NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Robot Collaborative content exploration and path planning

Christina F. Katsarlinou

Supervisor: Stathes P. Hadjiefthymisades, Professor

ATHENS

OCTOBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Συνεργατική εξερεύνηση περιεχομένου και σχεδιασμός μονοπατιών ρομπότ

Χριστίνα Φ. Κατσαρλίνου

Επιβλέπων: Ευστάθιος Π. Χατζηευθυμιάδης, Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2020

BSc THESIS

Robot collaborative content exploration and path planning

Christina F, Katsarlinou

S.N**.:** 1115201500068

Supervisor: Stathes P. Hadjiefthymiades, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Συνεργατική εξερεύνηση περιεχομένου και σχεδιασμός διαδρομών ρομπότ

Χριστίνα Φ. Κατσαρλίνου
Α.Μ.: 1115201500068

Επιβλέπων: Ευστάθιος Π. Χατζηευθυμιάδης, Καθηγητής

ABSTRACT

In recent years, the rapid development and evolution of the Internet of Things (IoT) and robotics seems unstoppable. The new possibilities added to the nodes, open horizons for new research as well as for new uses in the daily life of people and industry. One of the key features is the mobility of the nodes. Most nodes are no longer static, but move in space and offer a wide range of new applications that can offer like the ability to make decisions without human intervention, their durability, the use of embedded sensors (temperature, pressure, humidity, etc.), as well as for their reprogrammability. Based on these features, mobile nodes can be used for example in cases of surveillance of areas and borders, for image recognition and alarm signaling, as well as for crisis management. For example, an unmanned land vehicle (mobile node) carrying a high-definition sonar and a high-definition thermal camera, combined with an object recognition algorithm can be used to find people trapped in wreckage.

In addition, this functionality can be enriched by the fact that two or more nodes can communicate with each other to work together to complete a mission. Let us consider a mission to find a lost hiker in a forest, with a single mobile node (unmanned aerial vehicle), the chances of finding him in a short time are much lower than when we have more than one to communicate with, exchanging images, measurements, the areas they have scanned, and finally if any of them have found the target. This group mode in the context of the Internet of Things and robotics is called a swarmi of nodes. More specifically, each node operates based on the knowledge of the whole team and not individually. This is also observed in nature, especially in insects, where they function on the basis of this method.

In this thesis, it is examined whether the swarm operation is more efficient both temporally and qualitatively in relation to the operation of each node as independent on collaborative search of sensor targets with no prior knowledge of the environment. More specifically, a series of experiments are carried out where two robots scan the entire space in detail exhaustively in order to identify the points where the value from an existing sensor sources is maximum. These values are detected by robots with the help of sensors that they carry. In the first case, the robots act independently without knowing neither the measurements taken by the other, nor its position. In the second case, the robots cooperate based on the operation of the swarm, in order to find the optimal possible value of the source.

The experiments were supported by the Ubuntu 16.04 operating system, the Gazebo and Rviz simulators, as well as two virtual TurtleBots running the ROS operating system, as well as a virtual XBOX Kinect sensor with a color camera and a depth sensor.

SUBJECT AREA: Context exploration

KEYWORDS: collaborative, context exploration, path planning, ROS, turtlebot, Particle Swarm Optimization

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια, η ταχεία ανάπτυξη και εξέλιξη του Διαδικτύου των πραγμάτων (IoT) και της ρομποτικής φαίνεται ασταμάτητη. Οι νέες δυνατότητες που προστέθηκαν στους κόμβους, ανοιχτούς ορίζοντες για νέα έρευνα, καθώς και για νέες χρήσεις στην καθημερινή ζωή ανθρώπων και βιομηχανιών. Ένα από τα βασικά χαρακτηριστικά είναι η κινητικότητα των κόμβων. Οι περισσότεροι κόμβοι δεν είναι πλέον στατικοί, αλλά κινούνται στο διάστημα και προσφέρουν ένα ευρύ φάσμα νέων εφαρμογών που μπορούν να προσφέρουν, όπως η ικανότητα λήψης αποφάσεων χωρίς ανθρώπινη παρέμβαση, η ανθεκτικότητά τους, η χρήση ενσωματωμένων αισθητήρων (θερμοκρασία, πίεση, υγρασία κ.λπ.) , καθώς και για τον επαναπρογραμματισμό τους. Με βάση αυτά τα χαρακτηριστικά, οι κινητοί κόμβοι μπορούν να χρησιμοποιηθούν για παράδειγμα σε περιπτώσεις επιτήρησης περιοχών και συνόρων, για αναγνώριση εικόνας και σηματοδότηση συναγερμού, καθώς και για διαχείριση κρίσεων. Για παράδειγμα, ένα μη επανδρωμένο χερσαίο όχημα (κινητός κόμβος) που φέρει ένα σόναρ υψηλής ευκρίνειας και μια θερμική κάμερα υψηλής ευκρίνειας, σε συνδυασμό με έναν αλγόριθμο αναγνώρισης αντικειμένων μπορεί να χρησιμοποιηθεί για την εύρεση ατόμων που έχουν παγιδευτεί σε συντρίμμια.

Επιπλέον, αυτή η λειτουργικότητα μπορεί να εμπλουτιστεί από το γεγονός ότι δύο ή περισσότεροι κόμβοι μπορούν να επικοινωνούν μεταξύ τους για να συνεργαστούν για να ολοκληρώσουν μια αποστολή. Ας εξετάσουμε μια αποστολή να βρούμε έναν χαμένο πεζοπόρο σε ένα δάσος, με έναν μόνο κόμβο κινητού (μη επανδρωμένο εναέριο όχημα), οι πιθανότητες να τον βρούμε σε σύντομο χρονικό διάστημα είναι πολύ χαμηλότερες από ό, τι όταν έχουμε περισσότερα από ένα να επικοινωνήσουμε, ανταλλάσσοντας εικόνες , μετρήσεις, τις περιοχές που έχουν σαρώσει και, τέλος, εάν κάποια από αυτές έχει βρει τον στόχο. Αυτή η ομαδική λειτουργία στο πλαίσιο του Διαδικτύου των πραγμάτων και της ρομποτικής ονομάζεται σμήνος κόμβων. Πιο συγκεκριμένα, κάθε κόμβος λειτουργεί με βάση τις γνώσεις ολόκληρης της ομάδας και όχι μεμονωμένα. Αυτό παρατηρείται επίσης στη φύση, ειδικά στα έντομα, όπου λειτουργούν βάσει αυτής της μεθόδου.

Σε αυτή την πτυχιακή εργασία, εξετάζεται εάν η λειτουργία σμήνους είναι πιο αποτελεσματική τόσο χρονικά όσο και ποιοτικά σε σχέση με τη λειτουργία κάθε κόμβου ως ανεξάρτητη στη συνεργατική αναζήτηση στόχων αισθητήρων χωρίς προηγούμενη γνώση του περιβάλλοντος. Πιο συγκεκριμένα, πραγματοποιείται μια σειρά πειραμάτων όπου δύο ρομπότ σαρώνουν λεπτομερώς ολόκληρο τον χώρο λεπτομερώς για να προσδιορίσουν τα σημεία όπου η τιμή από υπάρχουσες πηγές αισθητήρα είναι μέγιστη. Αυτές οι τιμές ανιχνεύονται από ρομπότ με τη βοήθεια αισθητήρων που μεταφέρουν. Στην πρώτη περίπτωση, τα ρομπότ δρουν ανεξάρτητα χωρίς να γνωρίζουν ούτε τις μετρήσεις που έχει λάβει ο άλλος ούτε τη θέση του. Στη δεύτερη περίπτωση, τα ρομπότ συνεργάζονται με βάση τη λειτουργία του σμήνους, προκειμένου να βρουν τη βέλτιστη δυνατή τιμή της πηγής.

Τα πειράματα εκτελέστηκαν σε λειτουργικό σύστημα Ubuntu 16.04, στους προσομοιωτές Gazebo και Rviz, και χρησιμοποιήθηκαν δύο εικονικά TurtleBots που λειτουργούν με το λειτουργικό σύστημα ROS, καθώς και από έναν εικονικό αισθητήρα XBOX Kinect με έγχρωμη κάμερα και αισθητήρα βάθους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αναζήτηση Περιεχομένου

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Συνεργασία Ρομποτ, Αναζήτηση περιεχομένου, σχεδιασμός μονοπατιών, ROS, turtlebot, Particle Swarm Optimization

Η εργασία αυτή αφιερώνεται στην μητέρα μου για την συνεχή και αδιάκοπη στήριξή της καθ' όλη την διάρκεια των σπουδών μου.

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Ευστάθιο Χατζηευθυμιάδη, που μου έδωσε την ευκαιρία να ασχοληθώ με τον τομέα της ρομποτικής και με εμπιστεύτηκε με το θέμα της παρούσας πτυχιακής. Επίσης θα ήθελα να ευχαριστήσω την διδάκτωρ Κυριακή Παναγίδη για την συνεχή υποστήριξη και βοήθειά της καθ' όλη την διάρκεια εκπόνησης της εργασίας.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

During the last years a significant evolution has been noticed regarding the Internet of Things (IoT). IoT refer to the network of physical objects that have sensors and software and connect with each other, via the Internet, in order to collect and exchange information. Nowadays everything can be turned into a part of the IoT. The connection of multiple different devices with sensors on them lead to an IoT with a great level of intelligence. In this thesis, we will focus on unmanned vehicles, UxVs, as parts of IoT. UxVs (UxVs- x stands for different type of environment, i.e. ‘s’ for sea, ‘a’ for air and ‘g’ for ground) are mobile nodes that can navigate in the environment an react in specific events. The key characteristic of the UxVs is the autonomous decision making without human intervention. Additionally, some other UxVs’ capabilities are endurance, multimedia streaming and payload carrying. Rapid technological advancements that occurred in the last decade expanded the possible uses of UxVs. Some of the most recent use cases are surveillance, security monitoring, and supporting crisis management activities. For instance, UGV with a thermal camera can locate, recognize and possibly rescue an errant hiker.

In several missions, the search area cannot be covered by a single device or the time is a critical factor therefore the researchers need to find efficient methods to overcome these barriers like the collaborative operation of the vehicles. During a collaborative operation￼ multiple UxVs communicate with each other in order to complete efficiently and successfully a same mission. Additionally, for the collaboration of the unmanned vehicles, the Particle Swarm Optimization theory will be studied in this Thesis. Swarm technology is inspired by swarm intelligence, which draws inspiration from the lives of social insects or birds ￼Swarm intelligence (SI) provides the possibility of SI behavior through collaboration in individuals that have limited or no intelligence [1]. Its potential parallelism and distribution characteristics can be used to realize global optimization and solve nonlinear complex problems. Swarms of UxVs

The goal of this Thesis is to examine if that collaboration through SI is more time and resource efficient than just letting the robots execute the mission extensively without communicating with each other. To achieve this goal, we use UGVs that aim to locate some sensor sources that are located at unknown positions in their ‘world’. The sensor sources transmit measurements that UGV sensors can collect. For instance, in real life missions, these sources could be a source of a fire or an SOS signal indicating a human trapped under the ground. Firstly during our research, we develop an algorithm that scans exhaustively the environment using multiple UGVs. The UGVs will not communicate with each other regarding the environment exploration for points of interest. The exploration of the environment is based only on the local knowledge of each UGV’s sensor’s measurements. In addition, we develop an algorithm for collaborative operation of UGVs based on PSO theory. In this case multiple UGVs communicate with each other, via a master, and they constantly know each other’s status. This knowledge can lead to a faster context discovery rather than exhaustively scan an unknown world. The results are presented in section X based on different case studies comparing the time and utility efficiency of the both methods.

2 BASIC ELEMENTS OF UNMANNED VEHICLES

2.1 Definition of Unmanned Vehicles

An unmanned vehicle is defined as a vehicle that operates with no person in it. The control of the unmanned vehicles can be guided by the commands from a remote source. For example, a person in a computer room could send commands through a message bus like Kafka. An unmanned vehicle can also operate autonomously by running special designed algorithms. In this Thesis we will analyze the second type of unmanned vehicles.

The autonomous vehicles are capable of sensing their environment either with the use of sensors, or the use of maps that describe the environment. The use of a map, however, requires previous knowledge of the environment, meaning that the unmanned vehicle or some other device has to explore the world and create the needed map before the unmanned device can operate autonomously in this environment. On the other hand, unmanned vehicles can explore their environment with a variety of different sensors, such as cameras, lidar and temperature sensors. In the last years many scientists are trying to develop new algorithms or optimize existing ones for unmanned vehicles to act autonomously and explore their environment with sensors. In this Thesis we will run experiments in which we do not have previous knowledge of the world or the obstacles that it might contain. The navigation and the obstacle avoidance is based only on sensors of the unmanned vehicle.

The types of the unmanned vehicles are UAVs, meaning unmanned aircraft aerial vehicles commonly known as drones, UGV, that are unmanned ground vehicles and USV, unmanned surface vehicles, also known as surface drones that operate on the surface of the water.

2.1.1 Unmanned aerial vehicle

Unmanned aerial vehicles (UAV) [2], commonly known as drones, are aircraft that do not have a human pilot boarded. The UAV’s flight can be fully autonomous, or controlled from a human operator usually on the ground. The use of drones was initially aimed to help the military missions, however nowadays drones are used for plenty of reasons, to simplify many professions or even as games for children. In Figures 1 and 2 examples of UAVs are shown.

Figure 1 - Drone (UAV) used for photography and videos

Figure 2 - Military UAV

2.1.2 Unmanned ground vehicles

Unmanned ground vehicles (UGVs) operate on the ground with no human on board. The operation of the vehicle can be controlled by human actors remotely or it can be autonomous due to specifically implemented algorithms. The use of the UGVs is considered necessary in cases where the human presence is dangerous. UGVs are used in many professions. For instance the exploration of Mars would have been impossible without the specially designed UGV. In Figure 3 the UGV used in Mars exploration is shown.

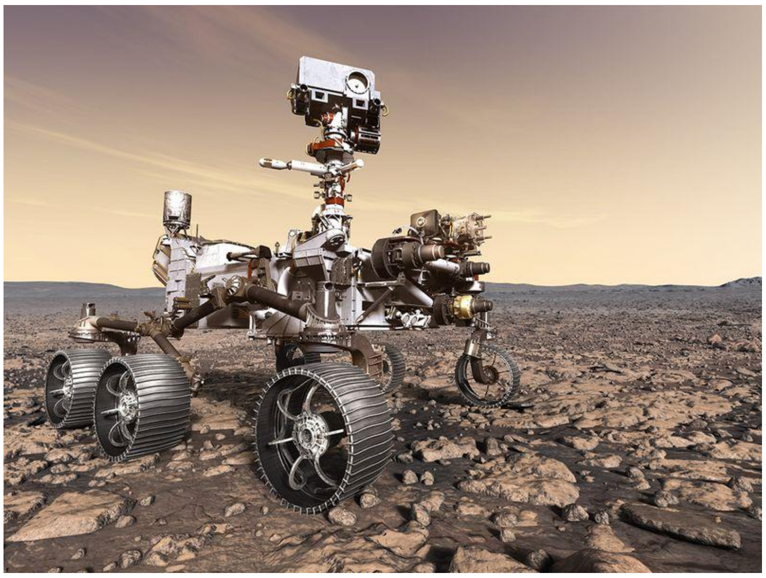


Figure 3 - UGV used in Mars exploration

Nowadays scientists all over the world focus their study on unmanned cars that will work autonomously and will be used for transportation as a normal car. The initial idea in 1921 was to construct a remotely controlled car and during the last years the idea became to create a self driving car which will be used for transportation of humans. The safety of humans on board a self driving car is of great importance. Hence the development of the algorithm especially for navigation must be well designed. For these algorithm sensors like lidar, gps and cameras are necessary.

As most unmanned vehicles, UGVs are used for military purposes as well. A military UGV is shown in the following figure (Figure 4).

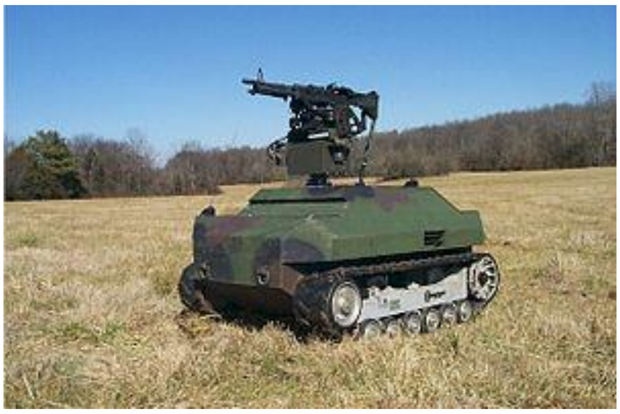


Figure 4 - military UGV

2.1.3 Unmanned surface vehicles

Unmanned surface vehicles (USVs) [3] are boats that operate autonomously without humans on board. In most cases USVs are controlled remotely by a human actor that usually is on the ground. USVs can be used in military missions, in oceanography and seaweed farming. In Figure 5 an example of a USV is shown.

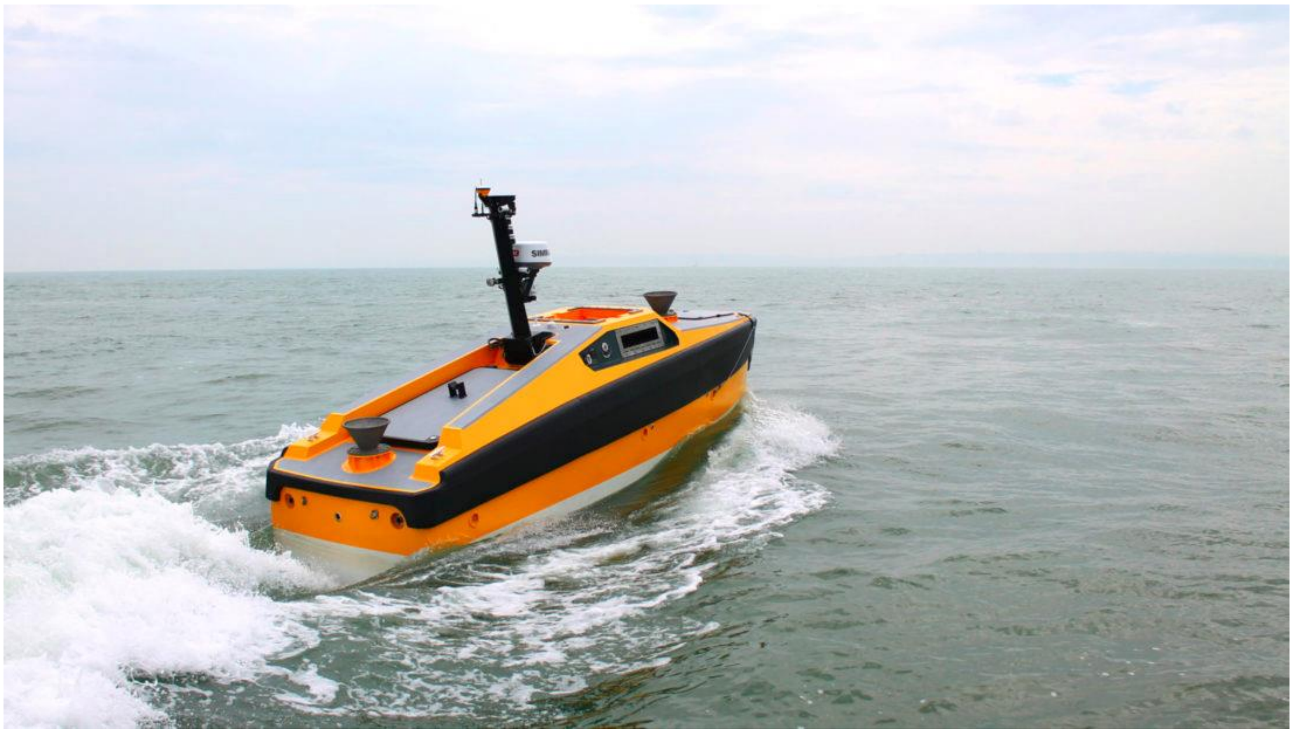


Figure 5 – Unmanned Sea Vehicle

2.2 Robotic Operating System – ROS

The progress noticed in the robotics community during the last years is impressive, both at hardware and software development. Nowadays, more reliable and inexpensive robot hardware is widely available. Many efficient algorithms have been developed that provide the robots with a certain level of autonomy. Even if a significant progress has been noticed, software development has still many challenges to deal with. The most crucial challenges have to do with the distribution of computations, the testing necessary to check the correctness of an algorithm and the reuse of the code. A proposed solution to these difficulties is a software platform called Robot Operating System (ROS).

2.2.1 Definition of ROS

Robot Operating System (ROS), is a Linux based software framework for operating robots [4] [5]. It is an open-source, meta-operating system, which means that it runs alongside with the operating system. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers [4].

ROS [5] s a collection of tools, libraries and conventions that aims in the simplification of the procedures for creating complicated and robust robot behavior across a wide variety of robotic platforms. It is important to highlight that ROS, as it is not an operating system, has to work alongside with a traditional operating system like Linux.

The main advantages of ROS are:

1 Distributed computation: ROS provide a simple and reliable mechanism for communication between multiple processes on the same or on different computers. This feature is necessary for robot systems' software, considering the fact that in most cases these algorithms span many processes that run in different computers.

2 Software Reuse: Due to the progress of robotic research, many effective algorithms have been developed for most of the common tasks of a robot (e.g. navigation, motion planning, mapping). ROS's standard package provides implementations of the most important algorithms of these tasks. In addition, several ROS packages with multiple useful algorithms are publicly available. All these can lead to even more rapid progress in robotics because developers can focus on new ideas and not spend their energy reinventing the wheel.

3 Rapid Testing: Most of the times, testing a robot's algorithm is time consuming and error-prone. Sometimes a physical robot is not available, or even if it is, running an experiment with it might be slow or expensive. Also testing extreme cases might be tricky. For these cases, ROS provide mechanisms that improve the testing process. Firstly, there are simulators specifically developed for running robot algorithms. ROS also provide a simple way of recording data. With these features the aforementioned issues are resolved and the testing process becomes easier and less time consuming.

ROS is not the only platform that has been developed to serve these purposes. The reasons that lead us and many other software developers to choose ROS over the other platforms for software environments are plenty. All these reasons come from the fact that ROS differs from other platforms because the robot community widely supports it. Nowadays, commercial companies develop their product in a way that they will be compatible with ROS. ROS uses the concept of packages, nodes, topics, messages and services.

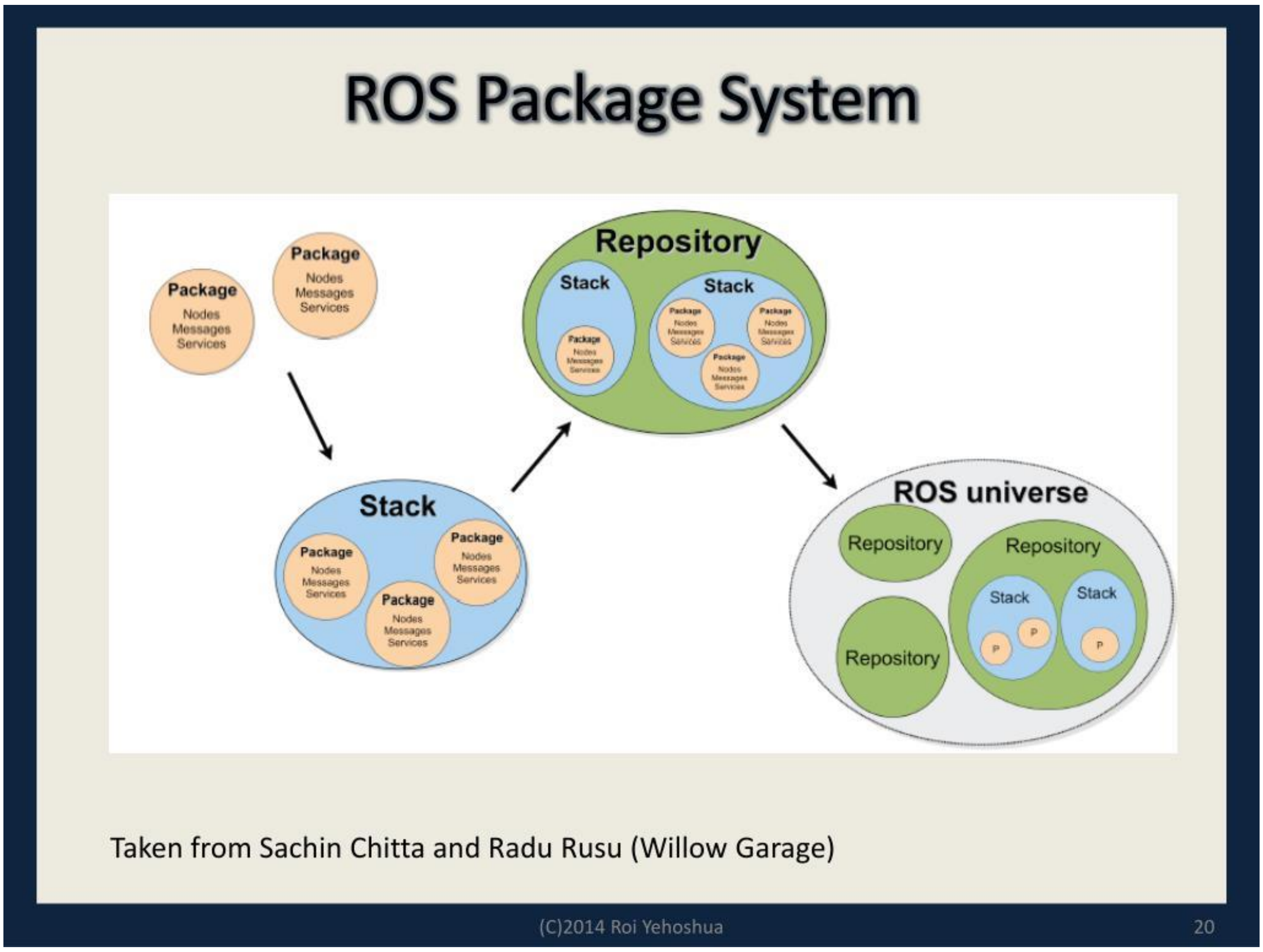


Figure 6 - ROS Package System

2.2.2 ROS Packages

ROS software is separated into packages. A package is folder that contains multiple files that serve a specific purpose. Such files are executables and supporting files like ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software. Packages are the minimum ROS unit of build and release. As a result, the ROS package is considered as the the smallest structure that a ROS program requires. ROS packages provide in an easy to consume way, very useful functionalities. The also contribute to software reuse. A ROS package must contain only the necessary in order to have the required functionality so that it can be used, and so that it is not heavyweight and difficult to use from other software. Packages can be created by hand or with tools like catkin_create_pkg.

2.2.3 ROS Stack

A stack is a collection of packages. The goal of collecting these packages is to simplify the process of code sharing. All packages in stack cooperate in such way to provide a certain functionality. Unlike the traditional software libraries, ROS stacks can add functionalities during the execution of the robot's program through topics and services. ROS has numerous stacks. Stacks have versions and sometimes can declare dependencies to other stacks.

Generally, a stack is a directory that among others it has a stack.xml file. Each package in this directory is considered as a part of the stack.

Every stack has a Stack manifest, which is a file with necessary information for the stack and the dependencies to other stacks.

2.2.4 ROS Catkin

Catkin is the ROS build system. It is the set of tools that ROS uses to create executable programs, interfaces, libraries, and scripts that other code can use.

2.2.5 ROS Nodes

ROS's philosophy is to organize the code into small independent pieces of code that run at the same time, in such way to simplify the debugging and to increase the reuse of the code. To achieve this goal, ROS has nodes. A node basically is the process in which computation is performed. In most of the cases, a node serves a specific purpose, for example one controls the navigation of the robot and another processes sensor measurements. Nodes cooperate with one another mostly through topics.

The use of nodes is beneficial to the overall system. As it is widely known, creating many independent code pieces has the advantage that the crashes are isolated. As a result, creating nodes that each one of them acts independently, if an error occurs at one node, it might not cause any damage to the whole system. Also, code complexity is reduced in comparison to monolithic systems.

A running node is uniquely identified to the rest of the system by its name. The type of a node has to be defined. The type of the node simplifies the process of referring to an executable node on the filesystem.

2.2.6 ROS Topics

Topics are named buses over which nodes communicate with each other. Basically, topics are the means to transmit ROS messages between nodes. Each topic has subscriber and publisher nodes. In general, a subscriber node is not aware of which node will consume its messages. The same applies for the publisher node. What happens in particular is that a node that is interested in data subscribes to a topic that provides the data. Respectively nodes that generate data, publish them to the relevant topic. Each topic can have multiple subscribers and publishers as soon as they have different names.

Every topic has a specific type. The type of the topic indicates the message type that the topic can transmit.

2.2.7 ROS Messages

Nodes communication with each other happens by exchanging messages through topics. A message is created by the publisher node and is published to a topic. Another or maybe the same node subscribes to the topic and consumes the message.

Each message has a specific type. The type of a message is defined in a .msg file which contains the description of the type. There are standard message types in ROS but one can create others simply by declaring an .msg file and building the type.

Table 1: Standard ROS message types

Primitive Type	Serialization	C++	Python2	Python3
bool	unsigned 8-bit int	uint8_t	bool	
int8	signed 8-bit int	int8_t	int	
uint8	unsigned 8-bit int	uint8_t	int	
int16	signed 16-bit int	int16_t	int	
uint16	unsigned 16-bit int	uint16_t	int	
int32	signed 32-bit int	int32_t	int	
uint32	unsigned 32-bit int	uint32_t	int	
int64	signed 64-bit int	int64_t	long	int
uint64	unsigned 64-bit int	uint64_t	long	int
float32	32-bit IEEE float	float	float	
float64	64-bit IEEE float	double	float	
string	ascii string	std::string	str	bytes
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time	
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration	

2.2.8 Master Node

As we analyzed previously, ROS software contains many files, nodes and topics. A very important issue that occurs at this point is how all nodes that run at the same time, are exchanging messages and perform computations synchronously. This happens successfully with the use of the ROS master. Before any code can start its execution, the command “roscore” must be used so that the master node will start. Master node has to be running for the entire time that we use ROS.

2.2.9 ROS Service

Services are another means of communication between two nodes. A service is defined by a pair of messages, one for the request and one for the reply. A node provides a service through a string name. A node requires a message and waits for the reply. A service is defined in a srv file.

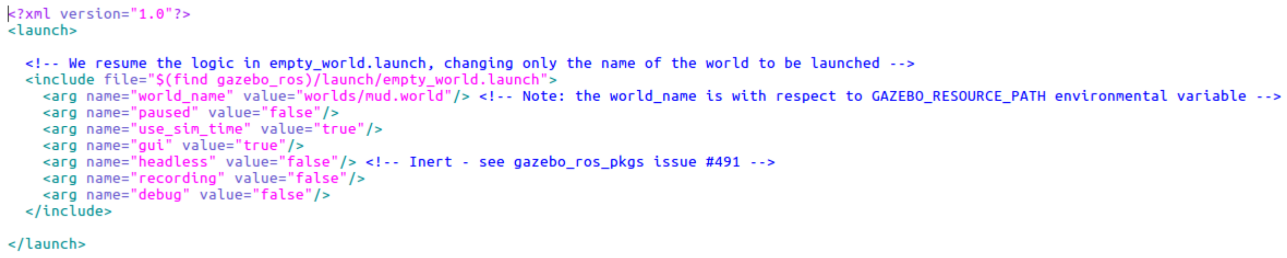
Services also have types. The name of the type is the same as the name of the srv file that defines the service.

2.2.10 Launch files

ROS provides a mechanism to start many nodes all at once, using files called launch files. Launch files are xml files with the .launch extension. Using launch files is necessary in order to have an easy to use ROS set of packages. They also simplify the process of executing the code and run nodes. Launch files have to be associated with a specific package. That is the reason why the launch file should be stored in the package directory. Although, the place of the launch files is not that important because when running roslaunch, the search of the file will extend to all the directory and the subdirectories as well.nThe command to execute a launch file is roslaunch package_name launch_file_name.

2.2.11 World Files

ROS simulators use world files to specify the characteristics of the environment that the turtlebots “live” in and interact with. Basically world files create the physical world in which the turtlebot moves. World files are xml files with the .world extension. These files contain detailed description of all obstacles that the world has. Some of the most important information for the obstacles are the name of the obstacle, its the exact position and its size. In each launch file it is necessary to define a world file because that is the only way to inform the simulator of what to show. In Figure 7 there is an example of such definition.



```
<?xml version="1.0"?>
<launch>

  <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="worlds/mud.world"/> <!-- Note: the world_name is with respect to GAZEBO_RESOURCE_PATH environmental variable -->
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/> <!-- Inert - see gazebo_ros_pkgs issue #491 -->
    <arg name="recording" value="false"/>
    <arg name="debug" value="false"/>
  </include>

</launch>
```

Figure 7 - Launch mud.world file

2.3 Sensors

Unmanned vehicles need sensors to be able to understand their environment and execute specific tasks. For instance, for the robot's navigation, several sensors are necessary, such as a lidar or kinect, maybe combined with a camera. In this thesis we use Kinect sensor for obstacle avoidance. In a real world experiment we would use another sensor for the measurements that we want to keep track of.

2.4 Turtlebot

TurtleBot [6] is a low-cost, personal robot kit with open-source software. TurtleBot is a small robot used by many developers for experiments and new software development. It belongs in the category of unmanned ground vehicles. It consists of a mobile base used for the movement of the robot and many sensors like distance sensors and cameras. Turtlebot does not have a processor. Turtlebot’s operation need an external unit that runs ROS, for example a laptop or Raspberry Pi. The algorithms that control the turtlebot are executed on this unit. Moreover, all necessary sensors needed for a mission have to be connected on the operating unit. A turtlebot is shown in Figure 8.

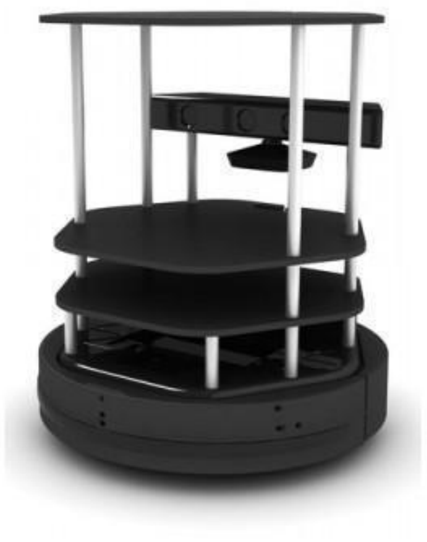


Figure 8 - Turtlebot

2.5 Simulators (Gazebo, rviz)

A well-designed simulator is essential for any robot development. Gazebo helps the developers to easily test their algorithms, design new robot models and new environments to navigate. It also provides convenient programmatic and graphical interfaces. Another important feature that we made use of in this thesis is the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. Gazebo is considered as the leader in robot simulation.

In order to connect the Gazebo simulator with ROS we have to create some ROS packages. This set of packages is called gazebo_ros_pkgs and contains the necessary files to simulate the robot and the environment in which we will test the algorithms. As always the communication between ROS and Gazebo happens through ROS messages, services and dynamic reconfigure. At the installation of ROS, gazebo_ros_pkgs comes with some default files and at most cases the developer does not need to modify at all.

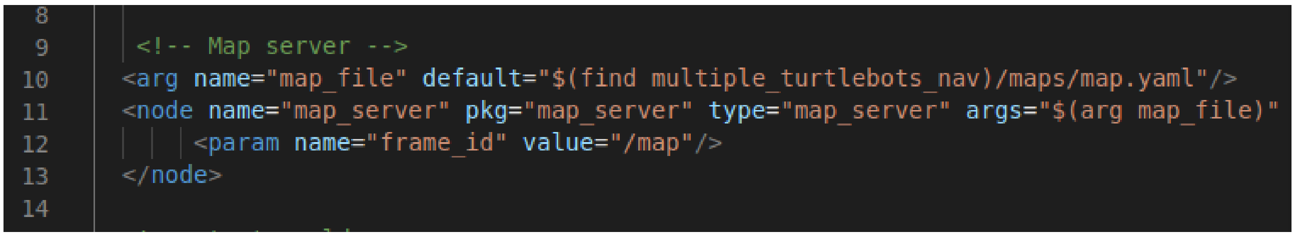
The process to start the Gazebo simulator with the default files is first to run the roscore command to start the Master ROS node and then the command roslaunch turtlebot_gazebo turtlebot_world.launch. The roscore command is not necessary, as each roslaunch command will initialise the Master node if it is not already initialized.

For the causes of this thesis we also used Rviz as a simulator. The main reason that led us to the use of Rviz in this thesis is the need to load a map to the simulator. Gazebo on the other hand, does not give us the opportunity to visualize a map. Also, in this thesis we used laser scan data for most of the development parts andRviz is considered to be an effective free tool for the visualization of these laser scans. Moreover, Rviz has a graphic section that shows the environment and the turtlebots and a section with the topics used for the simulation. That is very useful during the development of the algorithms as we can check these topics.

Running Rviz simulator is simple, just run the command roslaunch rviz rviz.

2.5.1 Map

Rviz gives us the opportunity to visualize a map for the turtlebot to navigate. A Map can be used in ROS simulators by combining two files. First of all we need a .png file that is the image of the map and a .yaml file that describes the map features. To open the map in the simulator the only thing necessary is to provide the map server through the launch file that initializes the simulators. As shown in the next figure (Figure 9), we initialize the map server and provide the yaml file that describes the map.



```
 <!-- Map server -->
<arg name="map_file" default="$(find multiple_turtlebots_nav)/maps/map.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)"
      <param name="frame_id" value="/map"/>
</node>
```

Figure 9- Launch file to open map in Rviz

2.6 Obstacle detection

Obstacle detection is the process of using sensors, data structures, and algorithms to detect objects or terrain types that impede motion [7]. Obstacle detection is generally a crucial issue of any unmanned vehicle that needs to navigate by making decisions on its own. Every unmanned vehicle must have the ability to identify obstacles during its moving it the world. In order to do that it uses many different types of sensors. The most common sensors for obstacle detection are kinect, lidar and cameras. These sensors make it possible for the robot to identify an obstacle and if necessary create clusters of the different types of obstacles.

Obstacle detection is necessary in plenty of robot's functionalities. Mapping, navigation and path planning would have been impossible to be performed without the detection of the obstacles inside the world. After an obstacle has been detected, the robot should move in such way to avoid it.

2.7 Collision Avoidance

Collision Avoidance is the process in which an unmanned vehicle tries not to crash on any obstacles. During a mission the robots travel from one point to another, it has to make sure that the path is clean of obstacles. If an obstacle is detected in this path, the robot has to update and replan the path to its destination. The process of replanning the path is not always easy. It is possible that the robot will not be able to find a path that ends on the desired destination. These are the trickiest cases that the developer of the collision avoidance algorithm has to take into consideration. Collision avoidance is a field of study in robotics that many developers and generally scientists find interesting. The optimization of obstacle avoidance is still an ongoing process. In the last few years, more and more scientists are trying to turn all moving objects into autonomous vehicles. Drones, cars, military devices are some of the basic examples of items that have to become autonomous. Unmanned vehicles of all kind must have well designed algorithms of collision avoidance to be considered safe. That is the reason why in the last years this field of study is attracting the robotics community interest.

3 RELATED WORK

3.1 UxV Path Planning

When dealing with unmanned vehicles, a very important issue is the path planning. Many years now, scientists have developed algorithms for path planning in static and dynamic environments. Real-time path planning is a necessary layer for the movement of the unmanned vehicle in an unknown environment. Firstly, the robot has to detect potential obstacles and avoid them. A proposed approach [8] is use the Voronoi diagram-based algorithm. This algorithm separates the world into cells and creates a grid. In this grid each cell contains an empty area or an obstacle. The Voronoi algorithm has been used in other research works [9], for the initial path planning of unmanned vehicles. It has been improved by adding utility function for each suggested path so that the unmanned vehicles create an optimal path eventually.

Another suggested algorithm is the D* [10] algorithm, which is an algorithm for path planning depending on sensors and a map with full, little of none knowledge of the environment. Like the A* algorithm [11], this algorithm constructs a directive graph with cost functions for every path. The important part of this D* algorithm, and what differentiates it from the A*, is that during the motion of the unmanned vehicle, the graph is constantly updated. As a result, the vehicle will come up with an optima path.

3.2 Maze challenge[12] – Touching the wall

Lately the robotics community has inserted a new challenge, the maze solving challenge. In this challenge a robot is placed in an unmapped maze and it tries to find a way out of it. Many algorithms have been developed to solve this maze challenge. The most popular ones are the random mouse and the wall follower [12]. The first algorithm can be applied in every maze and does not have specific requirements. The second one can be applied on mazes that are ‘simply connected’ or ‘perfect’, meaning that they do not contain loops.

In the random mouse algorithm, the unmanned vehicle picks a path randomly every time and it follows it until it reaches a dead-end. In this case, it picks a path again and follows it. It repeats this process until it finally gets out of the maze. This method uses no intelligence and the robot is acting like a mouse, hence the name. This algorithm will sourly find the exit of the maze eventually but the time that it will need cannot be calculated or even estimated.

The wall follow algorithm is considered the best algorithm so far for solving the maze challenge. This algorithm guarantees a solution to the problem, if the maze is ‘perfect’. In this algorithm the unmanned vehicle walks in the maze while ‘touching’ the wall with one hand. It is also called ‘left-hand’ or ‘right-hand’ algorithm depending on the hand that touches the wall. The advantage of this algorithm is that if an exit does not exist, then the robot will return at its initial position and it will have traversed every corridor in the maze.

3.3 Search of unknown environment

The research of searching for specific points of interest using unmanned vehicles started many years before. In 1940 Koopman [13] focused his work on exploring an environment using sensors to find an object that was placed in a random position, using random search patterns. In latter research works [14], the studies are focused in the optimization of the searching process and developing algorithm that use more intelligence and decision making. Finally, in a research for the mathematical modeling of the searching problem [15], the searching process is considered as a probabilistic phenomenon. The objects that the unmanned vehicles try to find, are placed in the environment according a known spatial likelihood. With this theory in mind, each position of the environment becomes a searching cell and each searching path is a sequence of multiple searching cells. In order to process these searching structures, advanced computers are necessary.

During the last years many researchers have focused their work on algorithms for cooperative unmanned vehicles that try to explore the environment or navigate in it while sensing measurements. There are many other different approaches for the use of collaboration on unmanned vehicles. In the research mentioned before, the effectiveness of the unmanned vehicle collaboration is tested. The collaboration of the unmanned vehicles offers a probabilistic estimation of the gain that an unmanned vehicle will have by searching in a previously visited cell. In this way, the decision whether the unmanned vehicle should visit or not the specific cell can be made before the vehicle actually visits the cell. Another interesting approach [16] for the collaborative unmanned vehicles focuses on the decentralization of the control in order to split the risk of failure. This research was developed for UAVs. In this case the control of the collaborative team of vehicles is based on the real-time creation of tasks from the user. The unmanned vehicles allocate tasks using onboard computation and UAV to UAV communication. Each UAV allocates a task only if it can complete its mission with lower cost than all the other UAVs. The aim of this research is to minimize the time until the last vehicle has completed its tasks. The allocation of the tasks has two steps. Firstly, the UAV finds the possible tasks depending on its local knowledge. Secondly, it checks the time that other UAVs have stated that they need to complete each task, and if it can accomplish less time, it allocates some extra tasks as well.

In many researches the collaboration of unmanned vehicles is based on Particle swarm optimization method (PSO). PSO can be used for a team of vehicles [17] [18] or for all available vehicles. This method is useful in cases when a team of vehicles is trying to explore an unknown environment or search for specific objects. At first all vehicles are moving in the world according to some limitations regarding their position. When a vehicle indicates that its position is better than the others, then all other vehicles have to move towards it. In cases of world exploration, each vehicle takes into consideration the knowledge of all the vehicles and not only the local knowledge. As a result, the unmanned vehicles complete their missions faster and more accurately.

4 COLLABORATIVE SEARCH PLANNING FOR MULTIPLE VEHICLES IN IOT ENVIRONMENT

4.1 Problem Definition

In this Thesis, we focused our research on developing effective algorithms for scanning a previously unknown world and at the same time, try to detect sources of measurements (context awareness) using autonomous ground vehicles that collaborate with each other. For our experiments we used two turtlebots although the algorithms developed can be used in experiments with more robots of any kind.

4.2 Challenges

The problem stated previously can be splitted in different smaller parts. Firstly, the unmanned vehicles have to be able to navigate in a previously unknown environment without crashing on each other or on other obstacles. Secondly, it tries to optimize the the context aware process, i.e. trying to find the best measurement of a sensor in the minimum time.

4.2.1 Context exploration

In this Thesis the turtlebots used for the experiments do not have any information about the environment in which they operate. The exploration of the environment can be separated into two tasks. Firstly, the task of finding which positions of the world are available and which are not, meaning that they have obstacles. Secondly, for the purposes of this Thesis, in the tasks of context exploration, we develop path planning algorithms in order to find the possible sensor sources in the world efficiently. The two turtlebots have to move in the world and trace the sensor sources and as a result at the end of the experiments they will have collected information about the level of measurements in each position.

4.2.2 Obstacle avoidance

In our experiments, given the fact that the turtlebots do not have a map of the world, we have to manage the cases in which the turtlebot’s path is blocked by an obstacle. Another important issue is that our world is surrounded by walls, and the turtlebots do not know where these walls are located. Lastly, the algorithms are developed for more than one turtlebot that operate as a team. As a result, we have to take into consideration the possibility of one turtlebot’s path crosses the others. For all these purposes, we used the laser scan data from a ROS topic and we created custom topics that will be analyzed in later chapters.

4.2.3 Efficiency and complexity

In the problem stated, on the one hand the turtlebots in order to find the best sensor’s measurement have to explore all the environment and at the end decide in which position they found the optimal solution. On the other hand this algorithm will consume much time but will surely find the optimal solution. The main question that we will try to answer in this Thesis is whether there is a way to ensure a good enough solution in a more acceptable time. In our first algorithm we will try to explore the most possible positions to find the best measurement and in our second algorithm we will program the turtlebots to follow the measurements collaboratively. Our experiments test these two scenarios.

4.3 Proposed Solution

4.3.1 Simulation setup

For the purposes of this Thesis, we created algorithms that run on simulators. For the experiments that test our research we created a custom map of measurements, in order to simulate the measurements that the robot could have been receiving from sensors. The basic idea is that in any position of the world corresponds a number that indicates the measurement of the sensor. Hence the map of measurements has the same structure as the map of the world. In Figure 10 there is the map of the world and example of the maps of measurements that we build depending on the map of the turtlebot's world is shown in the Figures 11 and 12.

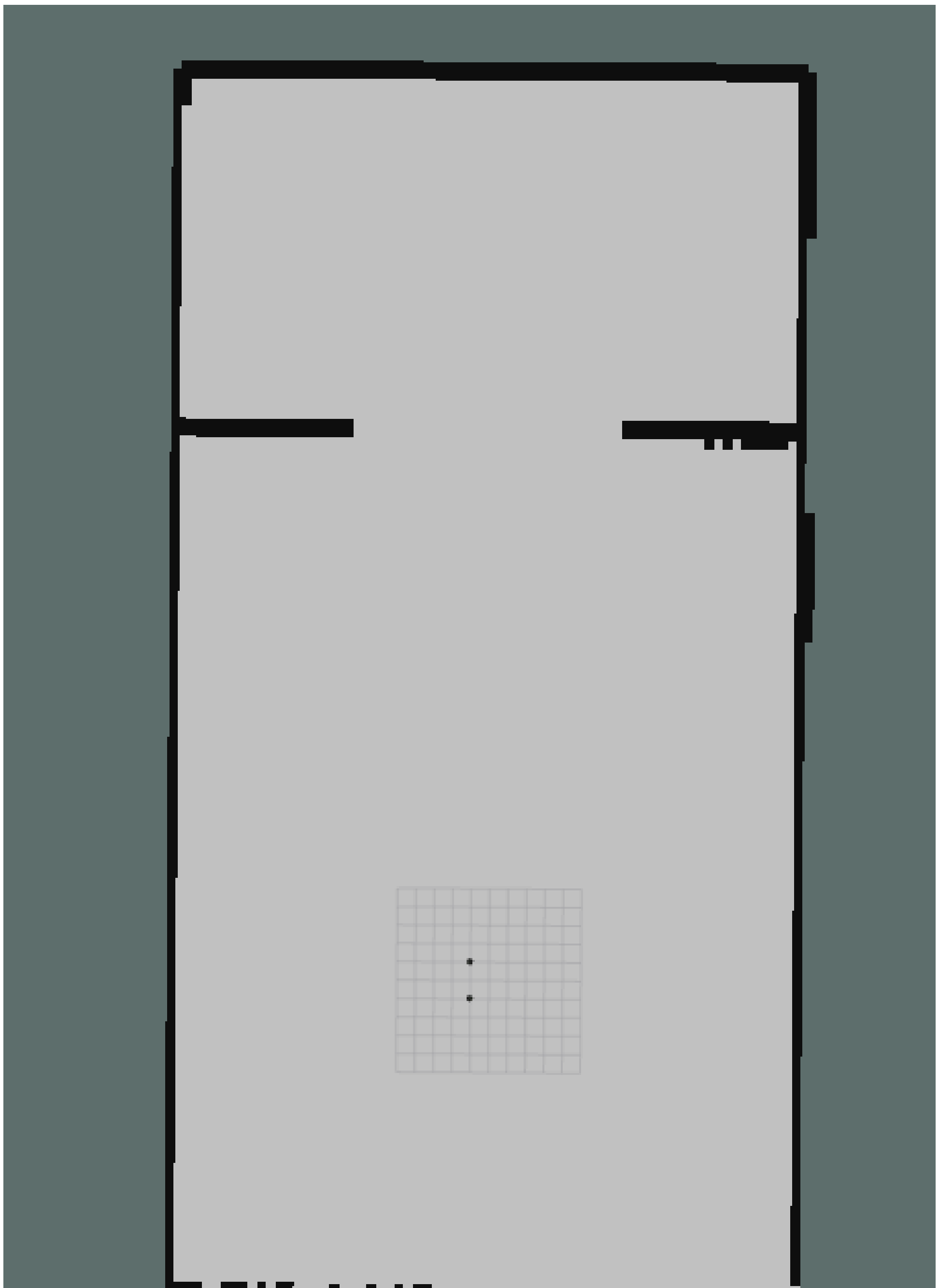


Figure 10 - Map of the world

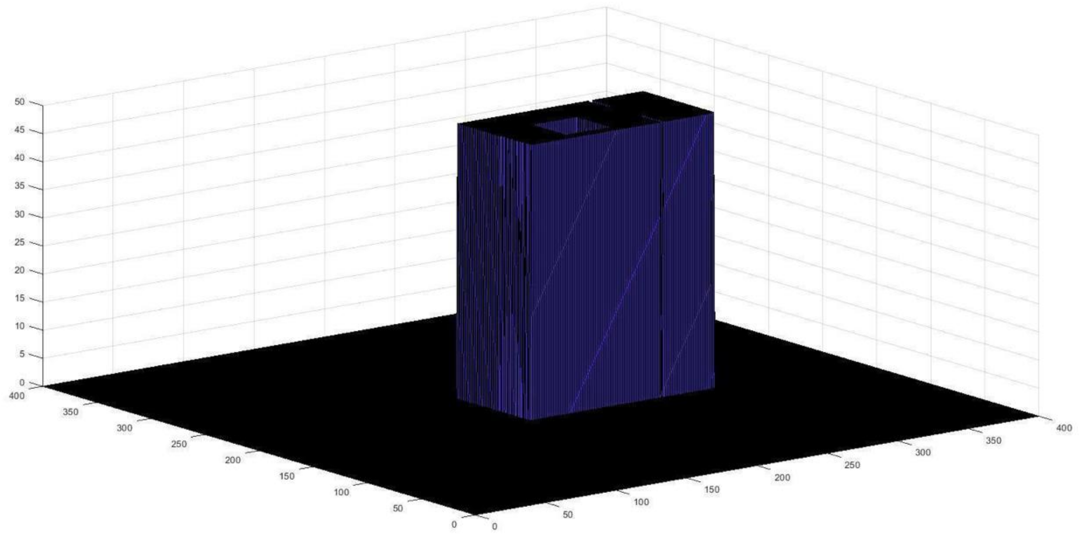


Figure 11 - Map of measurements with one source

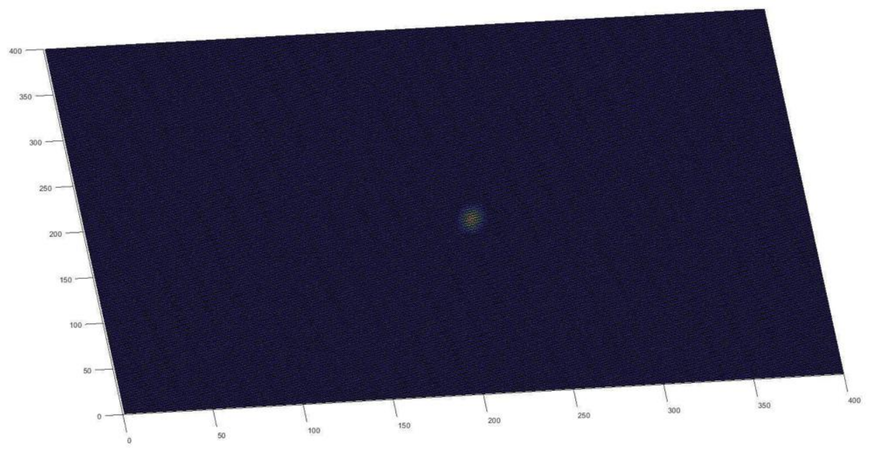
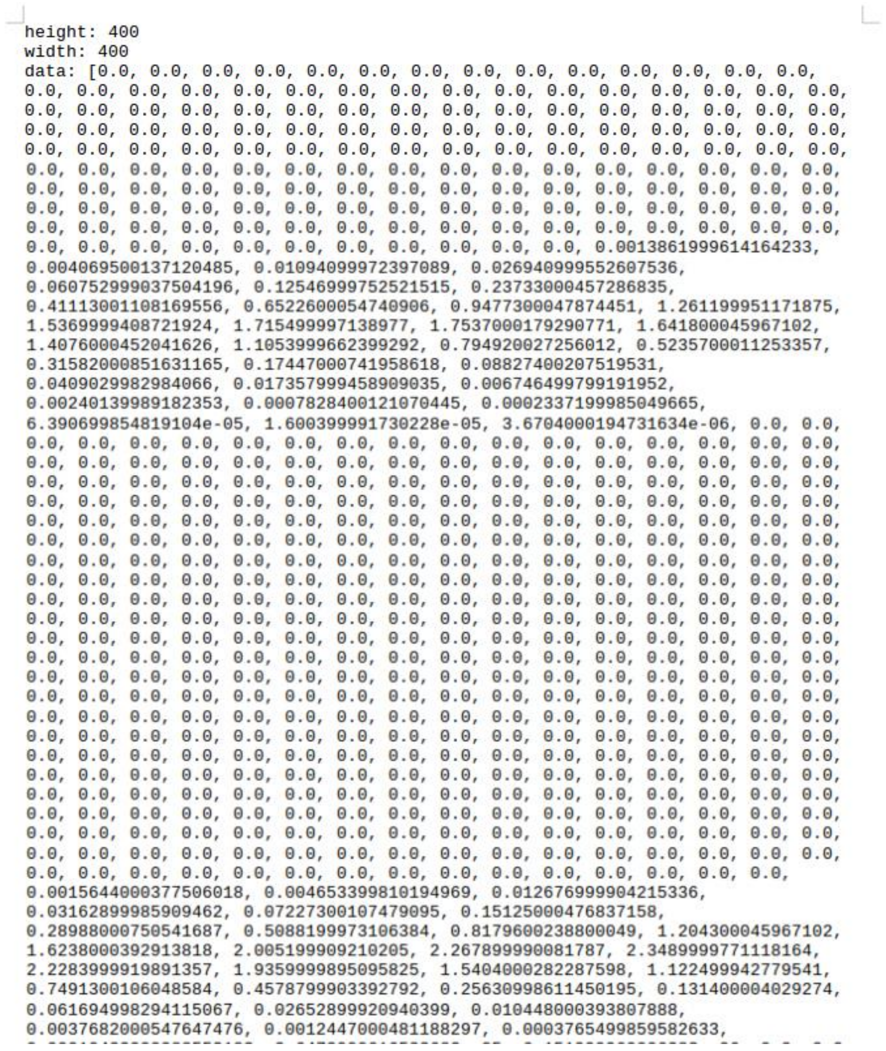


Figure 12 - Example of created map of measurements with one source

In the diagrams above, one source of measurements is depicted. The measurements follow the normal distribution and that is why in the second diagram we see the source as a circle.

We developed a matlab function that creates gaussian sensor sources that follow the normal distribution. These sources are randomly distributed in the map of the turtlebot's world. The input of our function is a generic map of the turtlebot and the output is a csv containing the sensor sources in space.

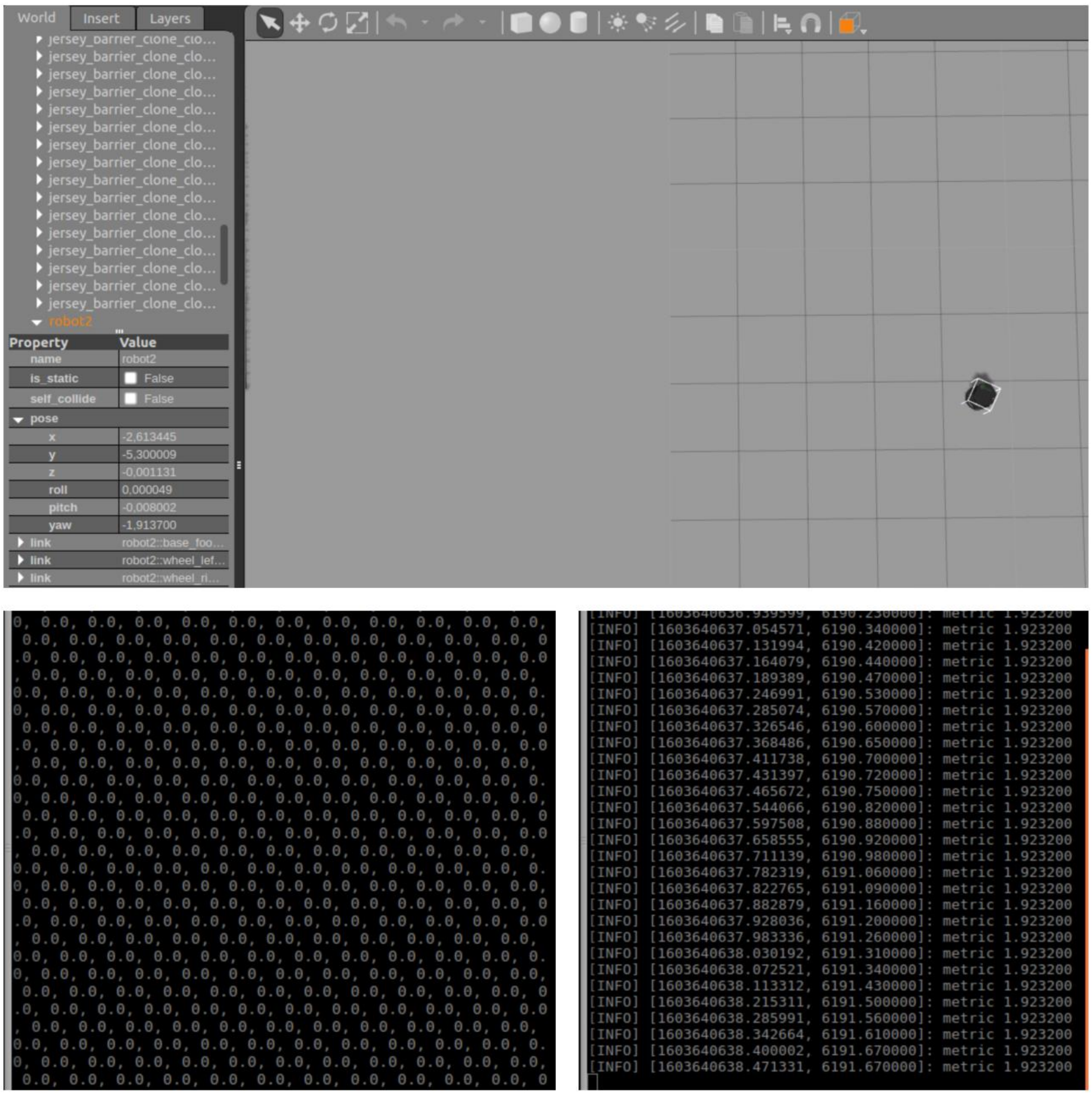
These measurements have to be published in a topic so that the turtlebots can have access to. This is the main purpose that the node " Publish Metrics" serve. We created a topic called "/metrics" of type Num. The type of this topic was inspired from the '/map' topic of ROS. The node opens the csv file that was created from the matlab program and is constantly publishing the measurements to the '/metrics' topic. The turtlebots are able to subscribe to this topic and read the measurements. In a real-life experiment the turtlebots will also read the sensor's measurements from a topic, that means that the algorithms developed for this research could be used in real life with the minimum changes. In the following figure (Figure 13) a part of the topic's data is shown.



```
height: 400
width: 400
data: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0013861999614164233,
0.004069500137120485, 0.01094099972397089, 0.026940999552607536,
0.060752999037504196, 0.12546999752521515, 0.23733000457286835,
0.41113001108169556, 0.6522600054740906, 0.9477300047874451, 1.261199951171875,
1.5369999408721924, 1.715499997138977, 1.7537000179290771, 1.641800045967102,
1.4076000452041626, 1.1053999662399292, 0.794920027256012, 0.5235700011253357,
0.31582000851631165, 0.17447000741958618, 0.08827400207519531,
0.0409029982984066, 0.017357999458909035, 0.006746499799191952,
0.00240139989182353, 0.0007828400121070445, 0.0002337199985049665,
6.390699854819104e-05, 1.600399991730228e-05, 3.6704000194731634e-06, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0015644000377506018, 0.004653399810194969, 0.012676999904215336,
0.03162899985909462, 0.07227300107479095, 0.15125000476837158,
0.28988000750541687, 0.5088199973106384, 0.8179600238800049, 1.204300045967102,
1.6238000392913818, 2.005199909210205, 2.267899990081787, 2.3489999771118164,
2.2283999919891357, 1.9359999895095825, 1.5404000282287598, 1.122499942779541,
0.7491300106048584, 0.4578799903392792, 0.25630998611450195, 0.131400004029274,
0.061694998294115067, 0.02652899920940399, 0.010448000393807888,
0.0037682000547647476, 0.0012447000481188297, 0.0003765499859582633,
```

Figure 13 - metrics_topic sample data

In Figure 14 the measurement read from the turtlebot in a certain position is shown.



```
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
jersey_barrier_clone_clo...
robot2
```

Property	Value
name	robot2
is_static	☐ False
self_collide	☐ False
pose	
x	-2,613445
y	-5,300009
z	-0,001131
roll	0,000049
pitch	-0,008002
yaw	-1,913700
link	robot2::base_foo...
link	robot2::wheel_lef...
link	robot2::wheel_ri...

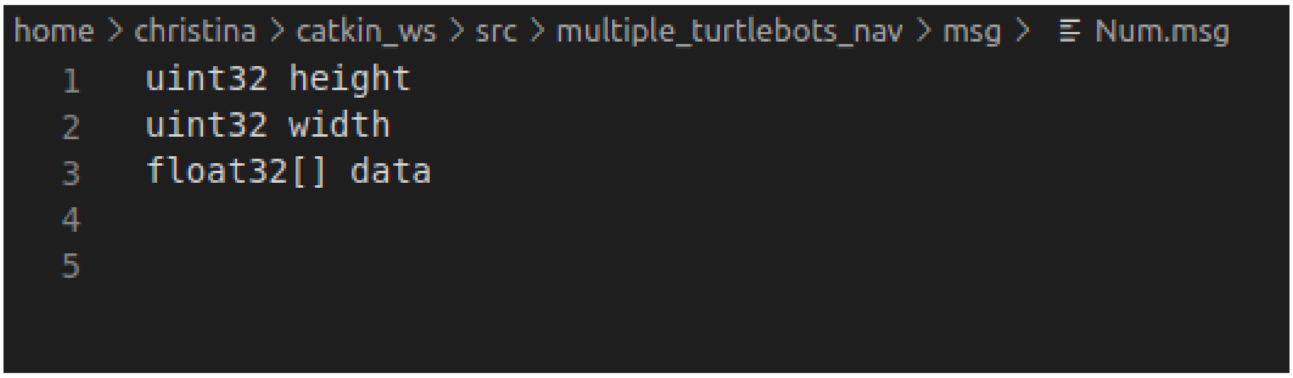
```
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
```

```
[INFO] [1603640636.939599, 6190.230000]: metric 1.923200
[INFO] [1603640637.054571, 6190.340000]: metric 1.923200
[INFO] [1603640637.131994, 6190.420000]: metric 1.923200
[INFO] [1603640637.164079, 6190.440000]: metric 1.923200
[INFO] [1603640637.189389, 6190.470000]: metric 1.923200
[INFO] [1603640637.246991, 6190.530000]: metric 1.923200
[INFO] [1603640637.285074, 6190.570000]: metric 1.923200
[INFO] [1603640637.326546, 6190.600000]: metric 1.923200
[INFO] [1603640637.368486, 6190.650000]: metric 1.923200
[INFO] [1603640637.411738, 6190.700000]: metric 1.923200
[INFO] [1603640637.431397, 6190.720000]: metric 1.923200
[INFO] [1603640637.465672, 6190.750000]: metric 1.923200
[INFO] [1603640637.544066, 6190.820000]: metric 1.923200
[INFO] [1603640637.597508, 6190.880000]: metric 1.923200
[INFO] [1603640637.658555, 6190.920000]: metric 1.923200
[INFO] [1603640637.711139, 6190.980000]: metric 1.923200
[INFO] [1603640637.782319, 6191.060000]: metric 1.923200
[INFO] [1603640637.822765, 6191.090000]: metric 1.923200
[INFO] [1603640637.882879, 6191.160000]: metric 1.923200
[INFO] [1603640637.928036, 6191.200000]: metric 1.923200
[INFO] [1603640637.983336, 6191.260000]: metric 1.923200
[INFO] [1603640638.030192, 6191.310000]: metric 1.923200
[INFO] [1603640638.072521, 6191.340000]: metric 1.923200
[INFO] [1603640638.113312, 6191.430000]: metric 1.923200
[INFO] [1603640638.215311, 6191.500000]: metric 1.923200
[INFO] [1603640638.285991, 6191.560000]: metric 1.923200
[INFO] [1603640638.342664, 6191.610000]: metric 1.923200
[INFO] [1603640638.400002, 6191.670000]: metric 1.923200
[INFO] [1603640638.471331, 6191.670000]: metric 1.923200
```

Figure 14 - Turtlebot's measurements in specific position

4.3.2 Create custom message types

Most of the software developers have faced the need to build custom message types. In this thesis we created two message types as well. In the following figures (Figure 15 and Figure 16) these messages are shown.

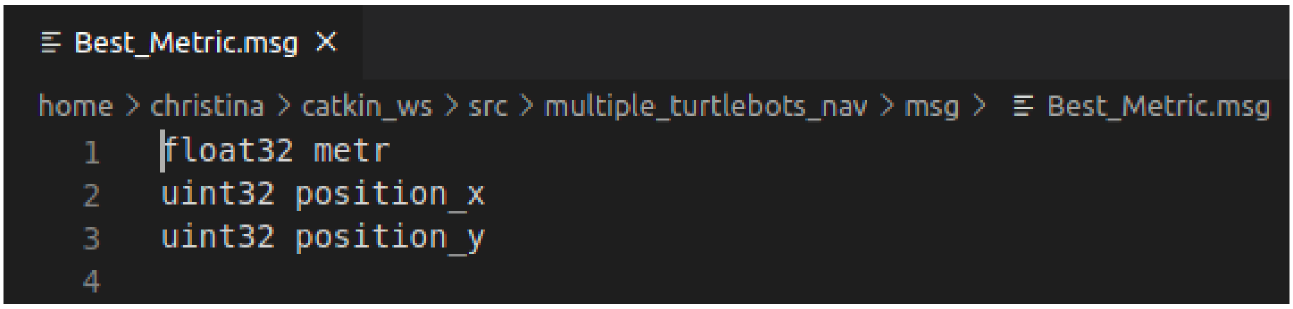


```
home > christina > catkin_ws > src > multiple_turtlebots_nav > msg > Num.msg
uint32 height
uint32 width
float32[] data
```

Figure 15 - Custom message type. The message type name is Num and it contains an array of integers and two integers, one for each dimension of the array

The 'Num' message type is used for publishing the measurements on the '/metrics' topic. The first two integers, the height and the width shows the dimensions of the world. In our experiments the world has 400 points height and 400 points width. These two integers have to be equal to the height and width of the map of the world otherwise the turtlebot might go to a point where no measurement is defined.

The array of floats stands for the actual measurements that the world has at each position. The array has only one dimension and it has size equal to height * width.

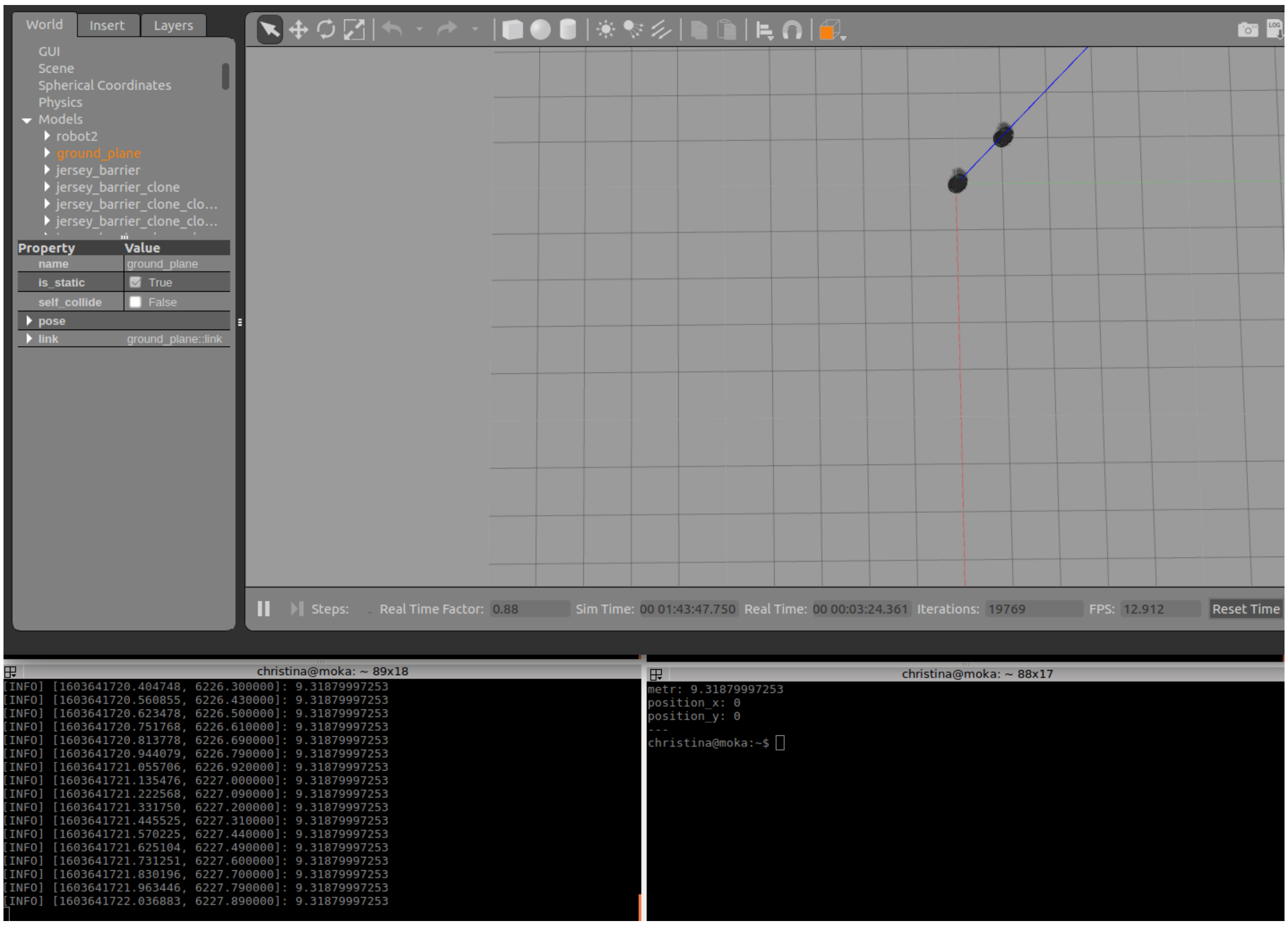


```
Best_Metric.msg
home > christina > catkin_ws > src > multiple_turtlebots_nav > msg > Best_Metric.msg
float32 metr
uint32 position_x
uint32 position_y
```

Figure 16 - Custom message type.

The message type name is Best_Metric and it contains two integers, position_x and position_y and one float number named metr

'Best_metric' message type was built to store the best measurements noted in a certain position. The integers position_x and position_y are the coordinates of this position and the float metr stands for the measurement in this position. These messages are published in a custom topic, called robotX/local_best. The first turtlebot can store its local best value in the topic robot1/local_best and read the other turtlebot's local best value from the topic robot2/local_best. In the following figure (Figure 17) the measurements published in these topics are shown. At the bottom left area of the figure is the best measurement that the turtlebot has found so far and at the bottom right area we see an instance of the robot1/local_best topic.



```
[INFO] [1603641720.404748, 6226.300000]: 9.31879997253
[INFO] [1603641720.560855, 6226.430000]: 9.31879997253
[INFO] [1603641720.623478, 6226.500000]: 9.31879997253
[INFO] [1603641720.751768, 6226.610000]: 9.31879997253
[INFO] [1603641720.813778, 6226.690000]: 9.31879997253
[INFO] [1603641720.944079, 6226.790000]: 9.31879997253
[INFO] [1603641721.055706, 6226.920000]: 9.31879997253
[INFO] [1603641721.135476, 6227.000000]: 9.31879997253
[INFO] [1603641721.222568, 6227.090000]: 9.31879997253
[INFO] [1603641721.331750, 6227.200000]: 9.31879997253
[INFO] [1603641721.445525, 6227.310000]: 9.31879997253
[INFO] [1603641721.570225, 6227.440000]: 9.31879997253
[INFO] [1603641721.625104, 6227.490000]: 9.31879997253
[INFO] [1603641721.731251, 6227.600000]: 9.31879997253
[INFO] [1603641721.830196, 6227.700000]: 9.31879997253
[INFO] [1603641721.963446, 6227.790000]: 9.31879997253
[INFO] [1603641722.036883, 6227.890000]: 9.31879997253
```

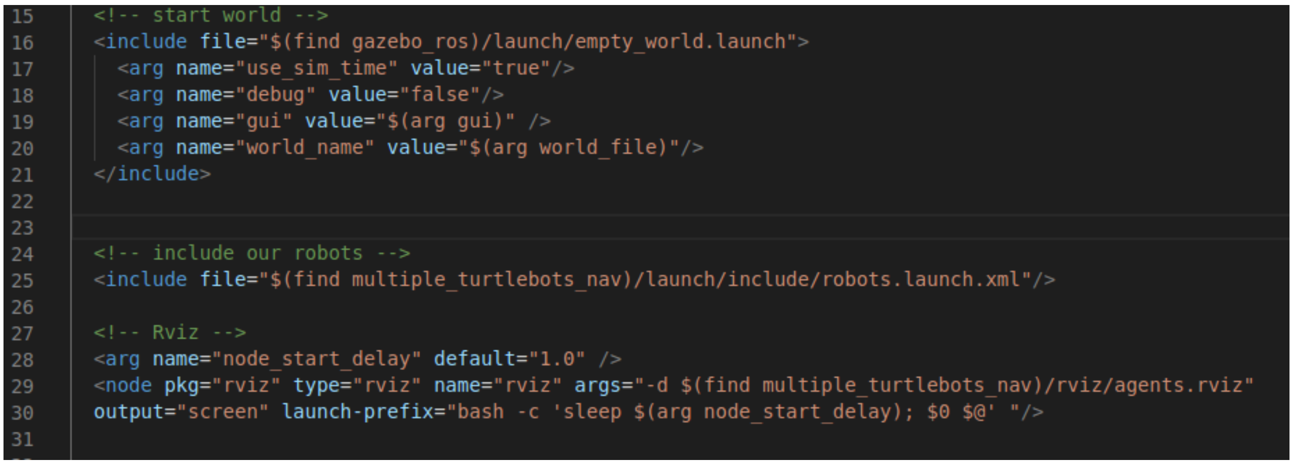
```
metr: 9.31879997253
position_x: 0
position_y: 0
---
christina@moka:~$ 
```

Figure 17 - best_metrics data

4.3.3 Create launch files

For the causes of this thesis we implemented many launch files. The main files created are the launch file used for the navigation of the turtlebots and the execution of the simulators.

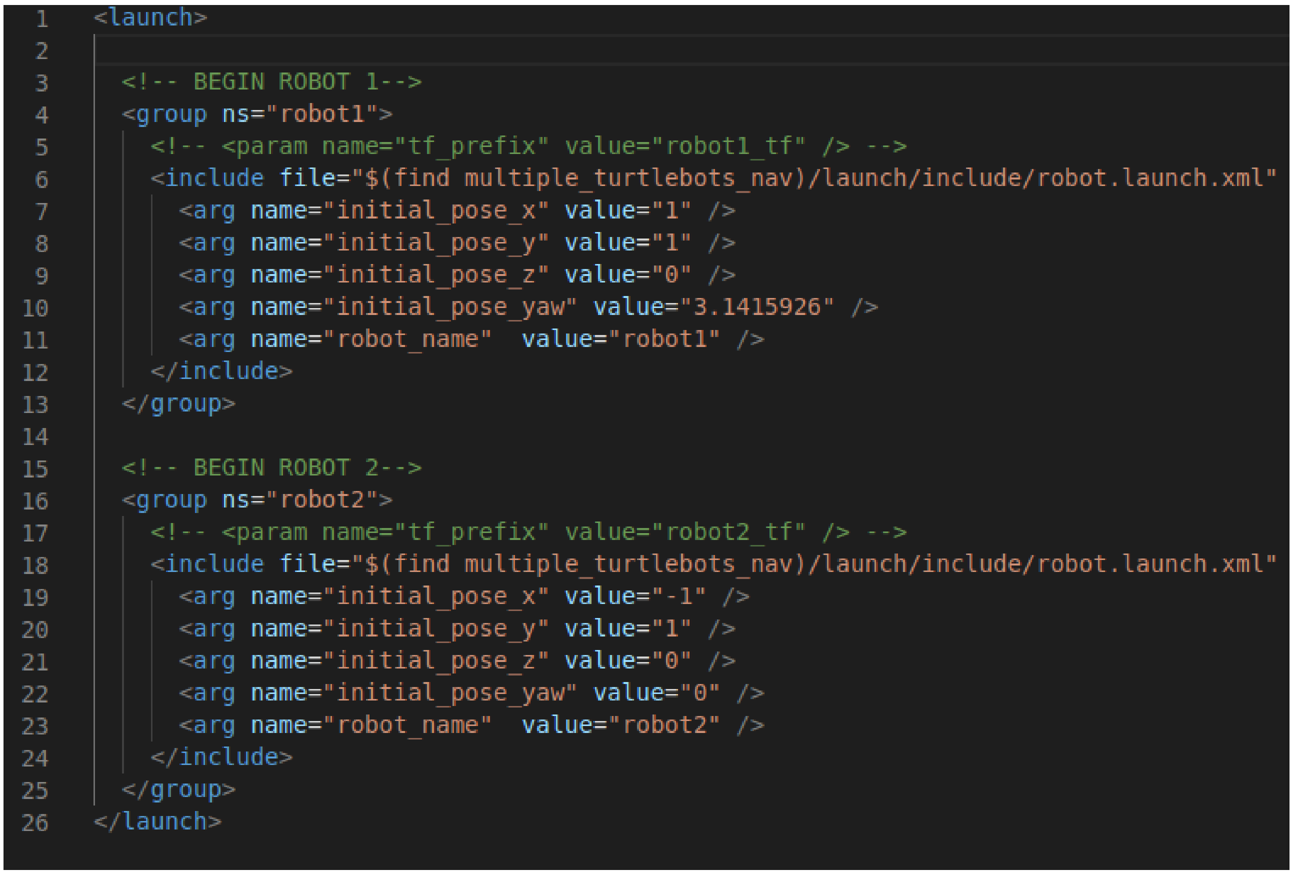
In this Thesis we need to start both simulations at the same time. Also, we have to add two turtlebots in the simulators. For this cause we have implemented a launch file. The launch file used in this thesis is shown in Figure 18.



```
<!-- start world -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="use_sim_time" value="true"/>
  <arg name="debug" value="false"/>
  <arg name="gui" value="$(arg gui)" />
  <arg name="world_name" value="$(arg world_file)"/>
</include>


<!-- include our robots -->
<include file="$(find multiple_turtlebots_nav)/launch/include/robots.launch.xml"/>

<!-- Rviz -->
<arg name="node_start_delay" default="1.0" />
<node pkg="rviz" type="rviz" name="rviz" args="-d $(find multiple_turtlebots_nav)/rviz/agents.rviz"
output="screen" launch-prefix="bash -c 'sleep $(arg node_start_delay); $0 $@' "/>

```

Figure 18 - launch file to start simulators

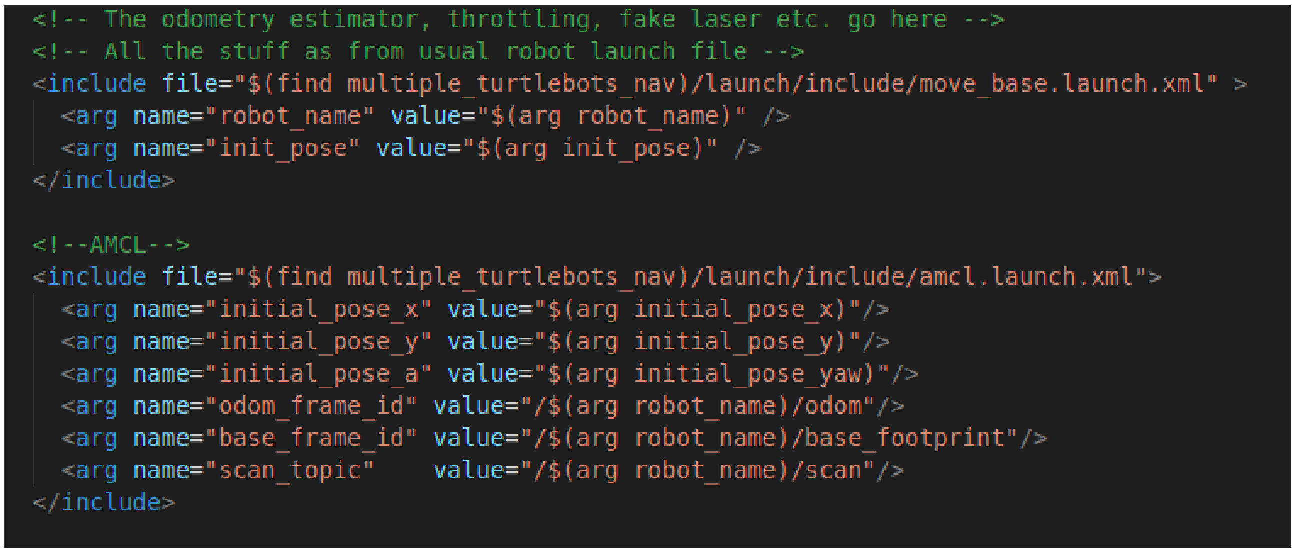
In the figure above, the lines 24-25, are responsible to mount two turtlebots in the simulators. The file robots.launch.xml defines how many and which turtlebot specifically will be shown in the simulators. In order to begin more turtlebots in the simulators we have to add code lines in the <group> tag at the end of this file and change the initial position of the third turtlebot, by changing the values in the <arg> tags . In Figure 19 a screenshot of this file is shown.



```
<launch>

  <!-- BEGIN ROBOT 1-->
  <group ns="robot1">
    <!-- <param name="tf_prefix" value="robot1_tf" /> -->
    <include file="$(find multiple_turtlebots_nav)/launch/include/robot.launch.xml"
      <arg name="initial_pose_x" value="1" />
      <arg name="initial_pose_y" value="1" />
      <arg name="initial_pose_z" value="0" />
      <arg name="initial_pose_yaw" value="3.1415926" />
      <arg name="robot_name"  value="robot1" />
    </include>
  </group>

  <!-- BEGIN ROBOT 2-->
  <group ns="robot2">
    <!-- <param name="tf_prefix" value="robot2_tf" /> -->
    <include file="$(find multiple_turtlebots_nav)/launch/include/robot.launch.xml"
      <arg name="initial_pose_x" value="-1" />
      <arg name="initial_pose_y" value="1" />
      <arg name="initial_pose_z" value="0" />
      <arg name="initial_pose_yaw" value="0" />
      <arg name="robot_name"  value="robot2" />
    </include>
  </group>
</launch>
```

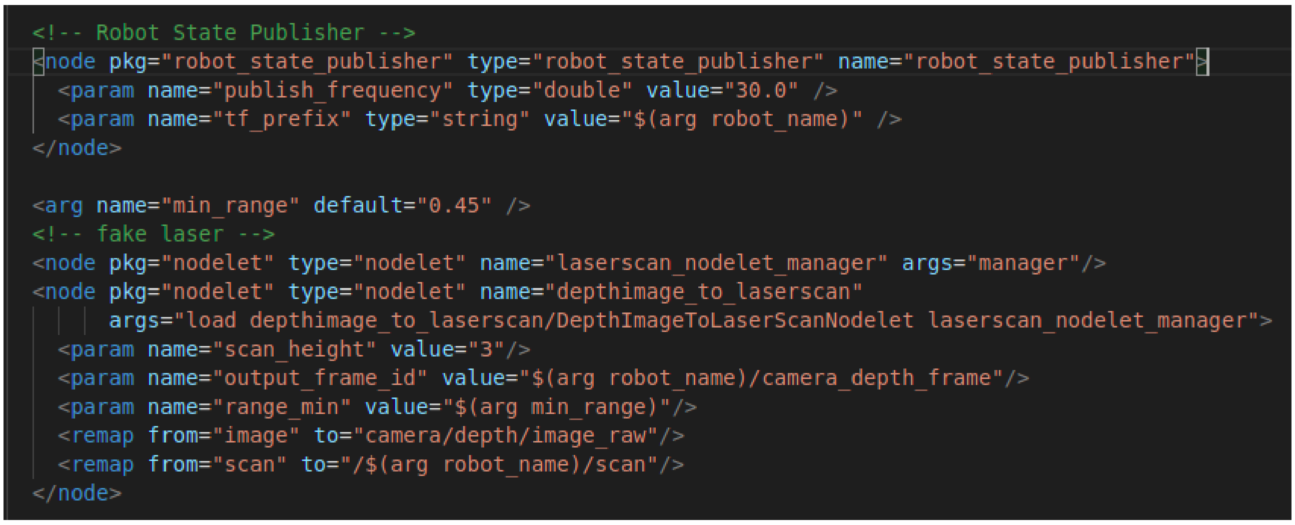
Figure 19 - Launch file to begin turtlebots

Finally, the xml file robot.launch.xml has the detailed characteristics of the turtlebot that will be shown at the simulator. In this file we firstly indicate which urdf file will be used for the turtlebot description. The turtlebot description is basically the external characteristics that the turtlebot has. In the installation of the gazebo simulation, a default urdf file is being downloaded, so we use that one for our simulations. There is no need to change the turtlebot description because it only affects the image of the robot. Some additional important information that we define in the robot xml file are the 3D sensor that the robot will use for its navigation, the name of the topics used for the navigation, the frequency that the robot will publish its position at the right topics, and the name of the laser scan topic. Some parts of this file is shown below (Figure 20 and 21).



```
<!-- The odometry estimator, throttling, fake laser etc. go here -->
<!-- All the stuff as from usual robot launch file -->
<include file="$(find multiple_turtlebots_nav)/launch/include/move_base.launch.xml" >
  <arg name="robot_name" value="$(arg robot_name)" />
  <arg name="init_pose" value="$(arg init_pose)" />
</include>

<!--AMCL-->
<include file="$(find multiple_turtlebots_nav)/launch/include/amcl.launch.xml">
  <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
  <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
  <arg name="initial_pose_a" value="$(arg initial_pose_yaw)"/>
  <arg name="odom_frame_id" value="/$(arg robot_name)/odom"/>
  <arg name="base_frame_id" value="/$(arg robot_name)/base_footprint"/>
  <arg name="scan_topic"    value="/$(arg robot_name)/scan"/>
</include>
```

Figure 20 - Robot.launch.xml file: Odometry estimator, name of topics for robot’s position



```
<!-- Robot State Publisher -->
<node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">
  <param name="publish_frequency" type="double" value="30.0" />
  <param name="tf_prefix" type="string" value="$(arg robot_name)" />
</node>

<arg name="min_range" default="0.45" />
<!-- fake laser -->
<node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager" args="manager"/>
<node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
      args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet laserscan_nodelet_manager">
  <param name="scan_height" value="3"/>
  <param name="output_frame_id" value="$(arg robot_name)/camera_depth_frame"/>
  <param name="range_min" value="$(arg min_range)"/>
  <remap from="image" to="camera/depth/image_raw"/>
  <remap from="scan" to="/$(arg robot_name)/scan"/>
</node>
```

Figure 21- Robot.launch.xml file: Initial position, sensor definition and urdf file import

4.3.4 Single Vehicle Path Planning

The first algorithm that we developed for this Thesis is an heuristic path planning algorithm based on meanders. It basically is a simple world scan algorithm. Turtlebot is not aware of the world around. Therefore, it is needed to scan the space and define the border limits of the world. The turtlebot at first tries to find a random wall. It starts moving forward until it reaches an obstacle. The detection of an obstacle is based on the laserscan data from the topic ‘robotX/scans’, where X stands for 1 or 2. In the next step of the algorithm the turtlebot turns 90 degrees left, so that it has the wall at it’s right hand. Based on the “Wall follow” algorithm, the turtlebot follows the wall while “touching it” with its right hand, until it reaches its initial position. In Figure 22 is the path followed by the turtlebot.

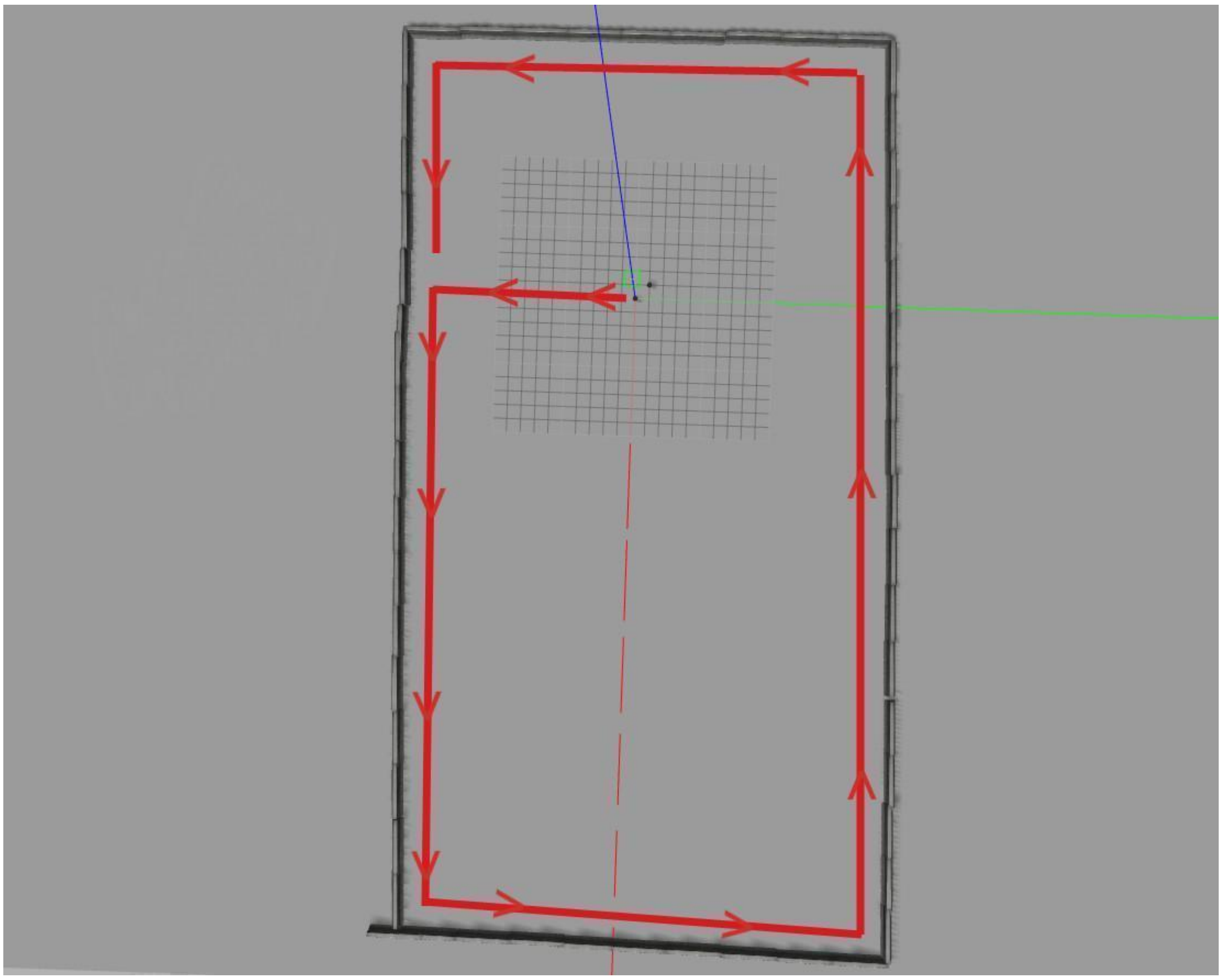


Figure 22 - Turtlebot follows wall with right hand

During the process described above, the node ‘Publish Metrics’ is also running. The turtlebot is subscribing to this topic and reads the measurement that its current position contains.

After the turtlebot scans the surrounding world, it starts scanning the inner world. The path that the turtlebot follows creates a meander, hence the name of the algorithm. As shown in the following figure (Figure 23), starting from the position A, it moves to the position B. When the turtlebot reaches the wall in position B, it turns 90 degrees left, then moves 5 meters ahead, reaching position C. Then it turns 90 degrees left again and goes forward until it reaches a wall once again (position D). Following the same logic, it turns right 90 degrees, goes 5 meters ahead and turns 90 degrees right again. The turtlebot repeats this process until it reaches the top or bottom wall of the world. At the inner scan part of the algorithm, we chose not to scan every possible position in the world. Specifically when the turtlebot reaches one of the side walls, it turns and moves ahead 5 meters and then turns again and moves towards the other wall. In this way, many of the inner points of the world are not examined regarding the measurements that they contain. We made this agreement based on two reasons. Firstly, the number of meters that the turtlebot will move have an important impact on the time consuming for the algorithm execution. Increasing the meters of the movement of the turtlebot reduces the times that the turtlebot will have to go from one wall to the other. On the other hand if we increase the number of meters beyond an ideal limit will reduce the possibility to find an acceptable source of measurements. Also, if we consider the fact that the measurements follow the Normal Distribution, it is understandable that measurements cannot be separated in the world of the turtlebot. This allows us to skip some points of the world, without risking to miss a source of measurements. Although we may not find the best measurement of the source, but for the purposes of this Thesis, this is considered acceptable. We decided that moving 5 meters gives us acceptable results and the algorithm is effective considering the time consumed.

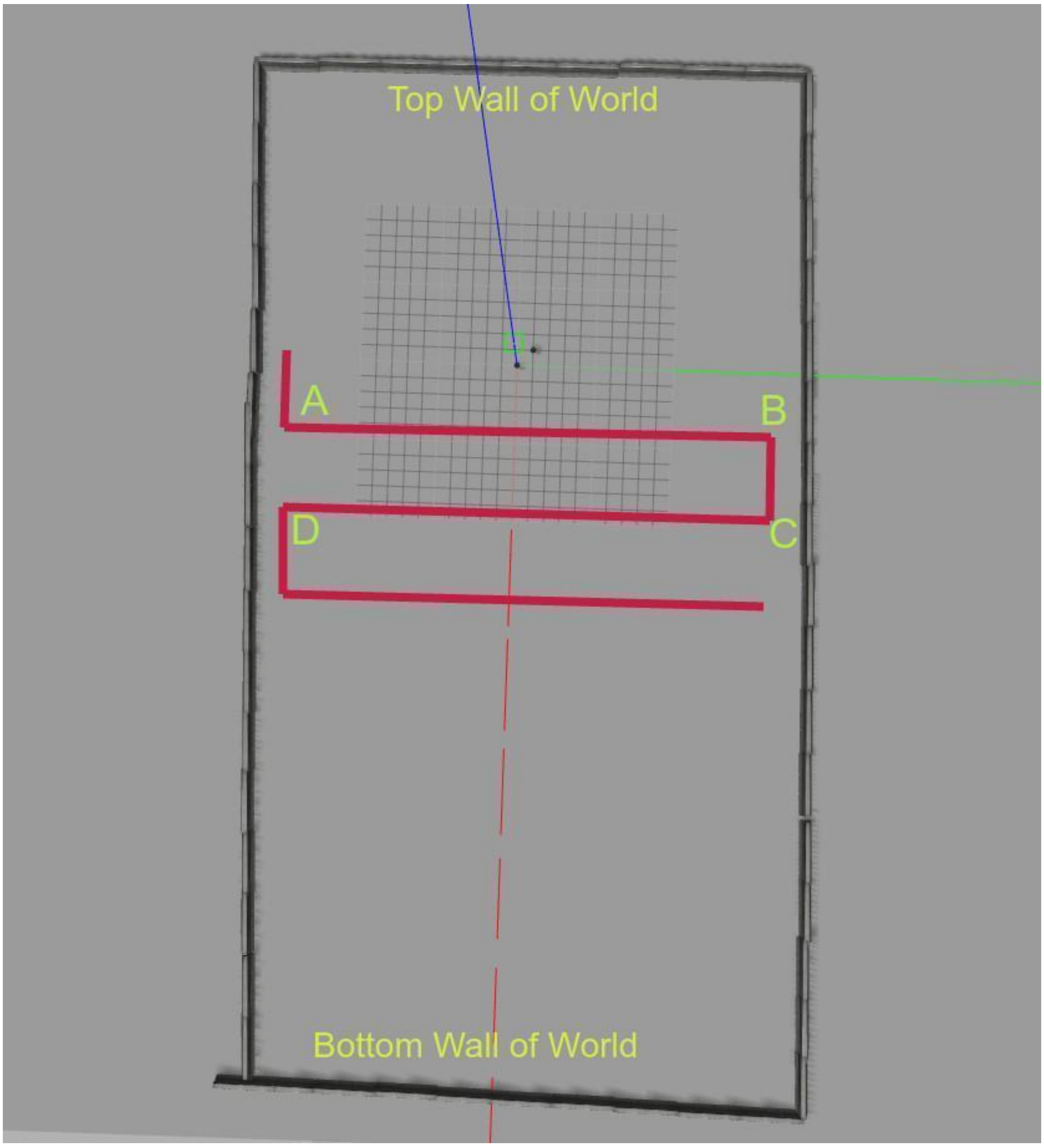


Figure *23* - Inner world scan

When the turtlebot reaches the “end” of the world (meaning the top or bottom wall), it reruns the algorithm explained before, but this time it goes in the reverse direction.

4.3.4.1 Two turtlebots running meander algorithm

The experiments contain tow turtlebots, which raises the need of a mechanism to make sure that these two turtlebots will not crash on each other. In this algorithm the two turtlebots do not have to communicate with each other except for this reason. In order to prevent a crash of the tow turtlebots, each turtlebot, when it scans an obstacle with laserscan data, they check the position of the other turtlebot. If the other turtlebot is blocking its path, then the turtlebot A tries to move around the turtlebot B, while the turtlebot B is staying still. In the following figure, Figure 24, the process of avoidance is shown.

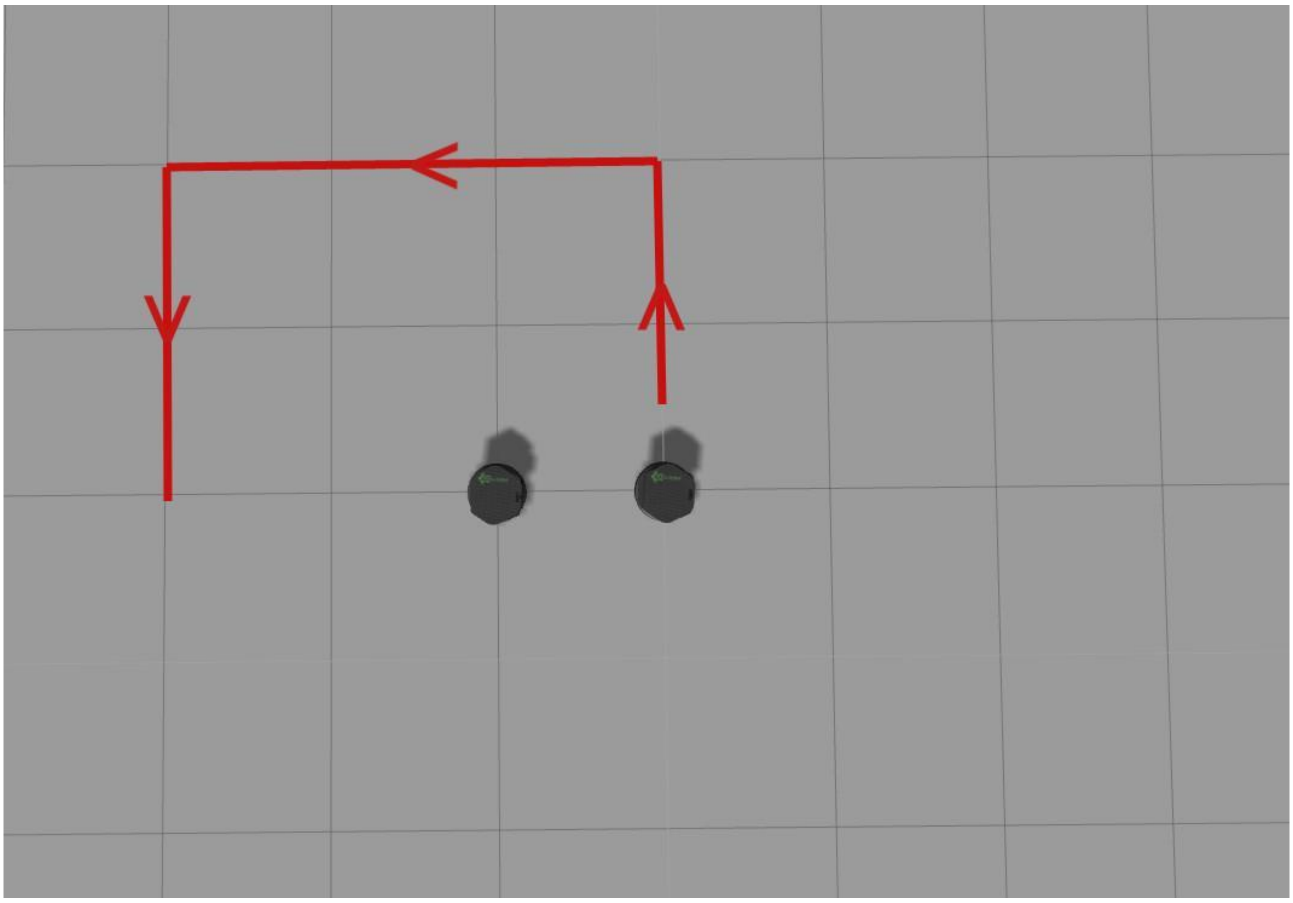


Figure 24 - Turtlebot crash avoidance

It has to be noticed that the crash avoidance could not depend only on laser scan data because both turtlebots move. This means that the position of each obstacle changes every moment, so the one turtlebot could not just move around the other turtlebot, in order to avoid it, because its position might have changed. A more sophisticated algorithm for moving obstacles avoidance could have been used, however this is not the purpose of this thesis.

4.3.5 Particle Swarm Optimization (PSO)

In computational science, particle swarm optimization (PSO) [18] is a computational method that tries to find the best solution by constantly improving the current solution. For each solution found a quality measure is been corresponded. In this way, the best solution is the one with the greader quality. This method is applicable to problems with plenty of possible solutions that have a certain quality.

The inventors of the PSO are Kennedy, Eberhart and Shi [19] who firstly used it for the simulation of social behavior. After the simplification of the algorithm it has been noticed that it performs optimization.

PSO can be characterized as metaheuristic as it makes a few or no assumptions about the problem that tries to optimize. Also, it can search in large spaces of candidate solutions. As all metaheuristic algorithms PSO does not guarantee that the final solution will be the optimal solution.

4.3.5.1 Algorithm

PSO algorithm works in a population (swarm) of feasible solutions (particles). The particles are navigating in the search environment based on the local and global knowledge of the environment. The local knowledge is the knowledge that the specific particle has gained on its own and the global knowledge is the knowledge that other particles shared. When a new better solution is found from a particle, the whole swarm is guided towards it. This process is constantly repeating and the particles are constantly improving the current solution. At the end of the algorithm, hopefully an acceptable solution will have been found.

```
for each particle $i = 1, ..., S$ do
   Initialize the particle's position with a uniformly distributed random vector: $\mathbf{x}_i \sim U(\mathbf{b}_{lo}, \mathbf{b}_{up})$
   Initialize the particle's best known position to its initial position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
   if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
       update the swarm's best known  position: $\mathbf{g} \leftarrow \mathbf{p}_i$
   Initialize the particle's velocity: $\mathbf{v}_i \sim U(-|\mathbf{b}_{up}-\mathbf{b}_{lo}|, |\mathbf{b}_{up}-\mathbf{b}_{lo}|)$
while a termination criterion is not met do:
   for each particle $i = 1, ..., S$ do
      for each dimension $d = 1, ..., n$ do
         Pick random numbers: $r_p, r_g \sim U(0,1)$
         Update the particle's velocity: $\mathbf{v}_{i,d} \leftarrow \omega\ \mathbf{v}_{i,d} + \varphi_p\ r_p\ (\mathbf{p}_{i,d}-\mathbf{x}_{i,d}) + \varphi_g\ r_g\ (\mathbf{g}_d-\mathbf{x}_{i,d})$
      Update the particle's position: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{lr}\ \mathbf{v}_i$
      if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ then
         Update the particle's best known position: $\mathbf{p}_i \leftarrow \mathbf{x}_i$
         if $f(\mathbf{p}_i) < f(\mathbf{g})$ then
            Update the swarm's best known position: $\mathbf{g} \leftarrow \mathbf{p}_i$
```

Figure 25 – PSO algorithm

In the algorithm above, f stands for the cost function that must be minimized. The goal of the algorithm is to find a solution a for which *f*(a) ≤ *f*(b) for all b in the search environment. This will mean that a is the optimal solution. S stands for the number of particles in the swarm. The values $\mathbf{b}_{lo}$ and $\mathbf{b}_{up}$ represent the lower and upper boundaries of the search-space respectively.

The algorithm ends when an acceptable solution is found, or a certain number of iterations has been performed. The parameters ω, φ_p, and φ_g are selected by the practitioner and control the behaviour and efficacy of the PSO method. **Lr** represents the learning rate (0 ≤ **lr** ≤ 1.0), which is the proportion at which the velocity affects the movement of the particle (where **lr** = 0 means the velocity will not affect the particle at all and **lr** = 1 means the velocity will fully affect the particle).

In our problem PSO can be applied so that the turtlebots work cooperatively to find the best measurement in their world. A basic variant of PSO can be used for this cause. In this algorithm we have a population of candidate solutions (swarm), that in our case is the total of the positions in the world and the robots (particles) that try to find the best solution. The particles are moving around the world guided by the measurements that they receive in the position that they stand at each time and the measurements that the other robot has found. When a better solution has been discovered, meaning a better measurement in a position, all the particle-robots will move towards this new optimized position. This process is being repeated until the robots cannot find a better solution. However, it is not guaranteed that the found solution will be the optimized solution.

4.3.6 Multi-Vehicle Path Planning based on PSO

The second algorithm that we developed for this Thesis is a combination of the meandre algorithm described before and the PSO logic. The basic logic of the algorithm is the following. At first the two turtlebots start executing the meander algorithm. When a turtlebot finds a source, it tries to follow the measurements with the purpose to find the best measurement of this source. In the meanwhile, it informs the other turtlebot that a measurement has been found. When the other turtlebot sees that the first turtlebot has found a source, it starts moving to its position in order to go as close as it can get.

This algorithm also uses the node ‘Publish Metrics’. To implement the second algorithm, we additionally created 2 different nodes for each turtlebot. Firstly, it is easily understandable that for this algorithm to work it is necessary for each turtlebot to know the exact position of the other and the best measurement noted. For this purpose, we created the nodes 'robot1’ and ‘robot2’. Each node runs a program that reads the position of the turtlebot from the topic ‘robotX/odom’ (where X is 1 or 2 respectively) and finds the measurement in this position using the topic ‘/metrics’. Finally, if the metric found in this position is better than the previous measurements, it publishes a new message of type Best_Metric to the topic ‘robotX/local_best’(where X is 1 or 2 respectively). The message that it publishes contains the measurement found and the position. As a result, this program makes it easy to access the best measurement that a turtlebot had and the position that the measurement was found, simply by subscribing to a topic.

The second node that was implemented contains all the other parts of the algorithm. The nodes ‘Meandre1’ and ‘Meandre2’ run a program that at first is similar to the meander without PSO. The differences are noticed when one turtlebot finds a measurement greater than 0. In this case it tries to go as close to the source as possible. In order to do that, it finds the best measurement among the front right and left measurements. If the best metric is in front of the turtlebot, it simply continues going forward. If the best measurement is right or left, then the turtlebot turns right or left respectively and then goes straight. At this point it is important to comment on the fact that there is a possibility that the turtlebot will not find the best measurement of the source that it examines. This is because the turtlebot scans only three of the possible positions and goes to the one with the best measurements. However, we can say surely enough that the turtlebot will find an acceptable measurement of the source.

While trying to approach the source, the turtlebot has to consider the possibility of the other turtlebot being in the position that it wants to go. In this case, the first turtlebot takes into account only the other two available positions. That means that if for example the turtlebot A is looking for the best measurement and the turtlebot B is at the position exactly in front of the turtlebot A, then the turtlebot A will check only the right and left positions. Maybe the best measurement would have been in the unavailable position, where the turtlebot B was standing, but this does not have an effect on the result of the algorithm because the turtlebots work as a team and we do not mind which turtlebot will eventually find the best source.

The turtlebot that found the measurement, informs the other turtlebot to follow it. The second turtlebot then reads from the 'robotX/local_best' topic the position of the first turtlebot. It calculates the orientation needed to face the other turtlebot and starts moving forward. During its motion towards the other turtlebot, it still keeps track of the measurements found in its path. If it finds a measurement greater than 0, it stops following the first turtlebot, and it tries to reach the source on its own. In this way, if there are many sources in the world, we have the opportunity to explore at least two of them and eventually find which one is the best. Moreover, while turtlebot A is going to the position that the turtlebot B pointed, it keeps reading the measurements posted by the turtlebot B. If turtlebot B publishes a measurement significantly better than the previous one, then the turtlebot A updates it's goal destination with the new position of the turtlebot B.In Figure 25 the path of the tow turtlebots is shown.

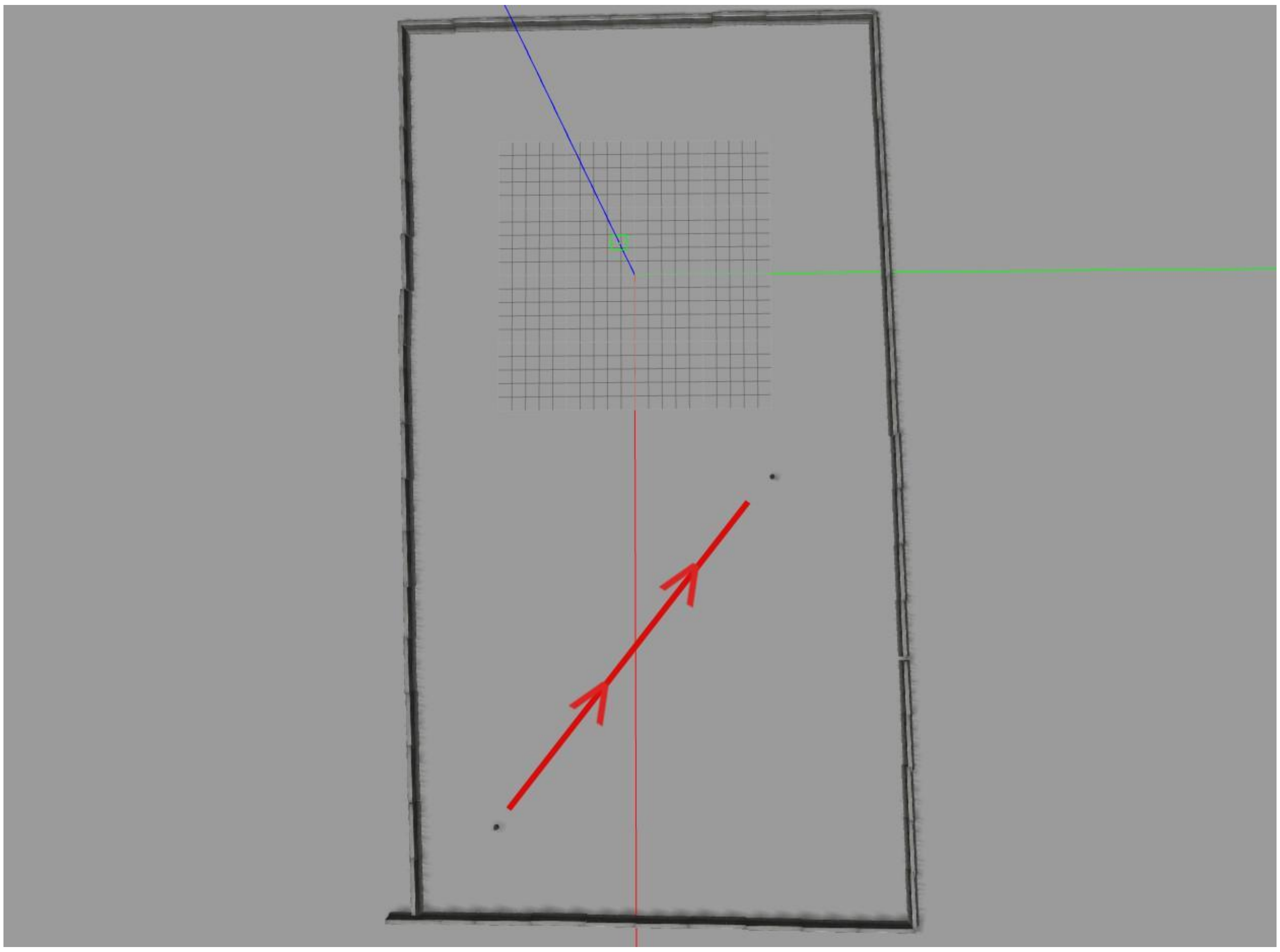


Figure 25 - The path of turtlebot

5 EXPERIMENTS

5.1 Set up Experiments

5.1.1 Create custom world

For the purposes of this thesis we created a custom world file, using Gazebo tools. Although the default ROS package for gazebo contains a variety of different world files, we decided to build one on our own so that it suits perfectly our needs and it tests our algorithms in all possible cases.

Building a custom world in gazebo is simple. We launch the gazebo simulator with the default world, using the command roslaunch turtlebot_gazebo turtlebot_world.launch. After the gazebo simulator is loaded completely, the screen of the computer looks like Figure 26

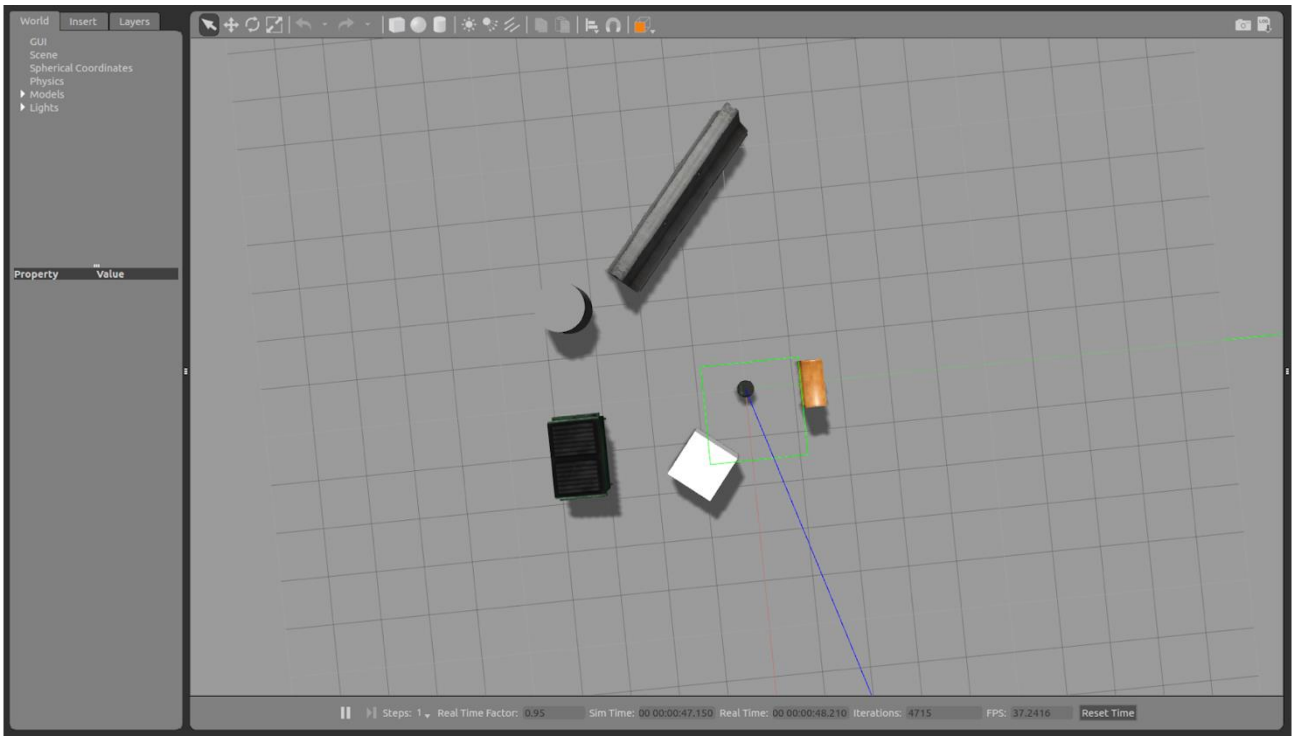


Figure 26- Gazebo simulator with default world file

The next step is to delete the items of the world that we do not need in our new world, simply by selecting the item and pressing Delete button. In our case we do not need any obstacle of the default world so we deleted them all. Insert new obstacles from the menu and place them at the desired positions. When the world is completed save the world using the menu of the gazebo. In this way, gazebo creates a .world file that we can use for our later gazebo launches. In this thesis, the world used for out experiments is the one shown in Figure 27.

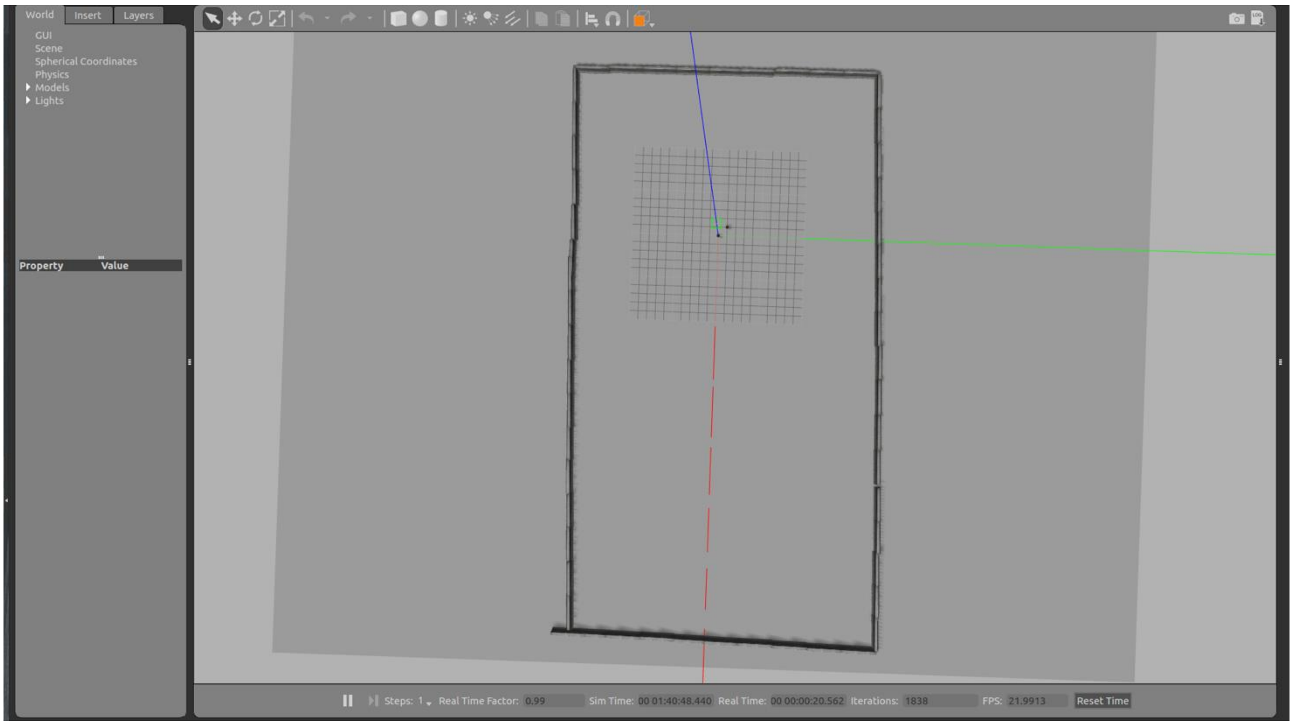


Figure 27 - gazebo simulator with the custom world file

5.1.1.1 Map the custom world

For experiments needed for this Thesis, a map of the world that the simulators launch is necessary. As mentioned before, the map of the world will be used as a base on which we will build a map of measurements. This map of measurements represent the measurements that a sensor on the turtlebot could receive. The reasons that lead us to create this map of measurements are firstly that as the experiments are executed in the simulators, real measurements of turtlebot’s sensors could not be used and secondly the measurements contained in the map are created to follow a normal distribution. Controlling how the measurements are shared in the room is essential for the research and the final results of this Thesis.

The mapping of the world that we built is based on the gmapping package of ROS. The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Generally, in order to build a map, we need to transform the laserscan and pose data that the turtlebot receives and transform them in such way to create a 2-D occupancy grid map. Slam_gmapping does all this work. We launch the gmapping package simply by typing in a terminal the command ‘roslaunch gmapping slam_gmapping scan:=scan’. In another terminal we launch gazebo with our custom world and drive around the turtlebot, using the turtlebot_teleop package. When we cover all the world with the turtlebot, we save the map using the map_server, by typing the command ‘rosrun map_server map_saver -f <mpa_name>’. The map of our custom world is shown in Figure 28.

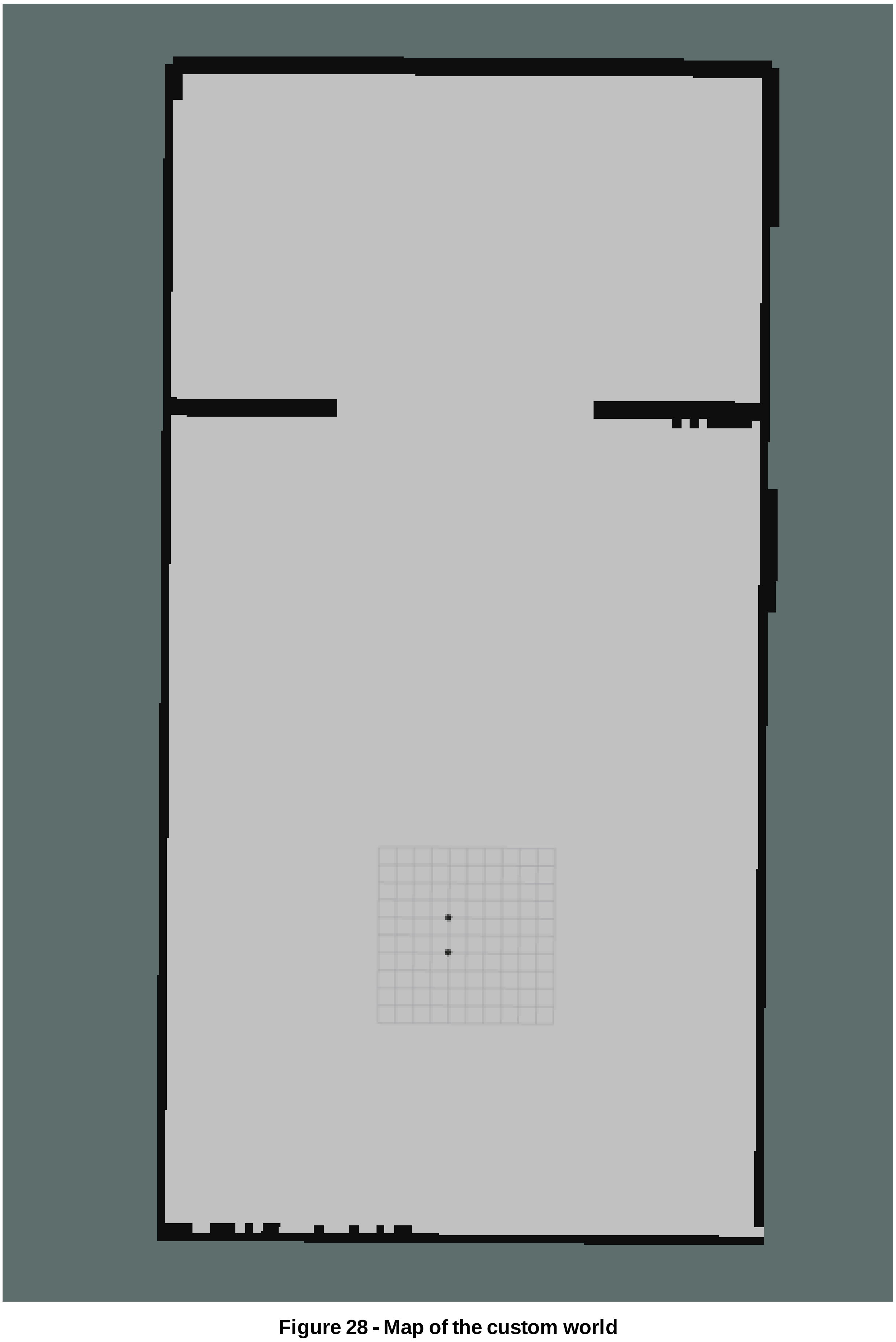


Figure 28 - Map of the custom world

5.2 Experiment execution

The performance of our two developed algorithms has to be tested under many different circumstances. For this reason we carefully designed different cases in which the algorithms should work. We executed experiments with different sources of measurements and different initial poses of the turtlebots. In some experiments the source of the measurements was not reachable from the turtlebots and they had to try to go as close as possible without crashing on any obstacle. In other experiments we added multiple sources with different measurements to check how the turtlebots will act and whether the turtlebots will explore more than just one source.

The first experiment was designed to check the most common case, in which we have only one source and it is reachable from both turtlebots. The initial position of the two turtlebots is at the middle of the world and they were pointed at different walls. The initial state of this first experiment is shown in the next figure (Figure 29) and the red circle show where the source is.

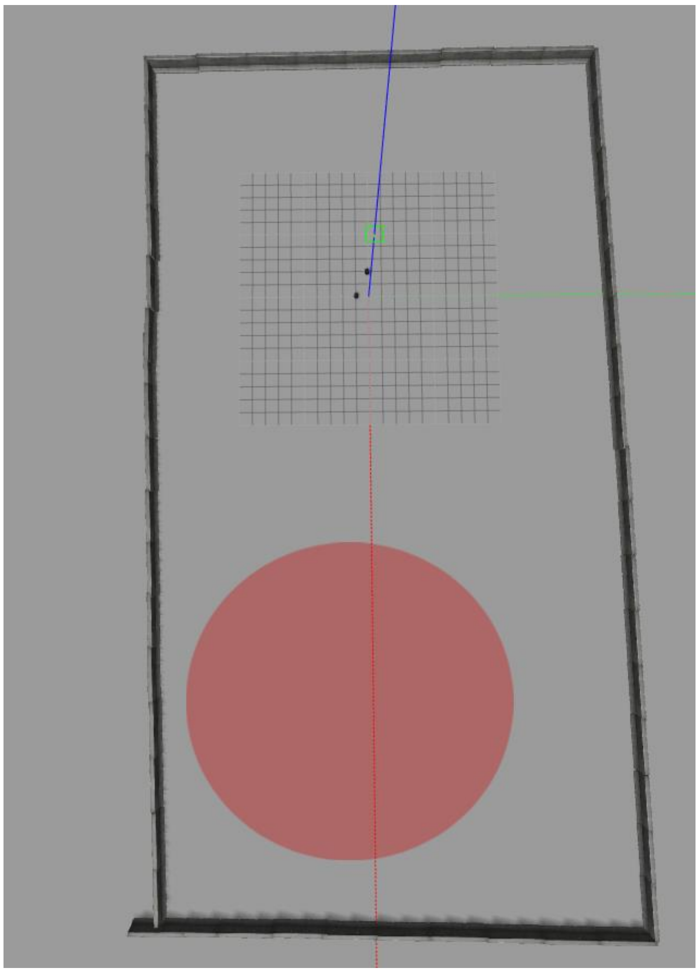


Figure 29 - Initial state of first experiment

In the second experiment the case of a hidden source was tested. The measurement map contains two sources and one of them is partly hidden. In this case the turtlebots are able to reach all measurements from one source and some, but not the best one of the second. The turtlebots have to try to find the best reachable measurement and of course not crash on any obstacle. The initial state of the experiment is shown in the following figure (Figure 30). The red circle stands for each source.

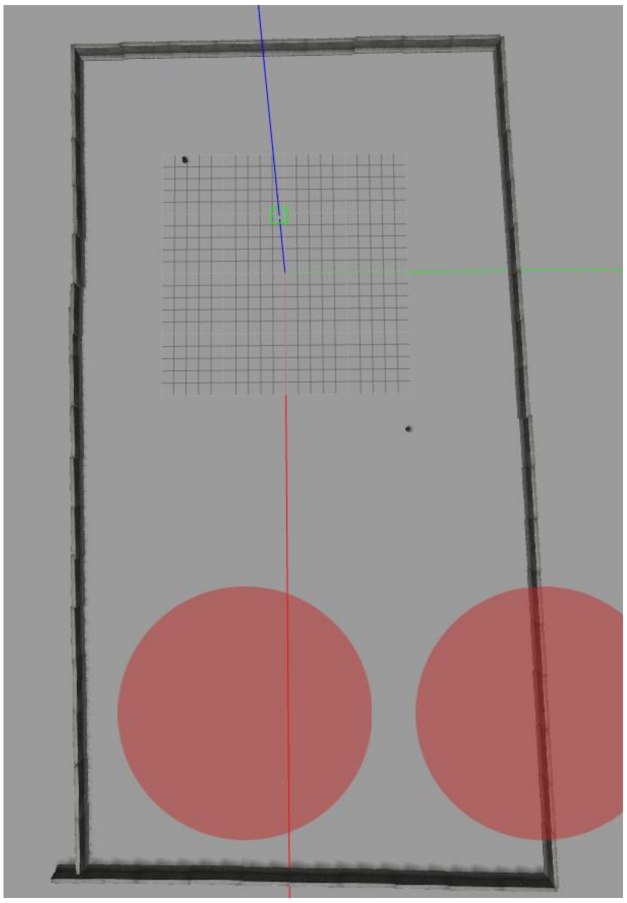


Figure 30 - Initial state of second experiment

The measurement map used for the second experiment is shown in the following figure (Figure 31).

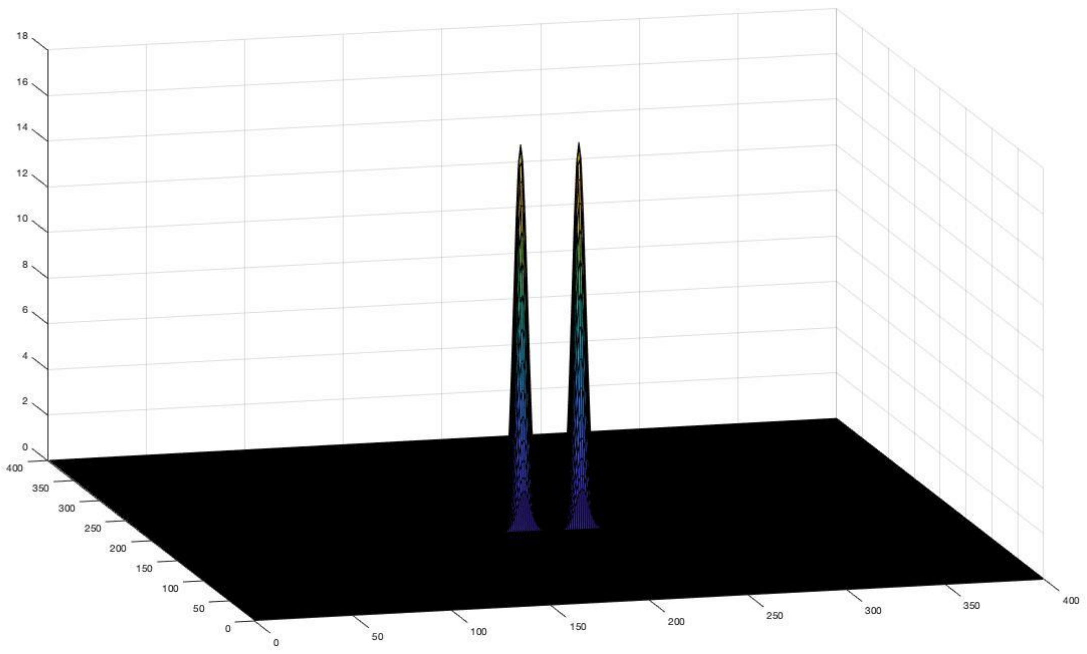


Figure 31 - Map of measurements with two sources

The two sources are close to each other without covering one another. The two sources have the same best measurement, however only the one optimal measurement is reachable from the turtlebots.

In the rest two experiments, we added more sources of measurement in the world. The expected result from the algorithm without PSO, is to explore all the sources and find measurements from all of them, but maybe not the best ones. The meander with PSO is the tricky part of this experiment. As described before, in this algorithm when the first turtlebot finds a measurement, despite its quality, it informs the other turtlebot to follow it. This means that the second turtlebot might be near a better source but not have reached a measurement yet, and it will not continue its way but it will follow the first turtlebot that might have found a weaker source. On the other hand, both turtlebots will explore a source in the minimum time. The following figures show the circumstances of each experiment (Figure 32 and 33).

For the third experiment, we used the measurement map shown in Figure 34. The map contains three sources. The two of them are identical and the third is a smaller one, meaning that it has a smaller best measurement. The smaller source is connected with one of the big sources.

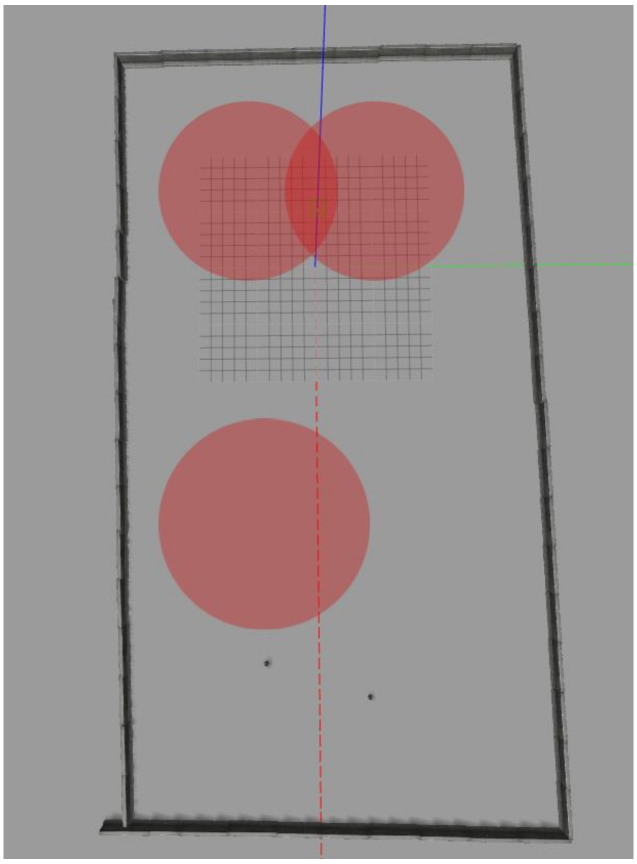


Figure 32 -Initial state of third experiment

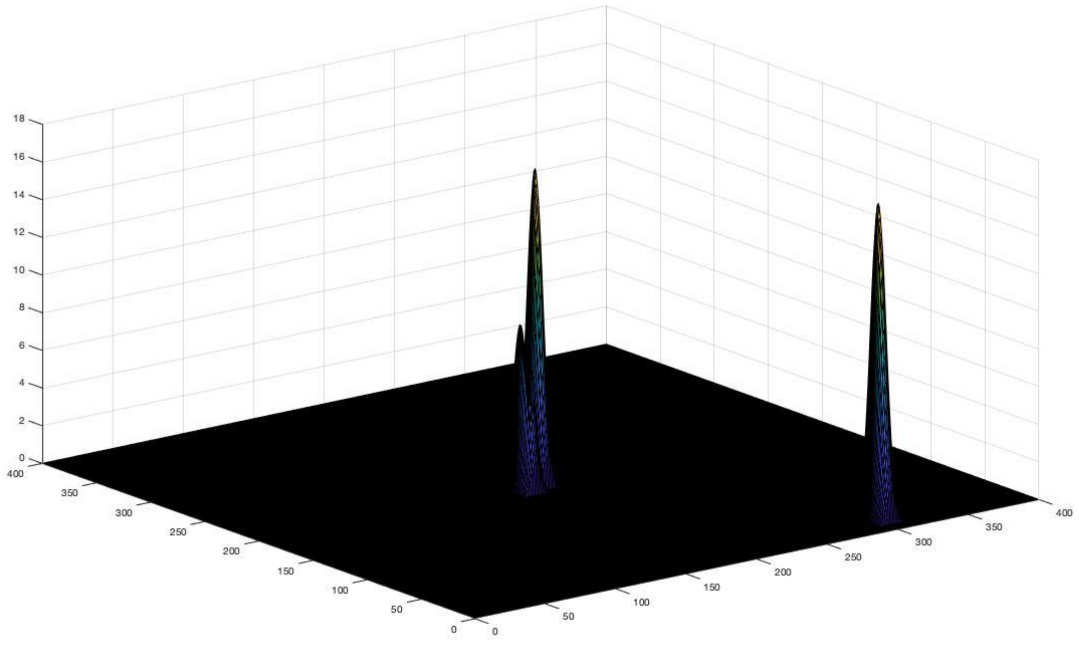


Figure 33 - Map of measurements with three sources

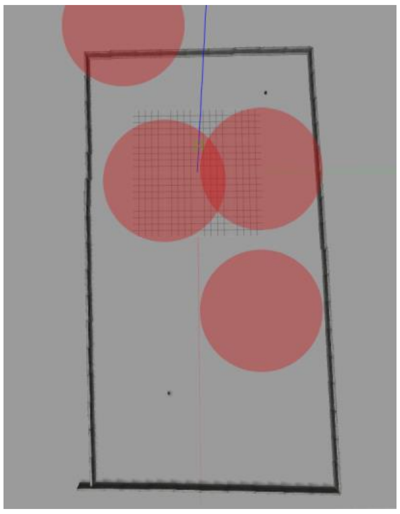


Figure 34 - Initial state of forth experiment

In the final experiment we used the map of measurements shown in figure 35. The map contains four sources, three identical, big sources and a smaller one that is connected with one of the bigger ones, as in the previous experiment. The one big source is not reachable by any turtlebot, as it is outside the walls of the world.

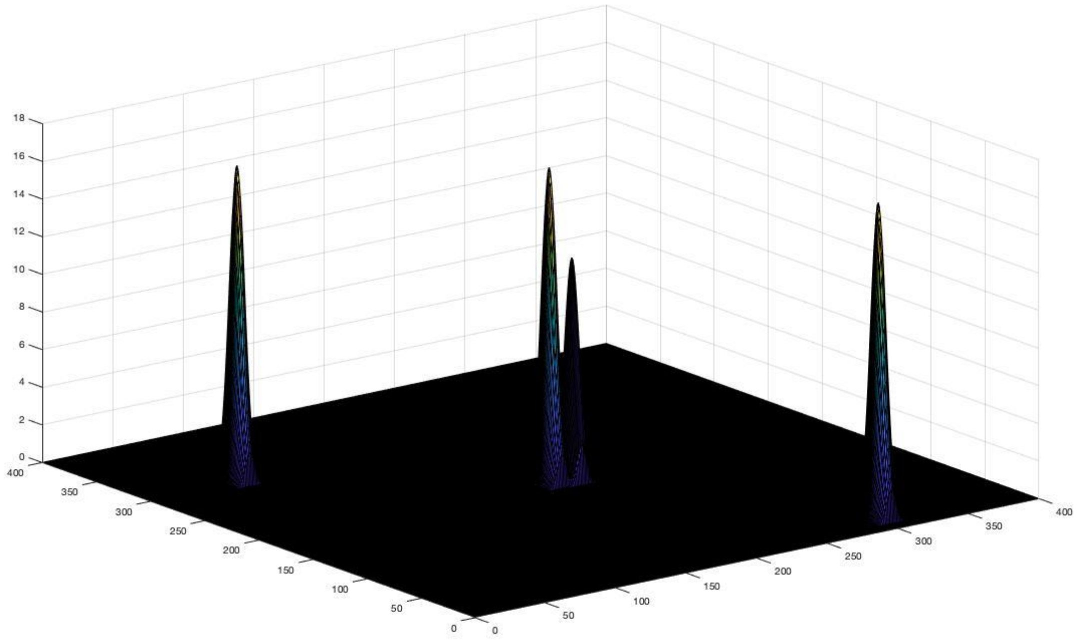


Figure 35 - Map of measurements with four sources

5.3 Experimental results

In the first experiment, there is only one source of measurements in the world. The source is fully reachable by both turtlebots. The results of this experiment show that in both algorithms, meander without PSO and meander with PSO, the turtlebots eventually reach the optimal source measurement. We also noticed that the time that the turtlebots need in order to find that optimal solution is significantly smaller in the case of PSO algorithm. As shown in the next diagrams, the time consumed by the PSO algorithm is almost four times less than the time consumed by the exhaustive meander scan algorithm.

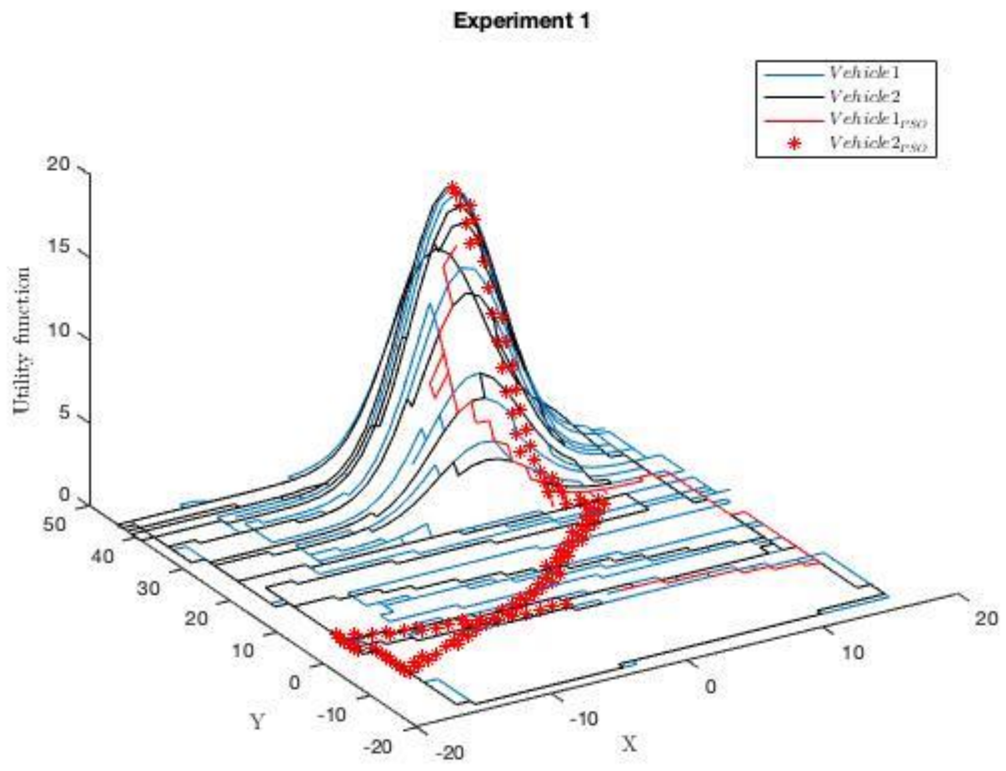


Figure 36 - Utility function in space for the first experiment

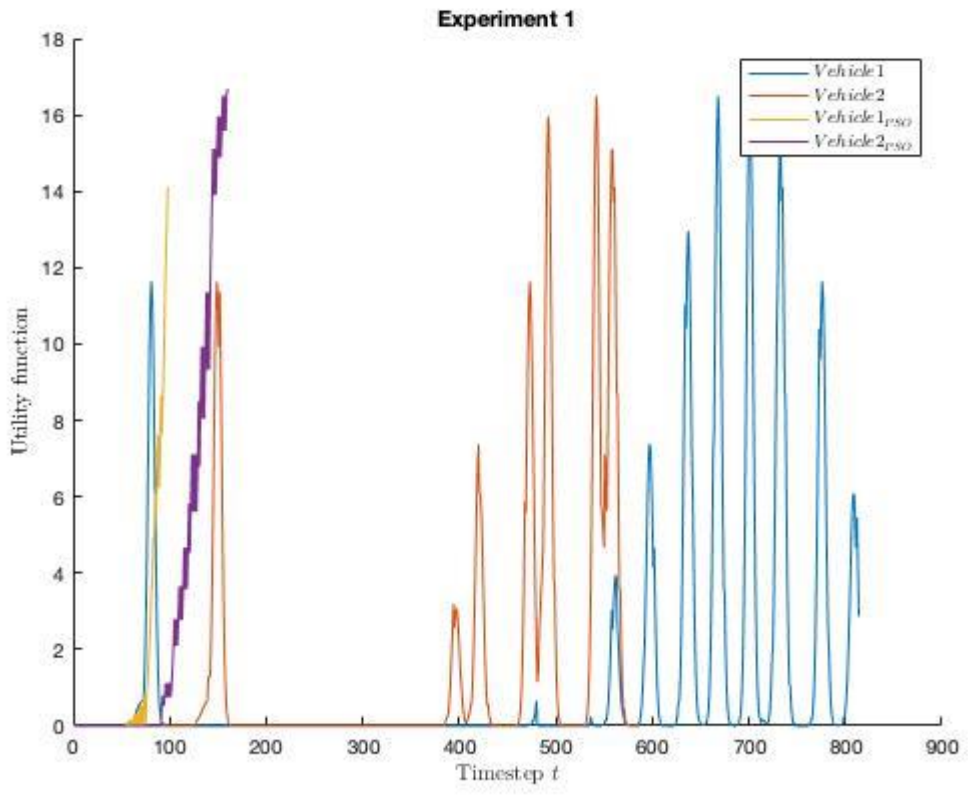


Figure 37 -Utility function in time for the first experiment

In the second experiment we have two sources in the world. In the execution of the exhaustive meander scan algorithm both turtlebots find measurements from both sources, while the PSO algorithm explores only the one source. Both algorithms find the optimal measurement in the end. In this experiment we noticed that the time needed to find the optimal solution does not differ between the two algorithms. As shown in Figure 39, the vehicle1 in the meander without PSO algorithm finds the optimal solution at the same time as the vehicle2 in the meander with PSO algorithm.

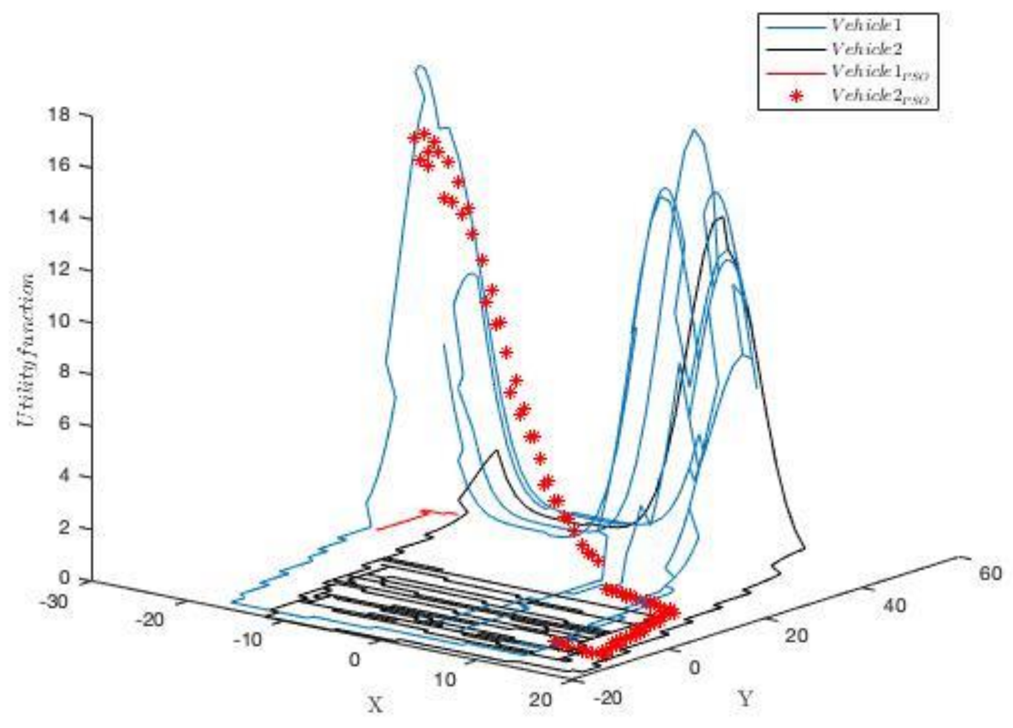


Figure 38 - Utility function in space for the second experiment

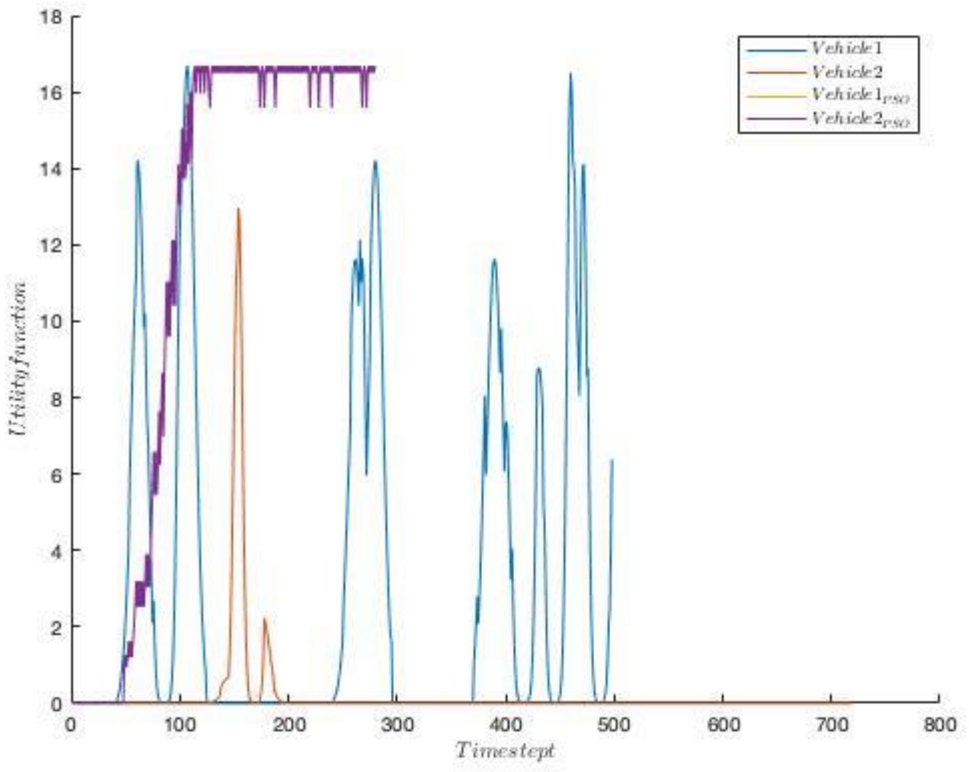


Figure 39 - Utility function in time for the second experiment

In the third experiment the results are similar to the second experiment. We noticed that the two turtlebots understand the two connected sources as one and when executing the PSO algorithm, the two turtlebots search for the best metric in the big and the smaller source. In the PSO algorithm, however the turtlebots never read measurements from the isolated source. This source is fully explored when executing the exhaustive algorithm.

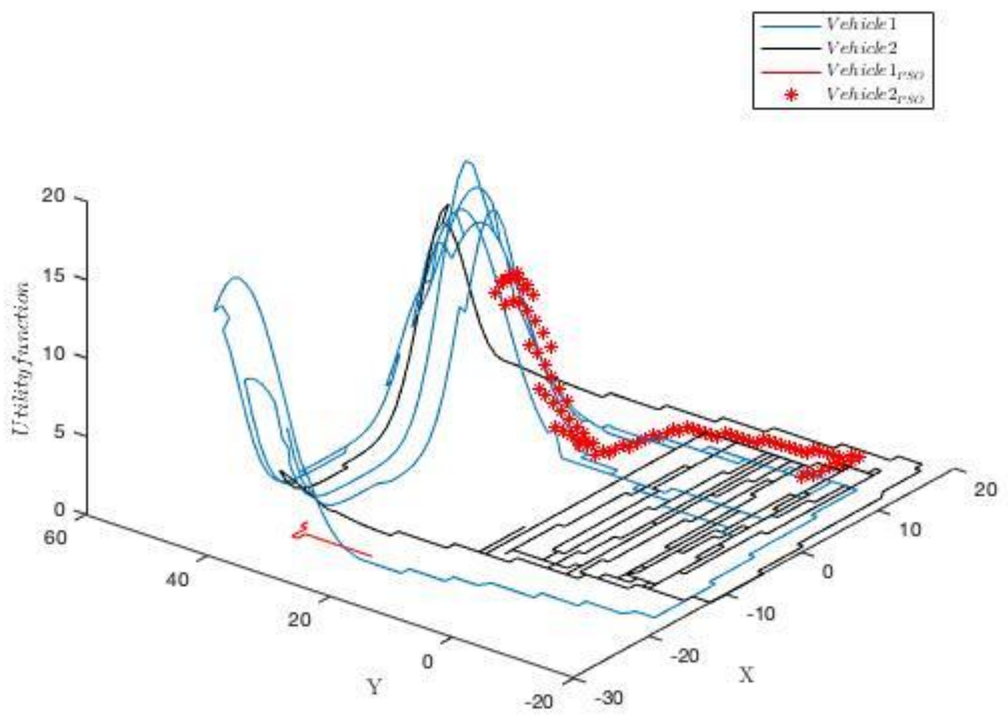


Figure 40 - Utility function in space for the third experiment

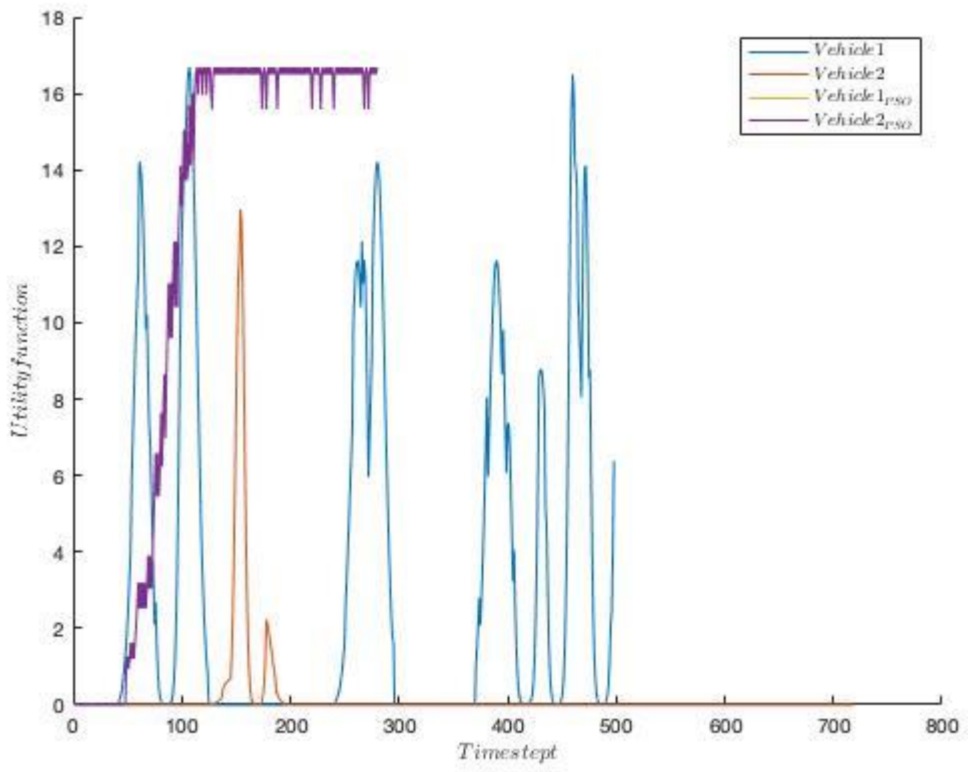


Figure *41* **- Utility function in time for the third experiment**

In the final experiment, in both algorithms the turtlebots find the optimal solution. However, the time consuming by the exhaustive algorithm is around six times greater than the PSO algorithm.

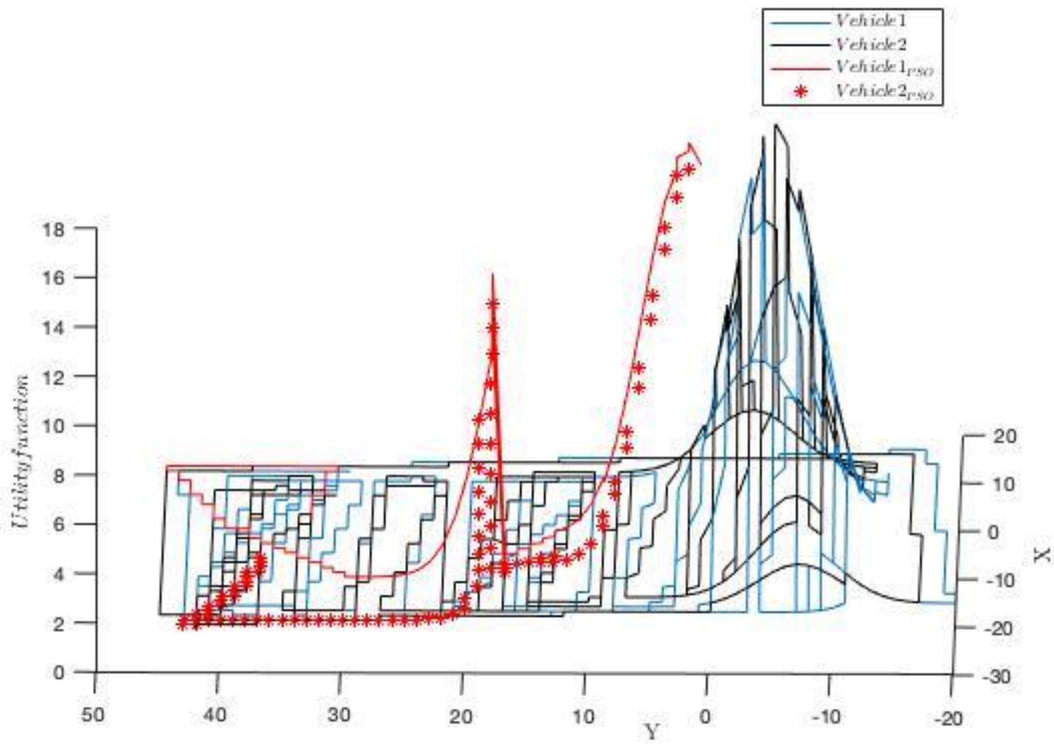


Figure 42 - Utility function in space for the fourth experiment

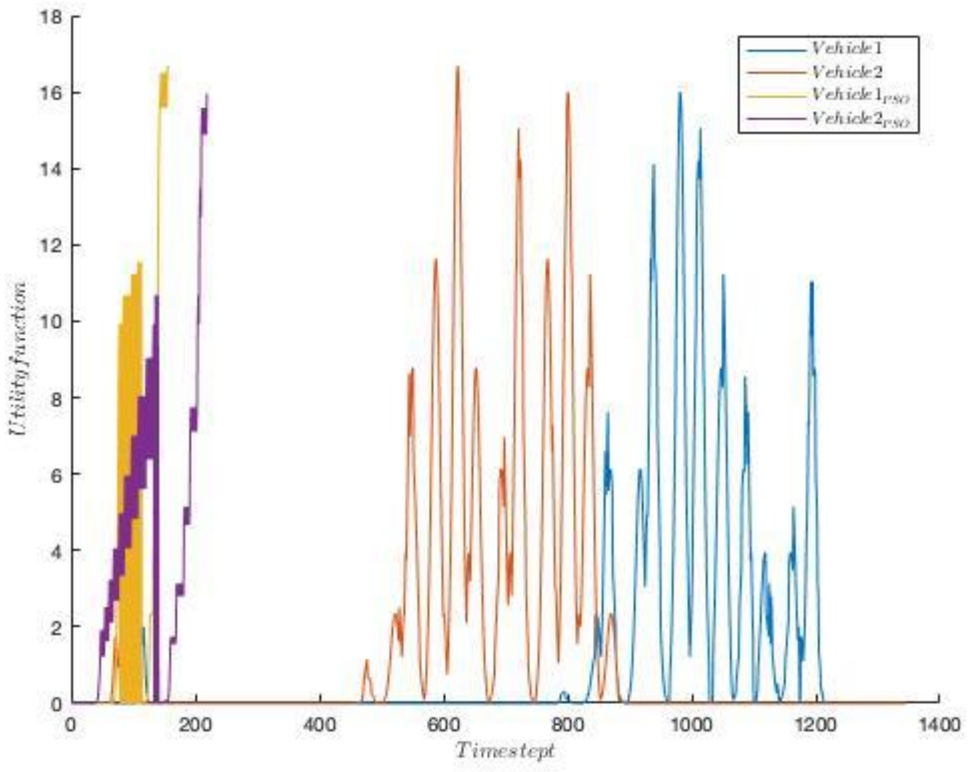


Figure 43 - Utility function in time for the fourth experiment

In the following figures (Figure 44 and 45) the maximum utility function for each turtlebot in each algorithm.

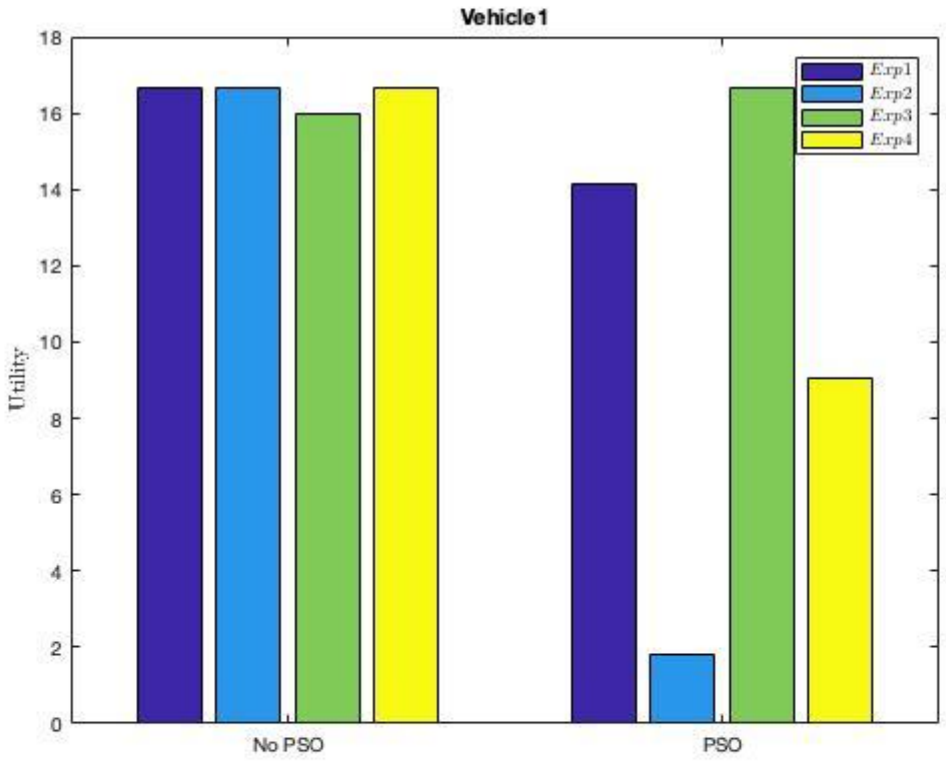


Figure 44 - Utility function of vehicle 1

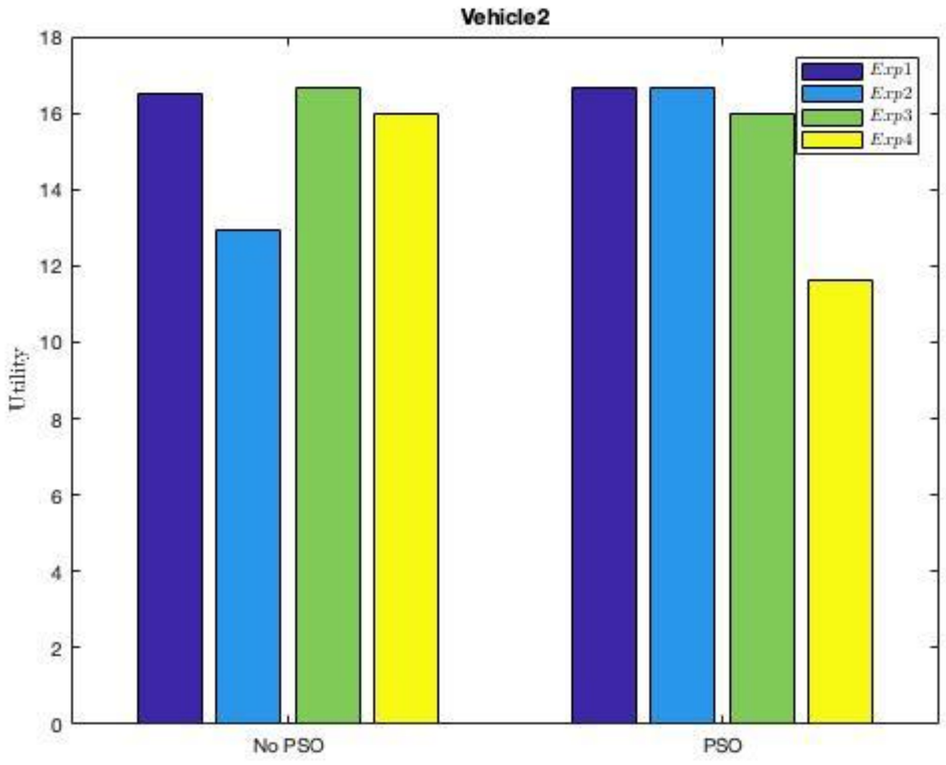


Figure 45 - Utility of vehicle 2

In the exhaustive algorithm the first turtlebot finds the optimal solution in the three out of the four experiments, and in the third experiment, in which it does not find the optimal solution, it reaches an acceptable one. The second turtlebot find the best solution in the first and third experiment. In the other two experiments it reaches a good enough measurement. As a team the two turtlebots in the exhaustive algorithm always find the optimal solution. In the experiments with the PSO algorithm we notice that the first turtlebot finds the best solution only in the third experiment. Moreover, the second turtlebot find the optimal solution in the first two experiments. As a team the two turtlebots reach the best solution in the first three experiments. In the last experiment we notice that in the end none of the turtlebots reach the optimal solution, however both reach acceptable measurements.

At this point it is important to focus on the time consuming by the two algorithms in each experiment. The following diagrams (Figure 46 and 47) show the time needed for each turtlebot to find its best measurement in every experiment. Even if the utility reached from noPSO and PSO methods are close enough this does not stand for the time. The time that PSO robots conclude in an acceptable solution is more than 1/8 of the exhaustive algorithm. For example, the time needed in experiment 3 from the noPSO robots is close to 1200s while the relevant time needed from PSO UxVs is close to 200s. Let us consider in larger or more complicated spaces then the PSO algorithm can help the UxVs to detect really fast a sensor source target.

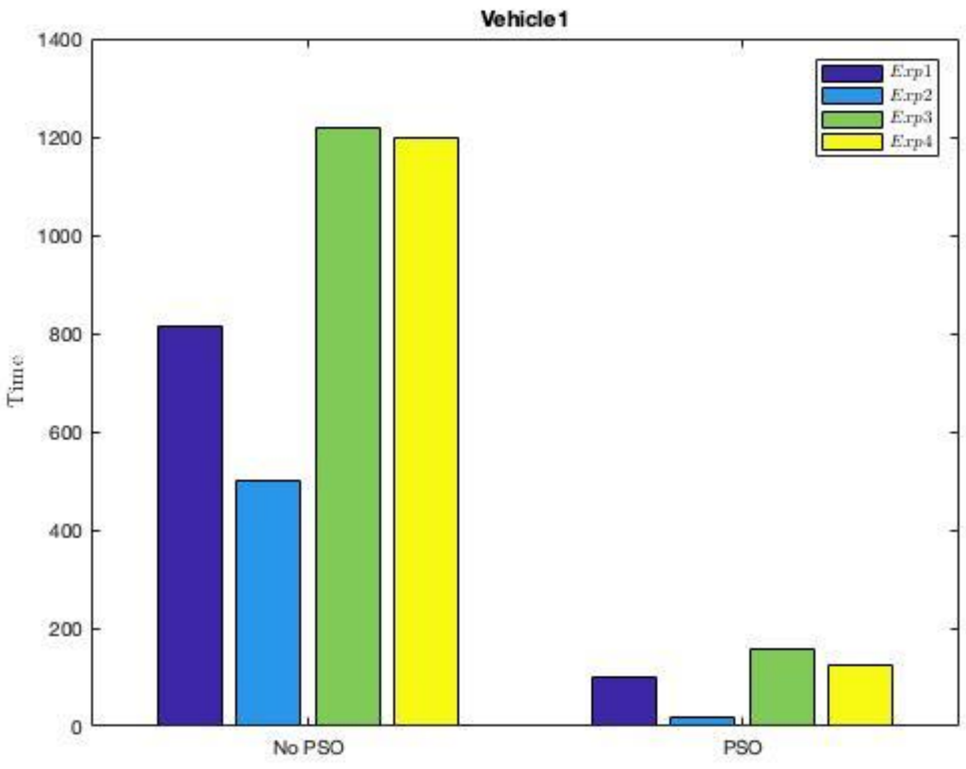


Figure 46 - execution time of vehicle 1

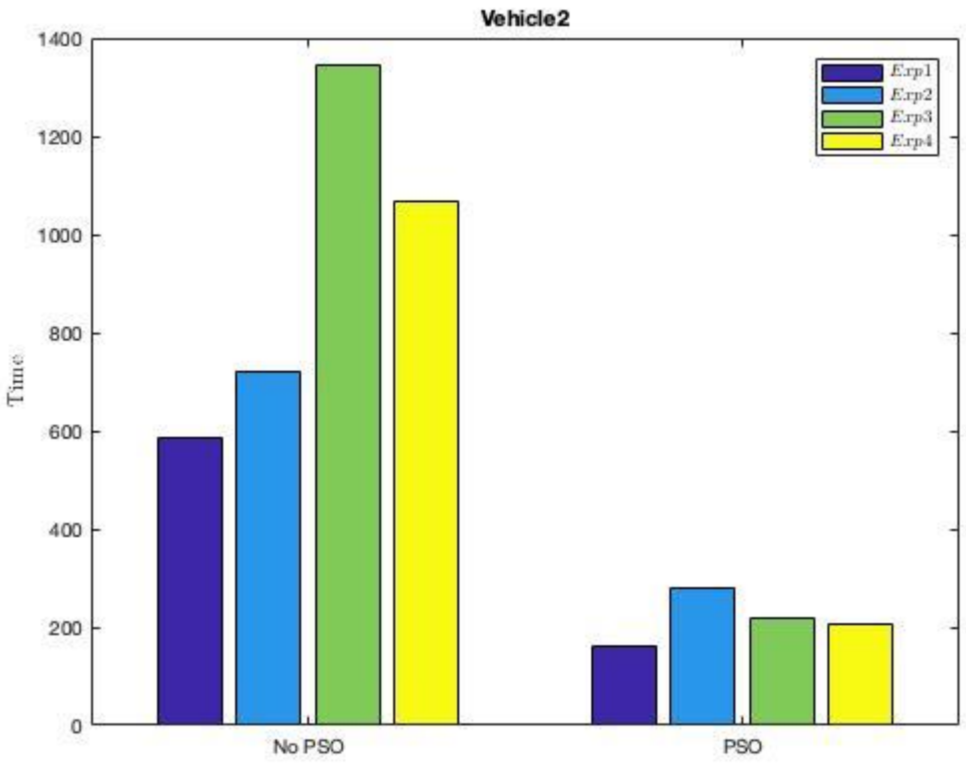


Figure 47 - execution time of vehicle 2

6 CONCLUSION

The aim of this Thesis was to test whether the use of PSO in a collaborative UGV world exploration is beneficial or not. In more details, we stated the problem of finding randomly placed sources of measurements in an environment, using multiple UGVs. We tested the case where the UGVs act individually and the case where the UGVs collaborate and exchange messages. For this reason, we developed two algorithms, the noPSO and PSO algorithm. In the noPSO algorithm the UGVs scan exhaustively the entire environment, without communicating with each other, following a meander path. In the second algorithm the UGVs start by scanning the environment with the same logic as in the first algorithm. But when the one UGV finds a measurement, it follows it with the aim of finding the optimal measurement of the source. Additionally, it notifies the other UGVs to follow it. This is the part where the PSO theory takes place.

To test the performance of these two algorithms, we executed multiple experiments under different circumstances. These experiments were executed in the Gazebo simulator with two virtual turtlebots. Our experiments show on the one hand, that the noPSO algorithm finds all the different sources and the optimal measurement. However, the execution time of this algorithm is in most cases unacceptable. On the other hand, the PSO algorithm is more time efficient and more acceptable for UxV operations, although it does not find the optimal solution every time, but it always ends up with a good enough solution.

The algorithms presented in this Thesis have good results and can be used in many cases. The two algorithms have been developed with different principals in mind. The exhaustive meander algorithm has been developed in such way to find the best solution in the environment under any circumstances. On the other hand, the PSO algorithm is developed for cases in which the time constraints are significant. The previous diagrams make it clear that in the PSO algorithm the turtlebots converge much faster to a solution than the exhaustive algorithm. The solution given by the PSO algorithm is not every time the optimal but it is always an acceptable one. In an environment that amount of time consumed is a crucial issue, the PSO algorithm is the algorithm to go. If there are no time constraints and the best solution is needed, then the exhaustive algorithm should be chosen. However, to choose the right algorithm we have to consider that the time consumption in any UxV mission is a crucial issue. The more time the UxV runs, the more resources are used.

ABBREVIATIONS - ACRONYMS

IoT	Internet of Things
UxV	Unmanned Vehicle, x stands for aerial, ground, or sea
UAV	Unmanned Aerial Vehicle
UGV	Unmanned Ground Vehicle
USV	Unmanned Surface Vehicle
PSO	Particle swarm optimization
ROS	Robot Operating System

ANNEX I

The source code of this thesis can be found by following the link:

https://github.com/ChristinaKats/ROS

REFERENCES

[1] *Sun W, Tang M, Zhang L, Huo Z, Shu L. A Survey of Using Swarm Intelligence Algorithms in IoT. Sensors (Basel). 2020;20(5):1420. Published 2020 Mar 5. doi:10.3390/s20051420*

[2] A Sharma, P. Vanjani, N. Paliwal, C. Basnayaka, D. Nalin K. Jayakody, H. Wang, P. Muthuchidambaranathan, Communication and networking technologies for UAVs: A survey, Journal of Network and Computer Applications, Volume 168, 2020,

[3] Yan, R., Pang, S., Sun, H. *et al.* Development and missions of unmanned surface vehicle. *J. Marine. Sci. Appl.* **9,** 451–457 (2010). https://doi.org/10.1007/s11804-010-1033-2

[4] Zaman, Safdar & Slany, Wolfgang & Steinbauer, Gerald. (2011). ROS-based mapping, localization and autonomous navigation using a Pioneer 3-DX robot and their relevant issues. Saudi International Electronics, Communications and Photonics Conference 2011, SIECPC 2011. 1 - 5. 10.1109/SIECPC.2011.5876943.

[5] A. Martinez and E. Fernández, Learning ROS for Robotics Programming. Packt Publishing, 2013.

[6] Turtlebot, https://www.turtlebot.com/about/ [accessed 25/10/2020]

[7] Matthies L. (2014) Obstacle Detection. In: Ikeuchi K. (eds) Computer Vision. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-31439-6_52

[8] B. Lau, C. Sprunk and W. Burgard, Efficient Grid-Based Spatial Representations for Robot Navigation in Dynamic Environment,Robotics and Autonomous Systems, 61(10), 2013, pp. 1116-1130.

[9] Magid, Evgeni & Lavrenov, Roman & Afanasyev, Ilya. (2017). Voronoi-based trajectory optimization for UGV path planning. 383-387. 10.1109/ICMSC.2017.7959506.

[10] A. Stentz, "Optimal and efficient path planning for partially-known environments," Proceedings of the 1994 IEEE International Conference on Robotics and Automation, San Diego, CA, USA, 1994, pp. 3310-3317 vol.4, doi: 10.1109/ROBOT.1994.351061.

[11] Nilsson, N. J., “Principles of Artificial Intelligence”, TiogaPublishing Company, 1980.

[12] Fattah, Mohammad & Airola, Antti & Ausavarungnirun, Rachata & Mirzaei, Nima & Liljeberg, Pasi & Plosila, Juha & Mohammadi, Siamak & Pahikkala, Tapio & Mutlu, Onur & Tenhunen, Hannu. (2015). A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips. 10.1145/2786572.2786591.

[13] B.H. Koopman, Search and Screening: General with Historical applications, Pergamon Press, Elmsford, NY, 1980

[14] L.D. Stone, Theory of Optimal Search, 2nd edition, ORSA Books, Arlington, VA, 1989

[15] Wettergren, Thomas & Baylog, John. (2009). Collaborative search planning for multiple vehicles in nonhomogeneous environments. 1 - 7. 10.23919/OCEANS.2009.5422151.

[16] A. Ryan *et al.*, "Decentralized Control of Unmanned Aerial Vehicle Collaborative Sensing Missions," *2007 American Control Conference*, New York, NY, 2007, pp. 4672-4677, doi: 10.1109/ACC.2007.4282397.

[17] H. Saadaoui and F. El Bouanani, "Information sharing based on local PSO for UAVs cooperative search of unmoved targets," *2018 International Conference on Advanced Communication Technologies and Networking (CommNet)*, Marrakech, 2018, pp. 1-6, doi: 10.1109/COMMNET.2018.8360276.

[18] . T. Hafez, M. A. Kamel, P. T. Jardin and S. N. Givigi, "Task assignment/trajectory planning for unmanned vehicles via HFLC and PSO," *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, Miami, FL, USA, 2017, pp. 554-559, doi: 10.1109/ICUAS.2017.7991407.

[19] Eberhart and Yuhui Shi, "Particle swarm optimization: developments, applications and resources," Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546), Seoul, South Korea, 2001, pp. 81-86 vol. 1, doi: 10.1109/CEC.2001.934374.