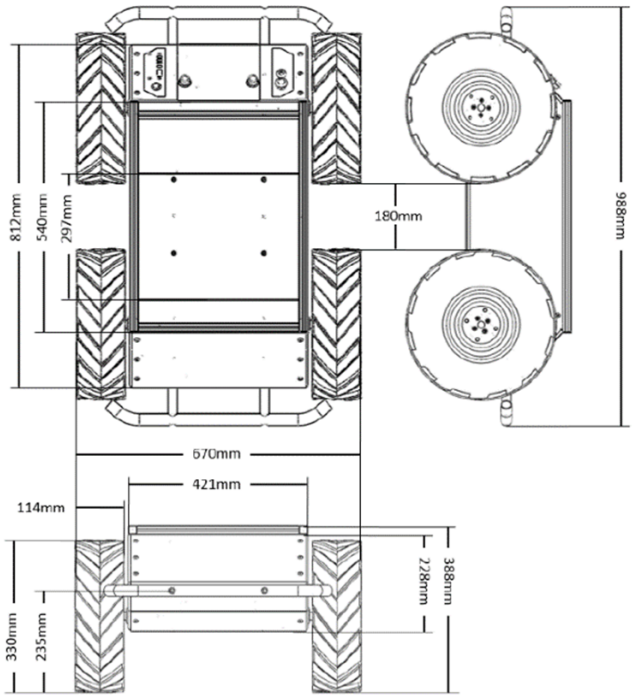

Figure 2: Dimensions of the robot.

2 Project Development

ROS is an open-source robotic middleware suite composed by a set of software libraries and tools that help to develop and program robots. It provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.

2.1 Unified Robot Description Format - URDF

The Unified Robot Description Format (URDF) is the most popular human-readable way to describe the geometry of robots and their composing parts in ROS; this format is an XML file. In particular, an URDF file:

- Build a visual model of the robot.
- Define movable joints.
- Add collision and inertial properties.

Using the Xacro language it is possible to clean up the code developed in URDF. In this project, for the harvesting use case developed, the model of the Husky A200, has been enriched adding the model of a fruit box, designed using a CAD software called Blender. The developed .dae Collada model, allow the store of the collected tomatoes. In Figure 3 it is shown the 3D model of the fruit box in Blender.

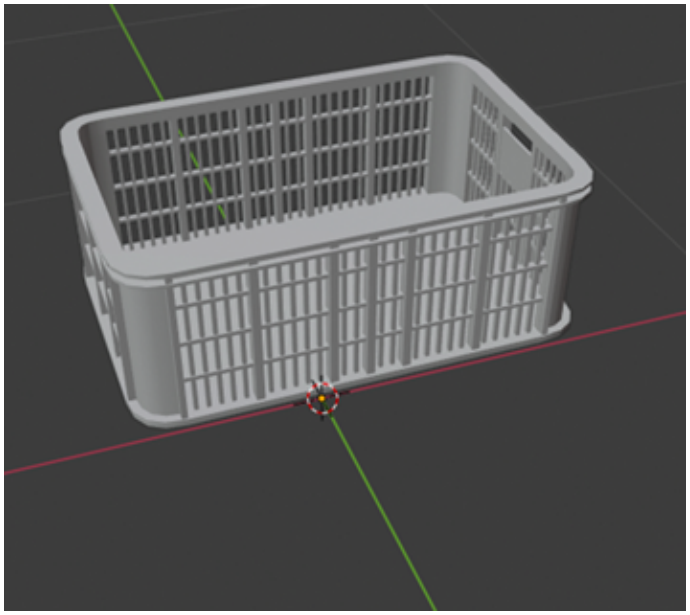

Figure 3: 3D model of the fruit box in Blender.

The fruit box is placed on the top plate of the robot and it has been scaled and modified in order to adapt it to the robot's model dimensions. Inside the fruit box, during the navigation and in correspondence of the saved waypoints are spawned the tomatoes.

Figure 4: Harvesting task.

For the other task (Irrigation/Fertilization) instead of the fruit box there is a tank, that has also been designed in Blender as a .dae Collada model, on the top plate of the robot to simulate the irrigation/fertilization of the field.

Figure 5: Irrigation/Fertilization task.

In the URDF file, after the geometry description of the robot, it is possible to define the sensors. For the project the LIDAR sensor's line SICK LMS1xx has been implemented. In this way, the robot can sense the environment, detect obstacles and build the static map.

2.2 Gazebo

Gazebo is a robot simulation toolbox that offers the ability to accurately and efficiently simulate robots in complex indoor and outdoor environments. It offers a robust physics engine and a high-quality graphical user interface (GUI). The customized outdoor environment developed is an agricultural field, bordered by a fence. It is composed of a series of parallel and narrow rows of tomato plants that the robot must avoid during its navigation. It is important that the "visual" elements of the plants have been specified as "collision" elements, otherwise Gazebo will treat them as "invisible" to range sensors and collision checking algorithms. The collision element is a direct sub-element of the link object and defines its shape with <geometry> tag.

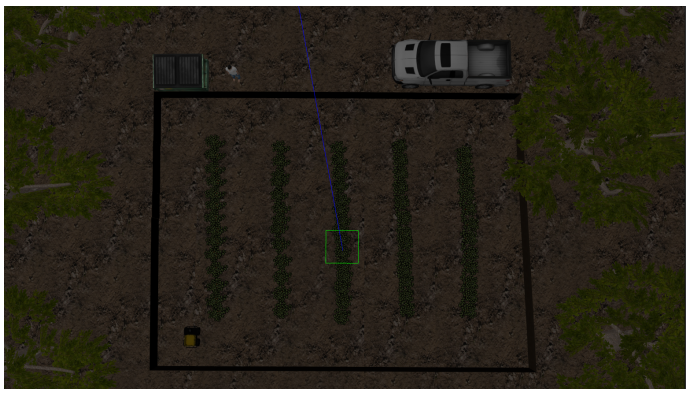

Figure 6: Gazebo World.

2.3 Navigation Stack

The Navigation Stack is a set of ROS nodes and algorithms that allow to autonomously move a robot from one point to another, avoiding static and moving obstacles taking information from odometry and giving as outputs the velocity commands to send to a mobile base. The scheme of the Navigation Stack is shown in the following Figure.

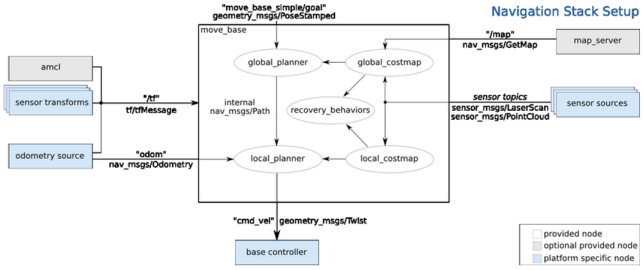

Figure 7: Scheme of Navigation Stack.

The autonomous navigation is divided into three fundamental steps:

2.3.1 Map Generation

The first step of the navigation is the map generation. A 2D static map is created starting from an unknown environment with a Real-Time simultaneous localization and mapping (SLAM) methods such as Gmapping, Cartographer or Hector. In this project, for the creation of the static map of the environment the Gmapping algorithm has been used, in Figure 8 is shown the mapping of the tomatoes field executed by the robot. The Gmapping has many parameters that need to be changed to optimize the performance, it is performed moving the robot in the agricultural field through the tele-operation via the keyboard .

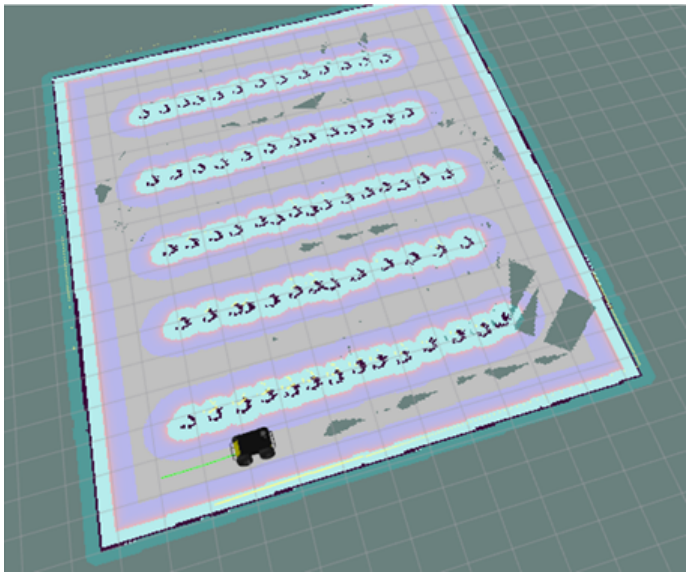

Figure 8: Gmapping.

The gmapping ROS package provides a laser-based SLAM method to create a 2D Occupancy Grid Map (OGM) with RViz. The OGM represent the environment as binary variables:

- White area is collision free.
- Black area is obstacles and inaccessible area.
- Grey area represents the unknown area.

After creating a complete map of the area, it is possible to save the map data to the local drive.

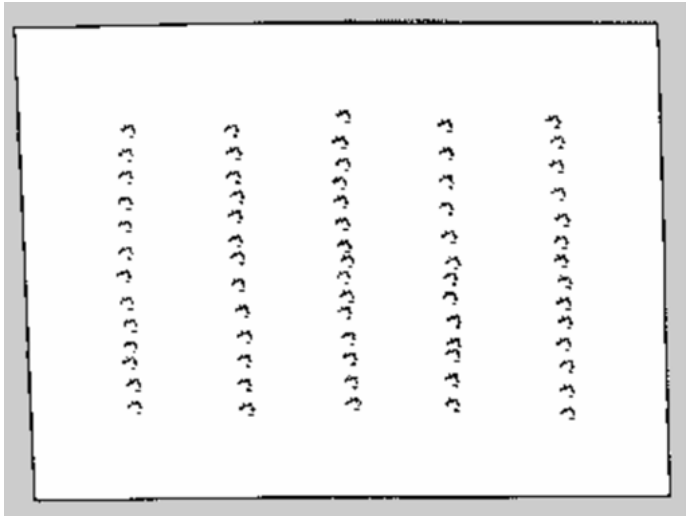

Figure 9: Generated map.

2.3.2 Localization

For this step it has been used the Adaptive Monte Carlo Localization (AMCL), which is a useful method to localize the robot in a static map. It uses a probability theory to track the pose of the robot with respect to the environment. The AMCL in Rviz displays the cloud of red arrow which indicates the estimated pose of the robot; the arrows are initially sparse because the filter parameters are initialized to the default values.

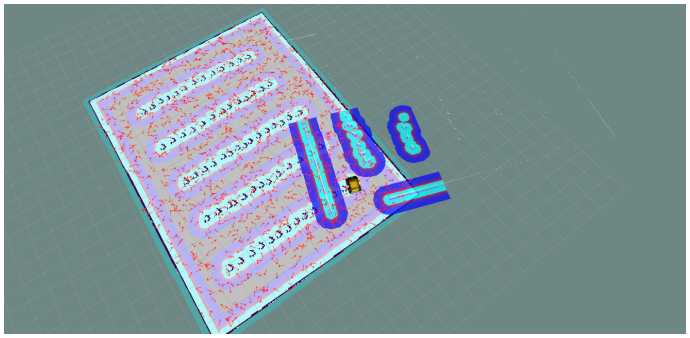

Figure 10: AMCL with default parameters.

The localization algorithm reads the laser scans and transformation messages, giving as outputs the pose estimation of the robot. As can be seen in Figure 11.
After giving a 2D Pose estimation of robot pose through Rviz, the pose of the robot is defined as vectors whose origins are the estimate positions of the robot whereas directions are their estimate orientations.

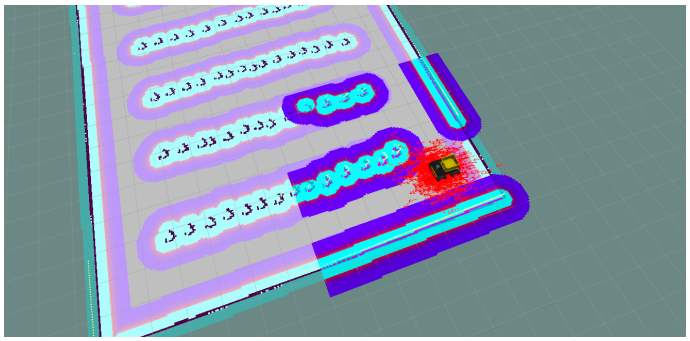

Figure 11: AMCL.

2.3.3 Path Planning

Finally there is the path planning step which is divided in global and local planner; the first one finds the optimal path which links the current position of the robot to the goal position, with a prior knowledge of the environment and static obstacles. The latter computes in real-time the new path in order to avoid dynamical obstacles.
The Navigation Stack uses the costmap-2D package that computes two different cost maps to store information about obstacles in the world. One costmap is used by the global planner to create long-term plans over the entire environment, the other one is used by local planner to avoid obstacles. For the local planner, the `base_local_planner` package provides a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the

planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way the planner creates, at least locally around the robot, a value function represented as a grid map. This value function encodes the cost of traversing through the grid cells. The controller's job is to use this value function to determine the linear velocities dx, dy and the angular one dtheta to send to the robot.

2.3.4 DWA Local Planner

In presence of only static obstacles, as the tomatoes plants and the barriers of the field, the Dynamic Windows Approach (DWA) can be used for the autonomous navigation. It is implemented in the ROS package `dwa_local_planner` [2]. Given a static map and a costmap, the local planner produces velocity commands to send to the mobile base.

The algorithm is composed of these steps:

- Discretely sample control space (dx,dy,dtheta).
- Forward simulation to predict what would happen if the sampled velocity is applied.
- Evaluate each trajectory with respect to a metric that takes into account proximity to obstacles, to goal, to global path and speed limit.
- Discard illegal trajectories (obstacle collisions).
- Highest-scoring trajectory is picked and send the associated velocity to the mobile base.

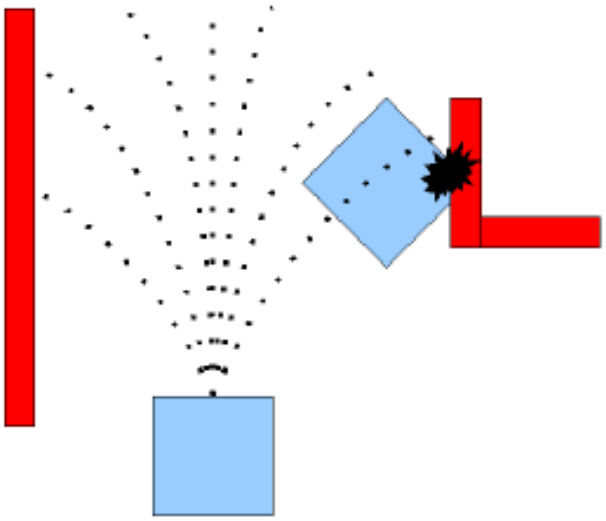

Figure 12: DWA algorithm.

2.3.5 TEB Local Planner

In presence of moving obstacles can be used the Time Elastic Band as local planner. The `teb_local_planner` package, [3] implements an online optimal local trajectory planner for navigation and control of mobile robots as a plugin for the ROS navigation package. The initial trajectory generated by a global planner is optimized during run-time to minimize the trajectory execution time (time-optimal objective), the distance from obstacles and the compliance with kinodynamic

constraints such as satisfying maximum velocities and accelerations. The optimal trajectories are efficiently obtained by solving a sparse scalarized multi-objective optimization problems. The user can provide weights to the optimization problem in order to specify the behavior in case of conflicting objectives. Since local planners, such as the Timed-Elastic-Band, get often stuck in a locally optimal trajectory as they are unable to transit across obstacles, an extension is implemented: a subset of admissible trajectories of distinctive topologies is optimized in parallel. The local planner is able to switch to the current globally optimal trajectory among the candidate set. In this way, the obstacles avoidance can be done in the presence of moving obstacles as required.

2.3.6 Save Goals

The definition of the required path that the robot has to follow is pursued through the sending of specific goals. These points are saved during a previous warm-up lap in which the robot is tele-operated by keyboard through the field previously mapped. During this lap, the waypoints have been saved by using a custom Python code which through, every time the “s” button is pushed, the current pose of the robot is saved. All the points are saved in a .json file in an ordered way, defining the position and orientation of the robot at each waypoints. An example of a .json file containing some poses of the robot is showed:

```
[
    {
        "posizione": {
            "x": 6.993728843224592,
            "y": 3.988651951240912,
            "z": 0.0
        },
        "orientamento": {
            "x": 0.0,
            "y": 0.0,
            "z": -0.7038142758376179,
            "w": 0.7103840265146517
        }
    },
    {
        "posizione": {
            "x": 7.632321842944352,
            "y": -3.512711177210358,
            "z": 0.0
        },
        "orientamento": {
            "x": 0.0,
            "y": 0.0,
            "z": -0.7233377737250651,
            "w": 0.6904943628317805
        }
    }
]
```

Once all the needed points have been obtained, it is possible to push the “d” button to save the poses and close the file. During the autonomous navigation the custom Python code named "Sending Goals" will read the .json file and will follow them retracing the desired path.

2.4 Sending Goals Node

To follow the desired path, after saving the intermediate waypoints in the .json file to reach the checkpoint, the idea is to provide them to the path planner, so that the robot can repeat the first

driven route. Firstly, it has been created the sending-goals node and when the file `sg_spawn_tomato.py` is running an object is created and it subscribes to the `/move_base/status` topic and became a publisher in `/move_base_simple/goal`.

At the beginning, the first goal is taken from the `poses.json` file and published into the `/move_base_simple/goal` topic; here, the move-base node reads the goal and provides the input velocities for the wheels to reach the goal. Once the goal is reached, subscribing to the `/move_base/status`, the value of the "status" read changes its value from `1` to `3` and the tomatoes are spawned in the fruit box (tomatoes are also spawned during the navigation between goals). After this, a new goal, read from the .json file that contains all the goals, is sent to the planner. When the box is at the maximum of its capacity (4.5 kg), the checkpoint coordinates, published on the `move_base_simple/goal` topic, are sent and the robot goes to the set checkpoint. The element "pose" is also passed to the function in Python, so that, when the check-point is reached, the function sends the successive goal relative to the last one before the break.

It has been also implemented a different task in which the robot has to come back to the checkpoint when the random counter vanishes. When the node is run, the first pose is sent only after a random timer is initialized. Once the goal is reached, the value of "status", read from `/move_base/status`, changes its value from 1 to 3 and a new goal is sent to the planner.

This scheme is adopted to prevent the sending of a new goal before the achievement of the previous one: in this case there could be problems with the correct tracking of the required path.

When time runs out, the `contr_timer()` function interrupts the sending of new goals from the path and gives to the planner the checkpoint position as seen before; then the timer is set to 0 and the task restarts.

2.5 Multiple node variant with custom messages

For both the tasks described it has also been created a multiple-node version with the creation of custom nodes and messages.

- One node handles the actions -goals sending- and -goal reached-.
- Another node handles the timer or the weight of the fruit box.

To accomplish this purpose two messages have been created:

- A Counter message that contains two Boolean values: toBase and Resets.
- A Weight message that contains nFruits(int8) and toBase (boolean) to emulate a counter and a weight scale.

Furthermore two topics of type Weight or Counter have been created depending on the task. In the sending-goals file we subscribe to one topic and publish on the other, as described in the following Figure.

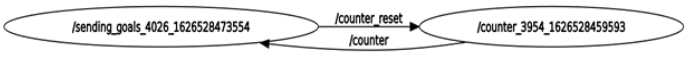

Figure 13: Publisher/Subscriber.

The topic names contain number because it has been set the Anonymous parameter to true to avoid conflicts. So now, when for example it is read the flag toBase as true, the code sends the checkpoint as the next goal. In this way it is possible to better emulate a real use case where there is an external flag sent to a specific topic that asks the robot to go to a goal position (in this case to empty its fruit box). The entire system produces this final graph of nodes and topics:

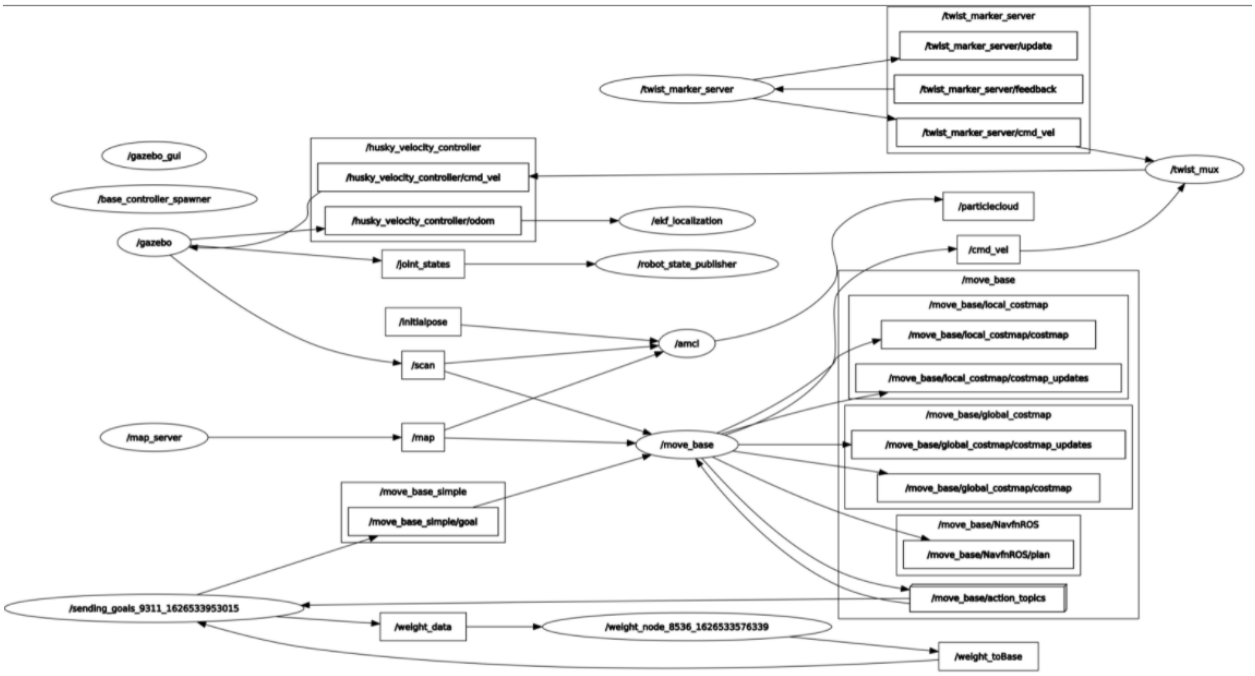

Figure 14: Nodes and topics graph.

2.6 Moving Obstacles

In Gazebo, an animated model is called actor, these extend common models, adding animation capabilities following predefined paths specified by some waypoints. Gazebo's actors are just like models, so it is possible to put links and joints inside them and their trajectories can be scripted directly in SDF model. Within the `<skin>` tag it is possible to use an installed meshes in Gazebo by referencing directly to their filenames, for example "walk.dae", and to synchronize the animation with the defined trajectory it is possible to set "interpolate x" to true inside the tag "animation". These animations are not affected by the physics engine. This means they won't fall due to gravity or collide with other objects. To solve this problem, it has been used the actor-collision plugin [4]. From this GitHub repo (which is the official Gazebo repo) it is possible to download a C program that generates a .so gazebo plugin file. So, it is possible to build and then copy via "`sudocp`" the .so file inside the folder whose path is (`/usr/lib/x86-64-linux-gnu/gazebo-11/plugins/`). Then it has been added inside the .world file the plugin tag to the actor model. In the environment two moving actors have been added in the field, to simulate the presence of farmers between the tomatoes rows.

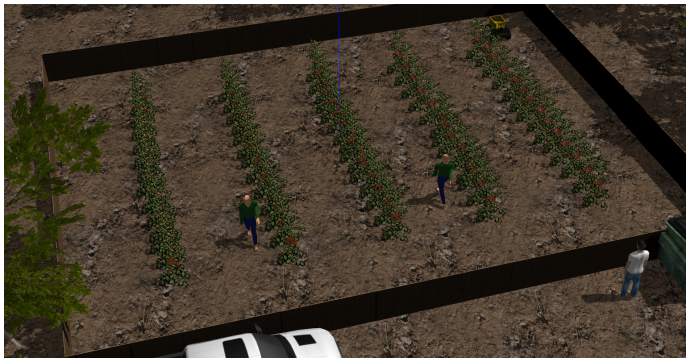

Figure 15: Gazebo World with dynamic obstacles.

3 Simulations and Conclusions

To verify the correct functioning and simulate the execution of the two described tasks, the following command must be executed on different shell windows.

Gazebo:

Launch the Gazebo world with dynamic obstacles:

```
$ roslaunch agromatic_gazebo husky_sim_dynamic.launch
```

Launch the Gazebo world without dynamic obstacles:

```
$ roslaunch agromatic_gazebo husky_sim.launch
```

Launch the Gazebo world without dynamic obstacles, and with the water tank:

```
$ roslaunch agromatic_gazebo husky_sim_water_tank.launch
```

RViz:

Launch the Rviz view of the robot:

```
$ roslaunch husky_viz view_robot.launch
```

AMCL/gmapping:

Launch Gmapping:

```
$ roslaunch husky_navigation gmapping_demo
```

Launch AMCL for static and dynamic obstacles:

```
$ roslaunch husky_navigation amcl_dynamic_demo.launch
```

Launch AMCL for static-only obstacles:

```
$ roslaunch husky_navigation amcl_demo.launch
```

Keyboard teleop:

To tele-operate the robot with the keyboard.

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Simulation nodes:

To save waypoints in a .json file:

```
$ rosrun sending_goals salva_punti.py
```

To run tomato harvesting simulation in a single node:

```
$ rosrun sending_goals sg_spawn_tomato.py
```

To run tomato harvesting simulation in a multiple nodes (weighter and controller), run in different terminals:

```
$ rosrun sending_goals weight_update.py
$ rosrun sending_goals sg_spawn_tomato_node.py
```

To run irrigation/fertilization simulation in a single node:

```
$ rosrun  sending_goals sending_goals_timer.py
```

To run tomato irrigation/fertilization in a multiple nodes (counter and controller), run in different terminals:

```
$ rosrun sending_goals counter.py
$ rosrun sending_goals sending_goals_timer_node.py
```

In the following Figures are shown the task's execution of the tomatoes collection in the case in which there are only static obstacles represented by the tomatoes plants.

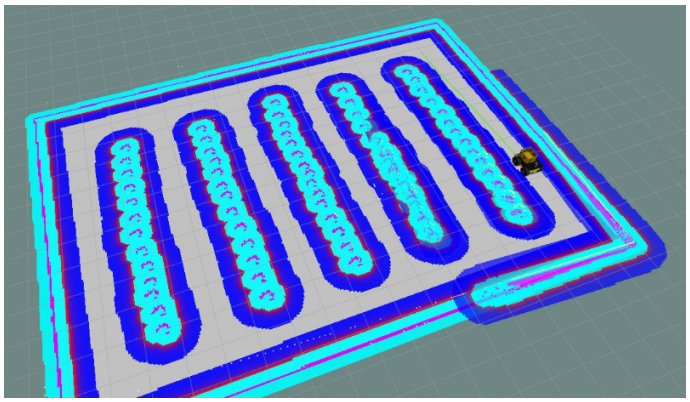

Figure 16: Rviz view in the static case.

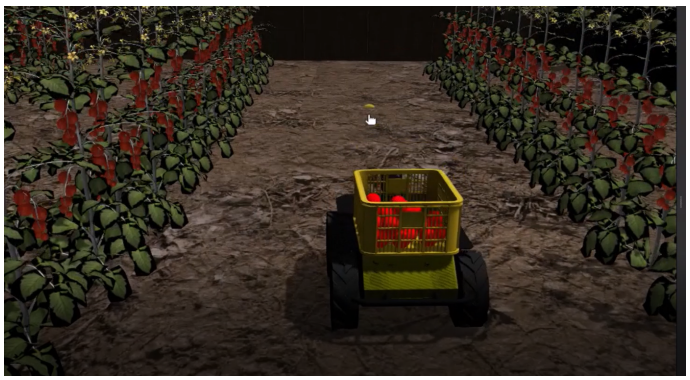

Figure 17: Simulation in Gazebo in the static case.

In the following Figure there is the performing of the unloading of the tomatoes at the checkpoint when the weight of the fruit-box reaches 4.5 Kg.

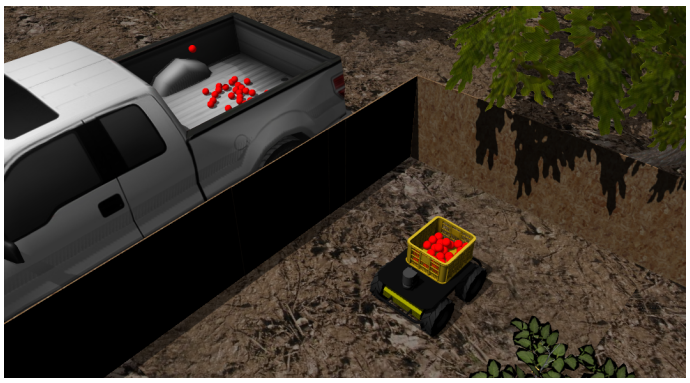

Figure 18: The unloading of the tomatoes.

In the following Figures is represented the case in which also the moving obstacles are present. It is possible to see how, during the navigation, when the robot perceives the presence of a moving actor it tends to stop and change its direction to avoid it.

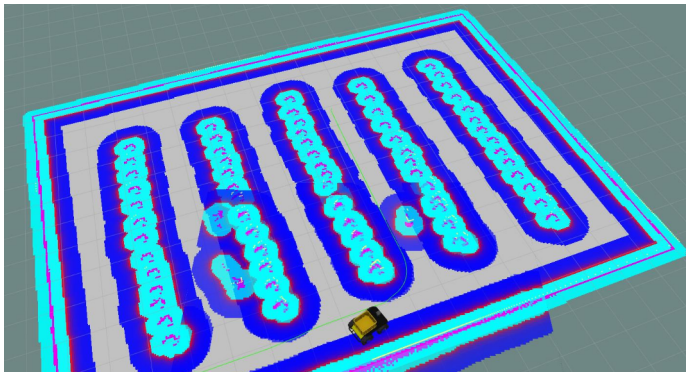

Figure 19: RViz view in the dynamic case.

Figure 20: Simulation in Gazebo in the dynamic case.

In the following Figures is shown the execution of the irrigation/fertilization task in which the robot has on the top a tank and, after a random time, the robot has to come back to the checkpoint.

Figure 21: Performance of tank task.

Ideally, at the checkpoint the robot has to fill the tank.

Figure 22: Filling of the tank.

A Guide for installation and use

To install and try the work presented in this project, the user can clone the GitHub Repository **linoe97/Agromatic** inside his `~/catkin_ws/src` folder. Make sure to have ROS installed and a `/catkin_ws` folder in your home directory with an `/src` folder.
Download and extract the repo from GitHub or use the following command :

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/linoe97/agromatic.git
```

Then, copy the custom Gazebo models inside user `~/.gazebo/models/` folder:

```
$ sudo cp agromatic/models/* ~/.gazebo/models/ -a
```

Same thing, for the file `libActorCollisionsPlugin.so`, the gazebo plugin file that implements collisions for actors, that we need to copy inside the plugin folder of gazebo:

```
$ sudo cp agromatic/plugins/* /usr/lib/x86_64-linux-gnu/gazebo-11/plugins
```

Then make and build the project:

```
$ cd .. && catkin_make
```

References

[1] *Clearpath Robotics website*. URL: https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/#downloads.

[2] *DWA Local Planner*. URL: http://wiki.ros.org/dwa_local_planner.

[3] *TEB Local Planner*. URL: http://wiki.ros.org/teb_local_planner.

[4] *Actor collision plugin*. URL: https://github.com/osrf/gazebo/tree/gazebo11/examples/plugins/actor%20-collisions.