Where Am I ?

Miguel A. Colmenares Barboza

Abstract —**The project presented in this document focused on creating a robot model with the unified robot description format (URDF). The robot model has a base, two wheels as actuators, a laser sensor and a camera. Said robot is placed in a Gazebo simulation environment with a training map and is asked to go to a goal. The standard ROS navigation stack and a Monte Carlo installation package are the robots for the robot that are on the map, that is, the robot is searched for where it is. Additionally, the Rviz program is used to know the readings of the sensors and how the robot interprets its surroundings.**

Index Terms —Robot, IEEEtran, Udacity, \LaTeX, deep learning.

I. INTRODUCTION

At the moment of developing a robot, one of the most common and important characteristics, is to make it an autonomous robot, that is, a robot that makes its own decisions. If the robot is mobile and can move from point A to point B, a decision must be made in its direction. The robot needs to answer the question Where do I go? and for this, you must first ask Where am I? Answering this last question is what is known as the **Location Problem** and is a common problem when building a robot.

The amount of information present and the nature of the environment of the robot, determine the difficulty of the task of locating the robot. The **Location Problem** is defined by more factors, but mainly there are three different types of location problems.

One of the most common localization problems is **local localition**. In this problem, the robot knows its initial position and the problem of location is in estimating the position of the robot as the robot moves through the environment.

A more complicated location problem is name **global location**. In this case, the initial position of the robot is unknown, therefore the robot must determine its position relative to the map of its environment. The amount of uncertainty in the global location is much greater so it is a much more difficult problem.

A more **challenging location** problem is the kidnapped robot problem. This problem is similar to the global location problem except that the robot can be hijacked at any time and moved to a new location on the map. It is considered the worst case in the subject of location.

This document presents a project that seeks to address this particular issue. First the steps to create the model of a robot using URDF are developed and then that robot is placed on a Gazebo map and given a goal. The robot will use the reading of a laser sensor and a camera to know where it is with respect to the map of its environment.

II. BACKGROUND.

As mentioned above, for a mobile robot it is important to know where it is to make the necessary decisions to reach a goal or to make other decisions if the robot is already at the goal.

There are several factors that define the difficulty of location for the robot but there are three most common factors. The first factor to consider is the map or environment of the robot that can be static where the environment does not change or can be dynamic where the environment changes constantly.

To get an idea of the environment, there is a range of sensors that the robot can use. Each sensor takes a measurement or reading of the environment and then these values converge on the robot code to determine a point of location. But the problem at this point is that every measurement or reading of the sensors contains noise and the more sensors are used, the more noise enters the system.

For this reason the noise of the sensors is the second factor that defines the difficulty of location.

And as a third factor, it is whether or not the robot can be hijacked and moved to another location.

If the environment is static and known, giving a map of the environment to the robot reduces the difficulty of localization by having a reference to lean on. And for the problem of robot hijacked, the robot only has the reading of its sensors to know where it is. This means that to get a good location the most important thing to develop is the way to process the sensor readings and filter the noise that comes with the readings. In this way, the location problem is mainly focused on the filtering of the sensor readings.

To solve the problem of location, here are the two most common location algorithms to be implemented in the robot.

The Kalman filters is a very prominent estimation algorithm in the controls. It is used to estimate the value of a variable in real time as the data is collected. The Kalman filter can take data with a lot of uncertainty or noise in the measurements and provide a very accurate estimate of the real value and it does it very fast.

Each time a measurement is recorded, the result is a two-step process in a continuous iteration.

The first step is an update of the measurement. And the second step is a state prediction. The current state information is used to predict the future state.

Then the Kalman filter operates under the following assumptions:

- The functions of the movements and the measurements are linear.
- The state of space is represented by a unimodal Gaussian distribution.

These assumptions are very limited and work well only for very simple robots. Most public robots mainly execute non-linear movements, such as moving in circles or following a curve. When a unimodal Gaussian distribution is subjected to a non-linear transformation, the resulting distribution is not a Gaussian distribution and can not be calculated in a closed manner. This situation means that the Kalman filter can not be applied to most robotics problems.

To solve this situation, there is an extended version of the Kalman filter, which seeks to linearize a movement function or non-linear measurement.

The Monte Carlo location filter is another algorithm based on sampled results. The Monte Carlo localization algorithm (MCL) is the most popular localization algorithm in robotics. MCL uses sensor measurements to track the position of the robot that uses particles to locate the robot. The particles, the virtual elements that resemble the robot. Each particle has a position and orientation, and represents an assumption of where the robot could be located. The particles are shown again each time the robot moves and detects its surroundings. Something important to consider is that the MCL algorithm is limited to global and local location problems.

The MCL algorithm, to use particles; It can be any distribution function that is not limited by a Gaussian and non-linear model. MCL is also a simpler algorithm in calculation and implementation compared to the Kalman filter algorithms. And MCL is also a good choice when you need to control the memory and resolution of a device, which may be necessary for small integrated computers with limited resources.

III. RESULTS.

To test the navigation of the robot, the following commands are executed:

```
Terminal #1: $ roslaunch udacity_bot xxx.launch

Terminal #2: $ roslaunch udacity_bot amcl.launch

Terminal #3: $ rosrun udacity_bot navigation_goal
```

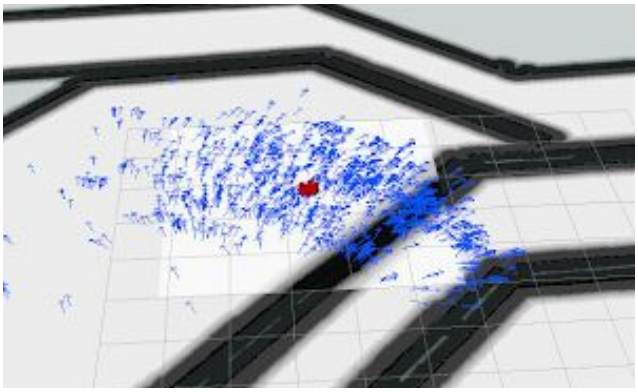
In Terminal # 1, a command is used that starts the Gazebo program that contains the model of the robot and the world for the simulation and also initiates the Rviz program which manages the characteristics of the robot, such as the readings of the sensors and the map of the robot. environment.

In Terminal # 2, a command that starts the amcl.launch node is used which contains the parameters and configuration of the filter that the robot implements.

In Terminal # 3 a command is used that gives the robot a goal to go to.

Result of the Standard Robot of the Class.

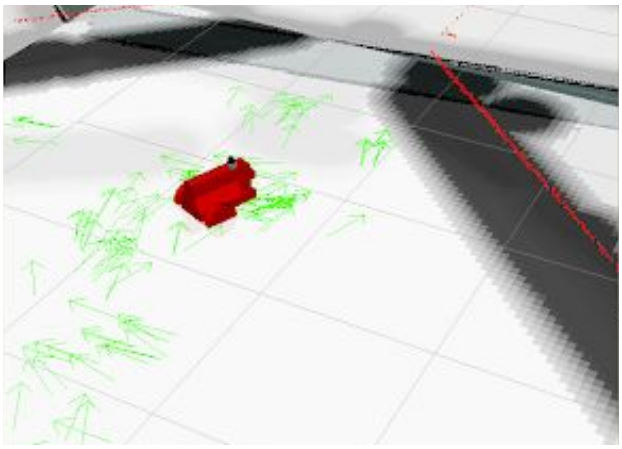
Running. `$ roslaunch udacity_bot udacity_world.launch`



Robot Udacity_bot in the goal

Result of the Custom Robot (My UchoBot).

Running. `$ roslaunch udacity_bot ucho_model.launch`



Robot UchoBot in the goal.

IV. CONFIGURATION.

Format of unified robot description (URDF).

The ROS URDF package contains a C ++ analyzer for the unified robot description format (URDF), which is an XML format to represent a robot model. This package contains a series of XML specifications for robot models, sensors, scenes, etc. Each XML specification has a corresponding analyzer in one or more languages.

To create the robot model, the `<Link>` tags are defined, which represent the robot blocks (head, trunk, wheels, etc.) and the `<Join>` tag that specifies the union of one link with another and at what point the link is located the join. Within the `<link>` tag, three other `<inertial>, <visual>` and `<colission>` tags are defined in which important parameters are described for the simulation, such as the dimensions and origin of the elements, type of element (sphere , cylinder or cube) and more features.

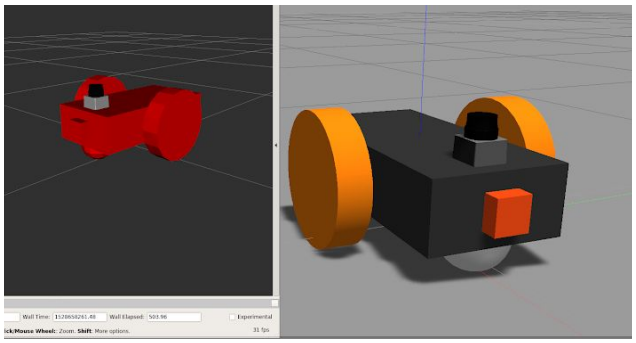
Both robot models have a base block, two wheels, a camera and a laser sensor.

The standard package of selected sensors were:

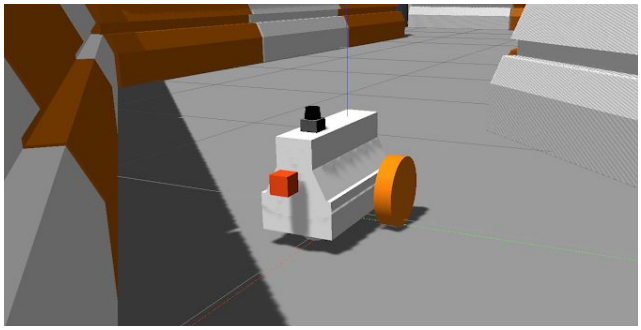
- **Laser Scanner (Hokuyou)**, placed at the top of the robot to have a free space of 180 degrees in front of the robot.
- **Camera**, placed in front to see what is in front of the robot. (In this work the sensors of the camera are not used for navigation purposes).

The biggest changes and differences are found in the dimensions of the base block and the location of the sensors.

View of the models:



Model of Robot Udacity.



Model of Robot Custom.

Configuration AMCL.

The adaptive localization algorithm of Monte Carlo uses a variable amount of particles depending on the certainty which helps to free up computational resources.

```
<param name = "min_particles" value = "50" />
<param name = "max_particles" value = "2000" />
```

The longevity of the transformation was established in 0.3 seconds.

```
<param name = "transform_tolerance" value = "0.3" />
```

The robot starts at the center of the map, so its pose was set [0, 0].

```
<param name = "initial_pose_x" value = "0.0" />
<param name = "initial_pose_y" value = "0.0" />
<param name = "initial_pose_a" value = "0.0" />
```

Parameters related to the odometry were configured where values for the noise of the movement of rotation and translation are specified.

```
<param name = "odom_alpha1" value = "0.002" />
<param name = "odom_alpha2" value = "0.002" />
<param name = "odom_alpha3" value = "0.002" />
<param name = "odom_alpha4" value = "0.002" />
```

And parameters were configured for the laser sensor.

```
<param name="laser_max_range" value="2" />
<param name="laser_min_range" value="2" />
<param name="laser_max_beams" value="20" />
<param name="laser_z_hit" value="0.95" />
<param name="laser_z_rand" value="0.05" />
```

Configuration Move_Base.

The base move configuration includes the configuration of the common parameters and the parameters of the local and global map.

For the common parameters, values that work well for the two robots were selected.

`footprint: [[-0.22, -0.20], [-0.22, 0.20], [0.22, 0.20], [0.22, -0.20]]`

`transform_tolerance` was configured according to the configuration of AMCL(0,3).

`inflation_radius` it was reduced from the default value of 0.55 to the closest of the robot's shape (0,4).

For the local cost map, the size of the map was reduced to 5 meters because it solved the problem when the robot moves in the opposite direction to the global route in an attempt to reach the target through the wall.

In `local_costmap_params.yaml:` width: 5.0 - height: 5.0

Parameters are defined as `min_vel_x`, `min_in_place_vel_theta` and `escape_vel` to overcome the friction and inertia of the mass.

It is useful to set target tolerance values to stop the robot's rotation and move around the goal so as not to overstep continuously. Target tolerance parameters:

```
xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.05
```

In `base_local_planner_params.yaml`:

```
max_vel_x: 0.5
min_vel_x: 0.2
max_vel_theta: 1.0
min_in_place_vel_theta: 0.3
escape_vel: -0.15
```

V. DISCUSSION.

After developing two robot models and testing them in a simulation environment, the following observations are considered.

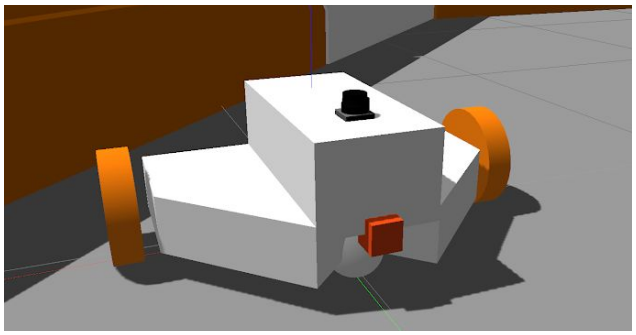
Both models are very similar. Both models have a base body, two wheels, a camera and a laser sensor of the same model.

The biggest differences are in the maximum number of particles for Udacity_Bot is 2000 and the maximum number of particles for UchoBot is 200.

The smaller number of particles consumes less computer resources but generated greater uncertainty about the position of the robot. In the images of the results you can see in the robot of Udacity_bot a group of blue arrows with certain direction in common and on the contrary in the image of the personalized robot Uchobot several green arrows scattered on the map can be seen.

The other differences between both models are in the definition of the configuration parameters, where it will be appreciated that Uchobot is configured slower than Udacity_bot.

Before developing the UchoBot robot model, another model called Colosus was developed.

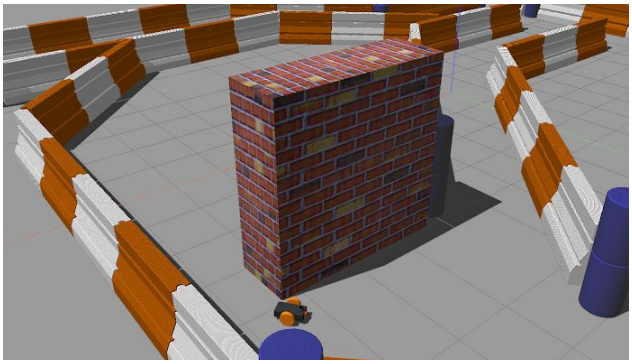


This model looks good but, when giving it a goal, it did not advance more than a few meters before hitting the wall in front of it. His sensors told him there was no wall in front and that he could continue. Thanks to Colosus it was possible to understand how sensitive the configuration parameters are, as well as the sensors themselves.

On one occasion, the same base body was considered an obstacle in some tests because of the configuration and location of the laser. Therefore, it was decided to use a smaller base model.

Another problem presented by Colosus was the problem of presenting a roll or tilt backwards when starting the forward march. this changed the position of the laser and the camera and disoriented the robot. A physical problem that was resolved by placing a bar on the bottom of the bot. It is not recommended but I help to get out of the way. In addition, said bar does not affect the bot's abilities to locate itself.

After getting good results, in the simulation it was tried to place a wall of bricks to see how the robot would react. This scenario was only tested with Udacity_bot.



A new brick wall for the bot.

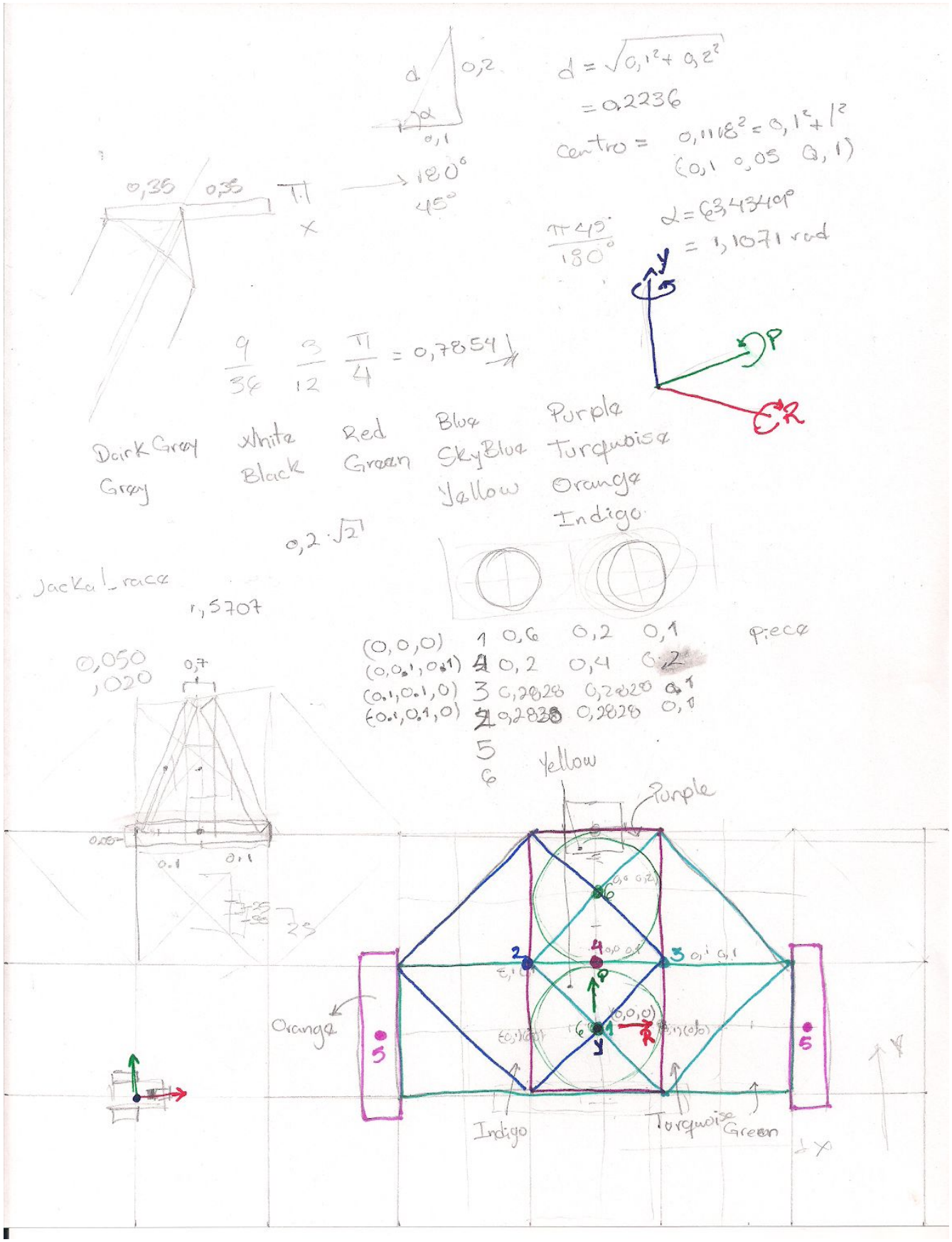
In this test it was observed that the bot was able to reach its new goal but was left disoriented, assuming a position on the map that was incorrect.

It is logical to think that something like this would happen. The code implemented in the bot was designed for a static environment and by putting a new element on the map, the environment changed and became a dynamic environment.

Many useful parameters were reversed after long hours of experimentation because the predetermined values showed better results than any adjustment.

The problem of a hijacked robot is not solved with the AMCL algorithm because if the robot is hijacked and moved to an area without particles, it could not recover its location.
Different methods of sequestration event were proposed in the literature, such as the distribution of weight particles or the entropy of information that could be extracted from the weights. Although as a personal comment the best solution for a location problem is to look for a reference point. Seeing how far you are and from which position the reference point is seen can give an initial idea of the current position. Even if you do not get the reference point you can have an idea of where you are.

The Monte Carlo location can be used for a home robot with the familiar map of the house or office. However, people moving and furniture of the place would pose problems in the location due to the changing map, although this could be solved by constructing dynamic maps of the environment.



Sheet with a series of annotations concerning the development of the model in URDF.

VI. FUTURE WORK.

For future work there is still a lot of ground to cover. There are still many pending details.

For example, there is still a wide range of sensors to be tested and similarly there are other algorithms that can be implemented. There are even other modes of locomotion that the bot can use to move from A to B, which would change the entire structure of the code (How to use an omnidirectional locomotion).

For the current configuration of the robot, the laser sensor of 180 degrees is sufficient to cover the solution of the location of the robot, it is only necessary to adjust and tune the configuration parameters better to reduce the error. Although as future work you can also consider looking for a sensor that gives better results.

It is important to keep in mind that the problem of localization depends on the needs of the developer. Locating the bot may or may not be important, the accuracy of the algorithm may or may not be important, the number and types of sensors that will be implemented may or may not be important. So it will be these factors and many others that define the way to approach and solve the problem of location of the robot.

Compensations in precision and processing time.

There are different ways in which you can increase the accuracy simply by changing the system parameters:

1) Number of particles in the AMCL filter: More particles give more possibilities to correct and locate the robot in unexpected places. But it takes more time to calculate.

2) Increasing the resolution for the global and local cost maps provides a greater granularity to have a more accurate plane. Again, with more cells to calculate and find a path, the computational load on the system is increasing.

3) It could also increase the number of sensors that take samples of the environment at the same frequency but with a 90-degree offset. In this way, one sample could be processed at a time and the resources managed better

In summary, there is a great control over efficiency and computational precision that is useful for implementation in smaller systems.