UNIVERSITÁ DEGLI STUDI DI PADOVA

Corso di Laurea Magistrale in
INGEGNERIA DELL'AUTOMAZIONE

Implementation of distributed partitioning algorithms using mobile wheelphones

Relatore: Prof. Luca Schenato

Laureando: Schiesari Pietro

Anno Accademico 2016-2017

Abstract

Multi-robot coverage of an area is a fundamental problem in robotics. This thesis presents the implementation process of partitioning algorithms from the theorical ideas to sperimental results. In particular, the algorithms used are two versions of the classic Lloyd method based on *centering and partioning* for the computation of Centroidal Voronoid partitions. The differences between these versions are the communication architectures: client-server and peer-to-peer. In the first architecture the robots are allowed to comunicate with a central server-base station, whereas in the second one the robots communicate among neighboring peers. Therefore the tests have been conducted first on a simulated control system designed using the softwares Matlab and Simulink and then on the control of mobile Wheelphones that are innovative robotic platforms for smartphones.

Contents

List of Figures

1 Introduction

Wheelphone is an innovative mobile platform for smartphones built in 2013 by GC-Tronics company. It allows mobile phones to move in the surrounding area thanks to the presence of two wheels. Moreover the phone itself is the hardware that generates the movement of the platform.
In this thesis we want to improve the capabilities of this device implementing on it two area partitionig algorithms. Their task is to divide the enviroment in regions in order to position the robots in the most effective way to cover the area.

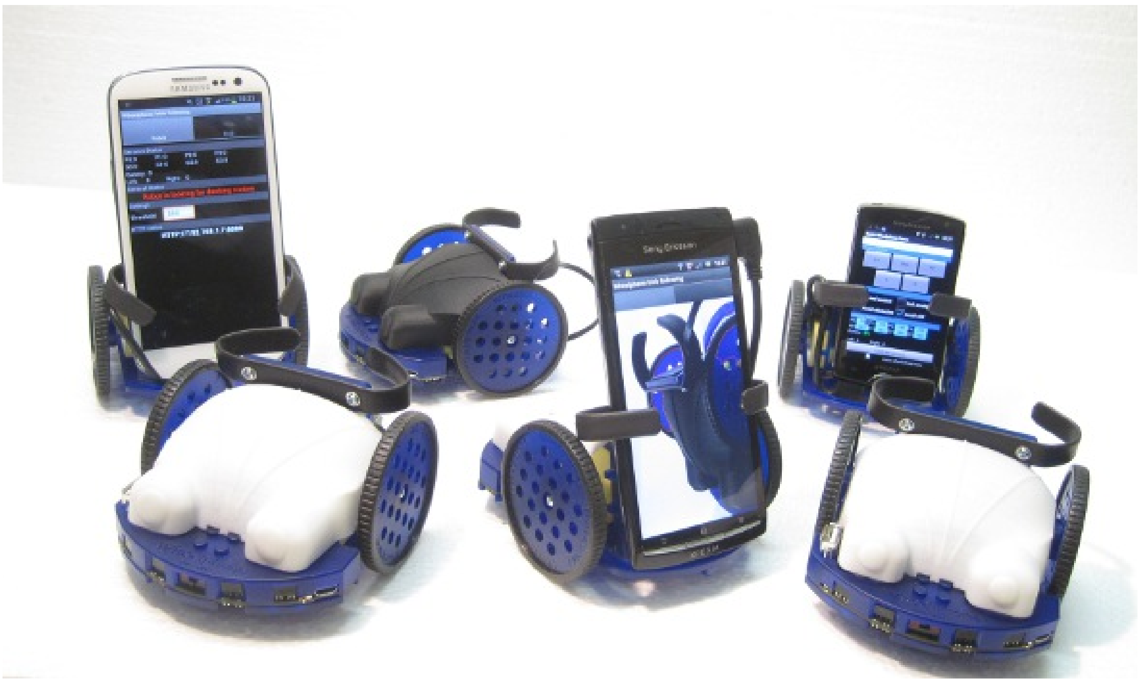


Figure 1: Wheelphone Robots

1.1 Motivation

Nowdays autonomous robots perform a broad range of tasks. Robotic camera networks can monitor airports and other public infrastructures. Teams of vehicles can perform surveillance, exploration, search and rescue operations. Groups of robots can have logistic capacities in the transportation of goods and in the delivery of services. There are many reasons for their use:

- **Quality**: for certain tasks that require high positioning precision and high repeatability, robots can be better than humans in terms of work quality.
- **Costs**: the high level of produttivity, from one hand, and the reduction in the number of wages, from the other hand, make robot more profitable.
- **Safety**: the application of robots is safer in certain situations such as working with dangerous materials or in extreme enviroment.

- **Flexibility**: a single robot can be used to perform multiple activities reducing time and improving quality.

It is interesting to analyze the robot performance in the coverage area application because of its importantance in everyday life and its difficulty in implementing it.
The coverage area topic consists in finding the optimal robot position in order to oversee a specific area. More robots are used, the more difficult it becomes to get good results. In fact the choosen zone has to be divided into N partitions each of which is associated whith a particular robot. Each robot has to monitor only the assigned sub-area. Moreover, assuming that the initial partitions and robots position into the area are not efficient, the comunication between the vehicles is needed. For this reason it is important to take into account the physical limitations of the robots: the range of communication signal and the computing power.
For example, considering a high communication range and a low computing power, it is better to use a server-based communication architecture; otherwise, with inverted values, it would be better to use a distributed architecture. In the first case the robots interact with an external server that has the task of performing calculations and redirecting them. In the second one the robots, communicating indipendently between them, decide their best location.
Another interesting challenge in analysing this topic is that any single area point can not have the same importance. To better explain this concept we consider the monitoring of a forest by a group of robots for detecting possible wildfires. It is easy to see that there is a direct correlation between the level of temperature and the possibility of fires, so the most important area points are the warmest. Assuming that, the robots should sorround the areas with the higher level of temperature leaving uncovered the other zones.
All these considerations are on the basis of the development of this thesis whose goal has been the implementation and the experimental validation of distributed partitioning algorithms using Wheelphone robots.

1.2 State-of-the-art

The last few years have seen a fast progression in the eld of the robotic control and new developments continue to expand the literature which presents continuously new solutions and approaches along with the arising of always new and harder challenges.
In the classical coverage literature, there are many works [1]-[4]-[5]-[6]-[10] that present a gradient descent strategy for a class of functions which encode optimal coverage policies. The authors exploit the concept of centroidal Voronoi partitions to optimally divide the monitored area focusing on different aspects. In [1] and [10] they consider a non-convex enviroment with the presence of obstacles whereas in [4]-[5]-[6] the coordination problems for networked robots are presented. In [12] the authors propose a policy for the optimal coverage of a line with perfectly known non-uniform sensory function. In [7] only a limited number of noise-free samples of the sensory function are considered. Finally, a distributed solution to the coverage problem in the presence of known time-varying density functions is presented in [11].
Another research direction can be seen in [21]-[23]-[22] considering the sensory func-

tion not known by the robots. In these cases each robot independently estimates the function of interest based on its own measurements and those gathered by its neighbors.
We have also to consider what tasks Wheelphone robot can alredy perform. The following list shows the name and the feratures of the main Applications that can be downloaded from the web.

- **Wheelphone** : visualize all the sensor information on the phone and implement "move-around-on-table" behavior
- **Wheelphone_follow** : it allows the robot to follow an object in front of it, using the front facing proximity sensors
- **WheelphoneRecorder** : record video while moving
- **WheelphoneFaceme** : face-tracking application. It keeps track of the position of one face using the front facing camera, then controls the robot to try to face always the tracked face
- **WheelphoneBlobDetection** : the Wheelphone robot follows a blob chosen by the user through the interface
- **WheelphoneLineFollowing** : it allows the robot to follow a black (or white) line on the floor using the ground sensors.
- **WheelphoneNavigator** : environment navigation application. It allows the robot to navigate an environment looking for targets while avoiding obstacles. It has two modules to avoid the obstacles: (1) the robot's front proximity sensors and (2) the camera + Optical Flow.

1.3 Original contribution

The goal of this thesis is to describe the implementation process and comparison of two partitioning algorithms given Wheelphone robots.
The starting point was to analyze the tecnical features of a single Wheelphone robot because, as is possible to see in the previous section, was not created for this type of task. We verified that we had the possibility to access the wheels speed and to set these data as we please. Once we have finished the analysis, we were able to implement a motion control algorithm to drive Wheelphone in a specific position. These were the basis for the realization of our algorithms.
The next step was to select two algorithms with different characteristics : once based on a synchronous client-server communication architecture and the other based on an asynchronous and pairwise communication.
Once chosen them, we identified a cost function, named *Multicenter Function*, that gave us the goodness of an algorithm in terms of proximity to the optimal robots position.
Going more in deep, the real original contribution of this thesis was that all the above description was implemented for a smartphone application. Nowdays everybody have smartphones which are an increasingly important part of day-to-day life. Many

tasks like home automation control, vehicle security, human body anatomy and health maintenance have already been designed in the form of App which can be easily installed. Making an App partition control can led, therefore, to significant advantages:

- **Cost reduction** : the control hardware component is almost entirely provided sice it is the phone itself. That means there is no need to buy an external hardware.
- **Easier programming** : the code used by Android phones is Java, that is one of most common but not user-friendly. Indeed, if we want to change some part or parameter of the algorithm, we have to know in detail all the code. On the opposite, traditional controlling methods usually involve technical expertise and complicated software. For these reasons we decided to interface the smartphone with Simulink MATLAB software. In this way we devided the algorithm in indipendent Simulink-Blocks making easier the reprogramming.
- **More accessible** : it is easier for people to get in touch with these type of tecnology. Therefore they can develop by their own a partition App that fits as best as it can people needs.

Moreover, in this thesis we build a robot simulator in MATLAB code that allow to test the algorithms and to prevent programming errors.

1.4 Thesis overview

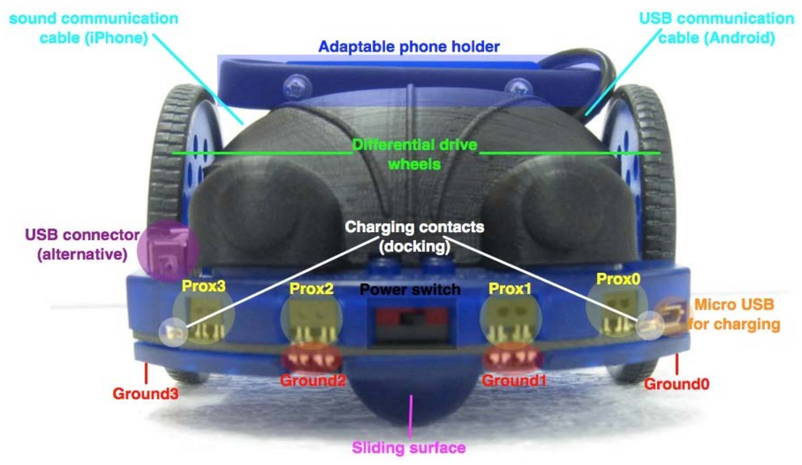
In this section is performed a short overview of the content of the thesis.
After the general introduction that has been presented in the previous paragraphs, we will illustrate in **Section 2** the experimental apparatus that includes both the hardware and the software components used first in the simulations and then in the experimental tests. In addition we will explain the process to interface the Wheelphone robot with Simulink MATLAB.
In **Section 3** the coverage issue will be defined in detail presenting, in particular, the approaches that we had selected. We will describe some geometric concepts like *Voronoid partition*, *centroid* and *Multicenter function* used for the enviroment patitioning. Furthermore in this section we will present the Server-Based algorithm and the Gossip Distributed one illustrating their foundamental characteristics.
The **Section 4** will introduce the mathematical model for an unicycle robot depicting its dynamic and kinematic. Thanks to them we will design a Motion Controller in order to move the vehicles to a define position in the area.
In **Section 5** we will focus on analysing the problems observed during the tests. We will solve these issues implementing the *Marker Labelling* and the *Pose Reconstraction* algorithms essential to estimate the positions and orientations of the robots. We will describe the implementation process of the partition algorithms in MATLAB reporting then the final results.
Finally **Section 6** summarizes the work done and the results obtained in this thesis along with an overview of the possible developments that could be explored in the near future.

2 Experimental Apparatus

This section presents the experimental apparatus used first in the simulations and then in the experimental tests.
All the simulations are run in MATLAB R2015b on a laptop with a processor Intel Core i3 and 4Gb of RAM. The tests, instead, are focus on the control of a robotic platform for smarphones, Wheelphone, whose hardware and software components will be respectively illustrated in Sections 2.1 and 2.2. We instantiated the partitioning algorithms in the smartphone Samsung GALAXY S3 mini through the MATLAB Support Package for Android Sensors. Finally, to perform a better motion control, we used the Motion Capture System of Padova's Engineering Department whose features will be described in Section 2.3.

2.1 Wheelphone Hardware

Wheelphone is a vehicle with two parallel wheels, each one mounted beside their center. It is able to steer thanks to a sliding surface placed on the front side of the robot. The Figure 2 shows the components of a Wheelphone and the Table below describes its features.



sound communication cable (iPhone)
Adaptable phone holder
USB communication cable (Android)
Differential drive wheels
USB connector (alternative)
Charging contacts (docking)
Prox3
Prox2
Power switch
Prox1
Prox0
Micro USB for charging
Ground3
Ground2
Ground1
Ground0
Sliding surface

Figure 2: Wheelphone Components

Feature	Tecnical Information
width	92 mm
length	102 mm
height	66 mm
weight	200 g
distance between wheels	92 mm
wheel diameter	68 mm
battery	LiPo rechargeable battery (1660 mAh, 3.7 V)
processor	microchip PIC24FJ64GB004; 16 MHz
memory	RAM: 8 KB; Flash: 64 KB

Wheelphone vehicle has two direct current (DC) geared motors, one for wheel, that can drive the robot with a maximum speed of 30 cm/s. There are not motor encoders and then the angular position of the wheels is estimated using the counter-electromotive force. There are also 16 sensors:

- 4 front and 4 ground ambient sensors, that can measure the ambient light,
- 4 front and 4 ground infra-red sensors measuring the proximity of objects up to 6 cm.

Wheelphone has a molded plastic case and an adaptable phone holder.

2.2 Wheelphone Software

GCtronic company provides users with the Wheelphone class, written in Java code, that need to be instantiated in the application in order to communicate with the robot. The robot-phone communication is established through a micro USB cable and the exchange of packets occurs every 50 ms. There are two different types of packets: the packet transmited to the vehicle (*Sending-Packet*) and that one received (*Receiving-Packet*).

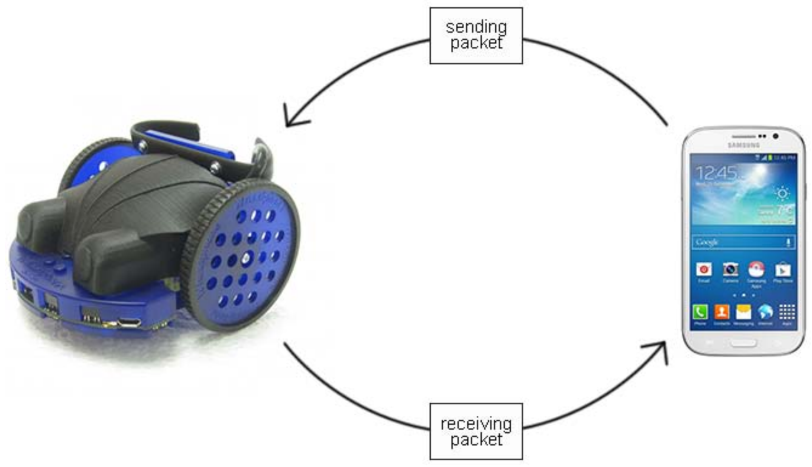


Figure 3: Phone-Robot Communication

The packets length is 63 bytes where only the first 4 bytes are used in the Sendind-Packet and only the first 23 bytes in the Receiving-Packet:

Byte	Sending Packet
1	Update State
2	Left Desired Speed
3	Right Desired Speed
4	FlagPhoneToRobot

Byte	Receiving Packet
1	Update State
2	Prox0 I-R Value
3	Prox1 I-R Value
4	Prox2 I-R Value
5	Prox3 I-R Value
6	Prox0 Ambient Value
7	Prox1 Ambient Value
8	Prox2 Ambient Value
9	Prox3 Ambient Value
10	Ground0 I-R Value
11	Ground1 I-R Value
12	Ground2 I-R Value
13	Ground3 I-R Value
14	Ground0 Ambient Value
15	Ground1 Ambient Value
16	Ground2 Ambient Value
17	Ground3 Ambient Value
18	Battery State
19	FlagRobotToPhone
20,21	Left Measured Speed
22,23	Right Measured Speed

The bits of *FlagRobotToPhone* and *FlagPhoneToRobot* are used to comunicate some specific information.

FlagRobotToPhone:

1 bit : speed control enable/disable

2 bit : soft acceleration enable/disable

3 bit : obstacle avoidance enable/disable

4 bit : cliff avoidance enable/disable

5 bit : calibrate sensors

6 bit : calibrate odometry

7,8 bit : not used

FlagPhoneToRobot:

5 bit : robot is charging

6 bit : robot is completely charged

other bits : not used

In order to implement the partitioning algorithms, we could either program in Java code and directly create an application that recalls the Wheelphone class or we could build the control scheme in MATLAB Simulink importing it in the embedded system through rapid prototyping. We decided to follow the second strategy as it brings many benefits:

- high-level programming and
- a faster addition or change of control components

To do this, the connection between Simulink and the low-level communication, specified in Wheelphone.java, has to be established through the use of an **S-function**. S-functions (system-functions) provide a powerful mechanism for extending the capabilities of Simulink adding your own blocks to Simulink models. S-functions use a special calling syntax that enables you to interact with Simulink's equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks. The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems.

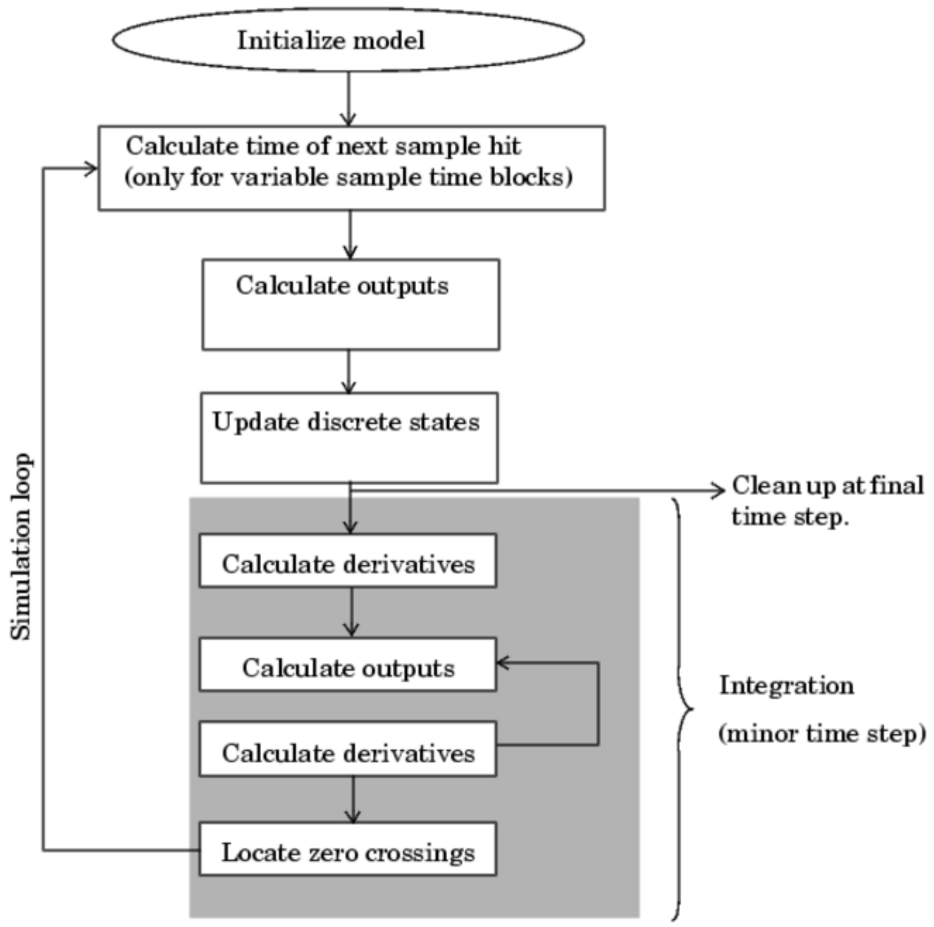


Figure 4: Simulink Block Process

In order to build our S-function, it is necessary understand which steps a Simulink model is ran (Fig. 4). In the first phase, initialization, Simulink incorporates library blocks into the model, propagates widths, data types, and sample times, evaluates block parameters, determines block execution order, and allocates memory. Then Simulink enters a simulation loop, where each pass through the loop is referred to as a simulation step. During each simulation step, Simulink executes each of the model's blocks in the order determined during initialization. For each block, Simulink invokes functions that compute the block's states, derivatives, and outputs for the current sample time. This continues until the simulation is complete.
For our tasks we built an s-function, **sfun_wheelphone**, written in C code, initializing as input and as output the values shown in Figure 5.

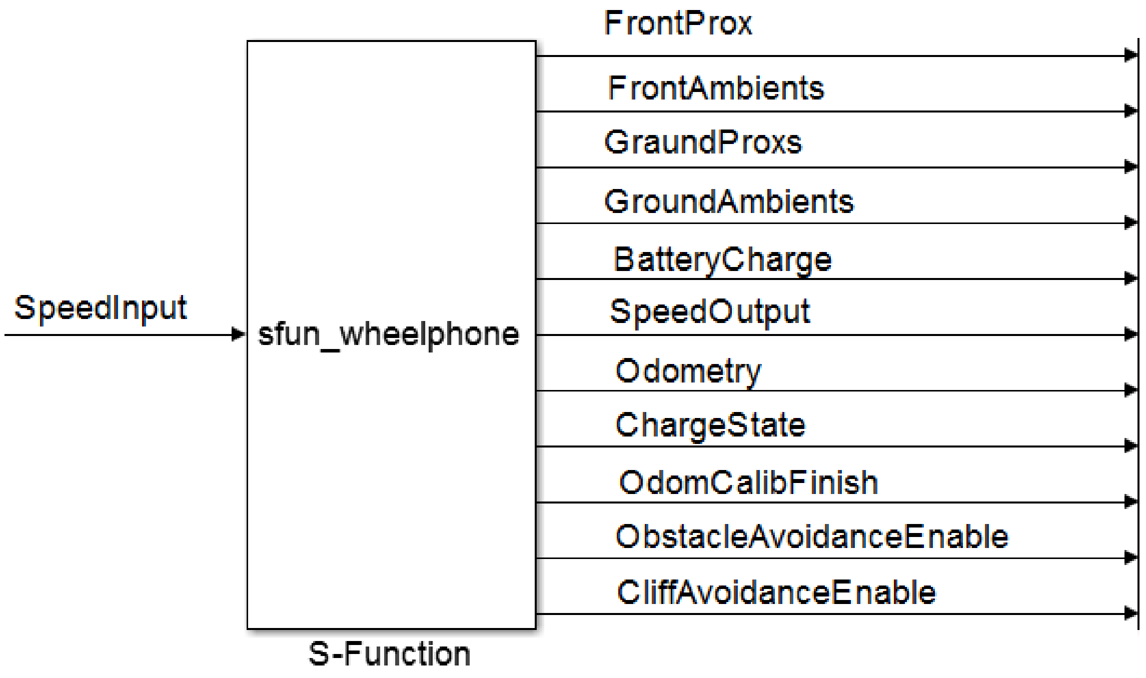


Figure 5: sfun_wheelphone

Now we have to istruite the Simulink Coder how generate the code linked to this block. We used an integral component of Real-Time Workshop, called Target Language Compiler (TLC), that transforms an intermediate form of a Simulink block diagram into code based on target file.
For this task we created a wrapped inlied S-function, *sfun_wheelphone.tlc*, and a file, *driver_wheelphone.c* . During each simulation step, *sfun_wheelphone.tlc* istruites the Simulink Coder specifying that the otuputs of *sfun_wheelphone.c* have to be taken using the methods of the file *driver_wheelphone.c*.

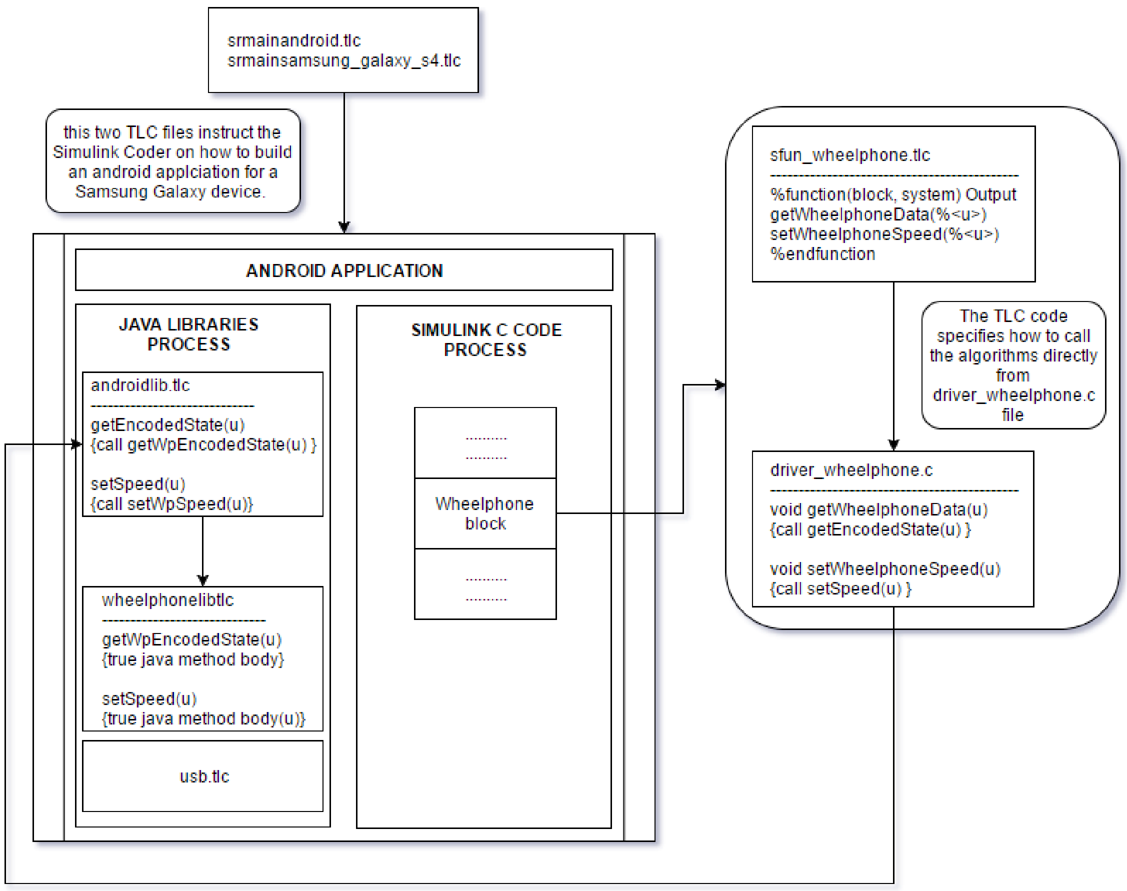
In *driver_wheelphone.c* file there are two methods that recall through *Androidlib.tlc* the pre-existing methods of the *Wheelphone* library:

- **getWheelphoneData()**: it acquires the *Receiving Packet* values
- **setWheelphoneSpeed(***leftWheel,rightWheel***)**: it sets the left and right wheel speed equal to *leftWheel* and *rightWheel*.

The code of *sfun_wheelphone.c*, *sfun_wheelphone.tlc* and *driver_wheelphone.c* are exposed in detail in Appendix A.
We use the MATLAB Support Package for android to download the application in our device creating inside it two foundamental process:

- **Simulink Process**: this is the process where our Simulink scheme runs.
- **Java Process**: it converts the Simulink scheme into Android code, using *srmainandroid.tlc* verifying that the code is compatible with our device through *srmainsamsung_galaxy_s4.tlc*. Furthermore, when Simulink coder enters in *sfun_wheelphone.c*, it activates through the process described above the methods of *Wheelphone.tlc* allowing the phone-robot communication.

The following diagram describe the life cycle of our Application.



srmainandroid.tlc
srmainsamsung_galaxy_s4.tlc
this two TLC files instruct the Simulink Coder on how to build an android applciation for a Samsung Galaxy device.
ANDROID APPLICATION
JAVA LIBRARIES PROCESS
androidlib.tlc
getEncodedState(u) {call getWpEncodedState(u) }
setSpeed(u) {call setWpSpeed(u)}
wheelphonelibtlc
getWpEncodedState(u) {true java method body}
setSpeed(u) {true java method body(u)}
usb.tlc
SIMULINK C CODE PROCESS
....
Wheelphone block
....
sfun_wheelphone.tlc
%function(block, system) Output
getWheelphoneData(%<u>)
setWheelphoneSpeed(%<u>)
%endfunction
The TLC code specifies how to call the algorithms directly from driver_wheelphone.c file
driver_wheelphone.c
void getWheelphoneData(u) {call getEncodedState(u) }
void setWheelphoneSpeed(u) {call setSpeed(u) }

Figure 6: Android Application's Life Cycle

2.3 Motion Capture

Motion Capture is the process of recording the movement of objects in the three dimensional space. The interest for this topic spread among various fields of research, from sport and entertainment, to military applications and robotics. In this Section, we want to analyze a optical Motion Capture System. It utilises data captured from

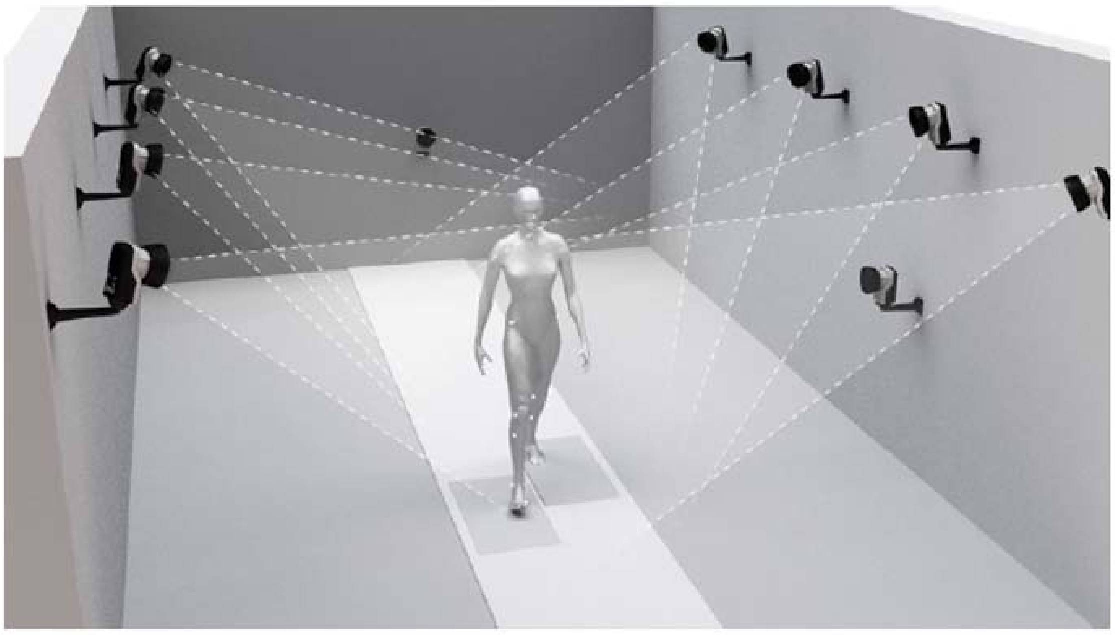


Figure 7: Optical Motion Capture System

image sensors to triangulate the 3D position and track a point of the subject in the space between two or more cameras calibrated to provide overlapping projections. The most common approach is still the so called *marker-based* approach. Special markers that can be easily detected through the image sensor are placed on the subject and the information about its movements can be retrieved once a proper description of the constraints that relate a marker to each other is deduced. The markers commonly used with these systems can be of two types:

- **Active Markers** are usually infrared LEDs placed on the subject. Rather then reflecting light back that is generated externally, the markers themselves are powered to emit their own light. These systems offer a simpler solution to the marker labelling problem because they allow to illuminate just one marker at a time but of course they are more expensive since each marker is an active electronic component. The advances in computational power and the elaboration of more sophisticated algorithms supported the widespread of passive motion capture systems.

- **Passive Markers** are coated with a retroreflective material to reflect the infrared light which is generated from infrared flash lights placed near the lens of each image sensor. An IR-pass filter is placed above each camera so that only the bright reflective markers are captured ignoring the rest of the image and simplifying the marker detection problem. The centroid of the marker is estimated as a position within the two-dimensional image that is captured.

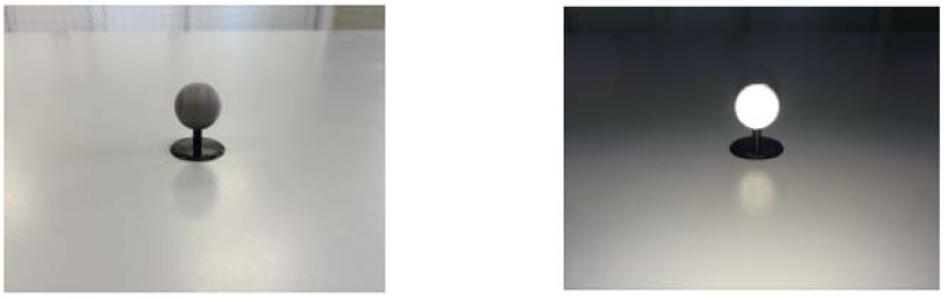

(a) Natural Light **(b)** Flash Light

Figure 8: Passive Markers

The laboratory, where we conducted the experimental tests, has a Motion Capture System with 12 infrared cameras capable of capturing images with a maximum frame rate of 360Hz provided by BTS Bioengineering [24].
The arrangement scheme of the cameras inside the room is represented in Figure 9 where the yellow area represents the actual usable volume of the laboratory.

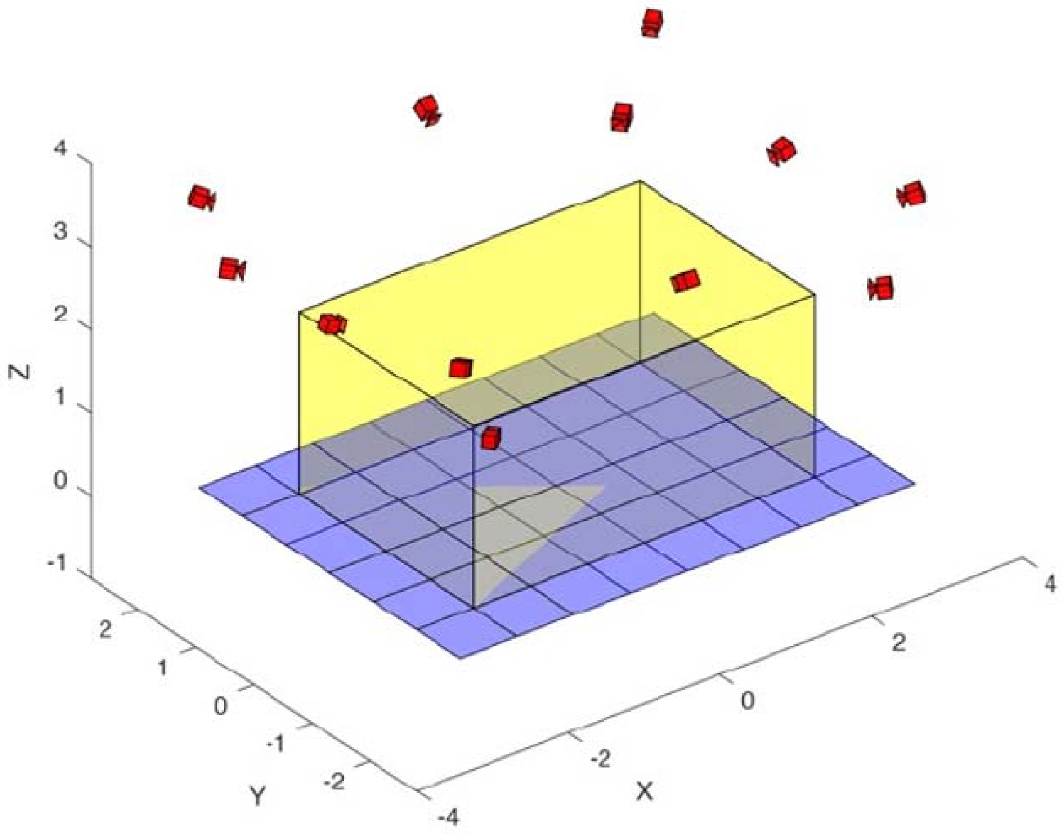
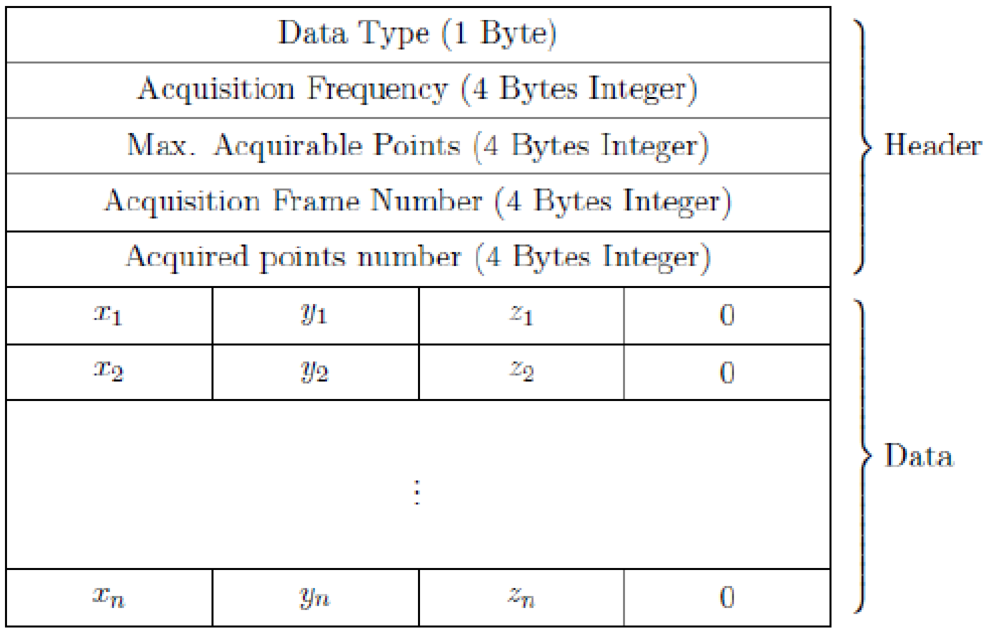


Figure 9: MAGIC Lab Configuration

The communication between the image sensors (cameras) and the central unit use a TCP/IP communication protocol:

1. each camera captures the two-dimensional position of the marker in its plane and transmits these data to the central unit through a TCP packet
2. the central unit receives the packet and, thanks to the triangulation algorithms, is able to derive the positions in 3D space of each marker
3. the marker positions are saved in 4 floating-point bytes and they are transmitted through a UDP packet with the following structure:



Data Type (1 Byte)			
Acquisition Frequency (4 Bytes Integer)			
Max. Acquirable Points (4 Bytes Integer)			
Acquisition Frame Number (4 Bytes Integer)			
Acquired points number (4 Bytes Integer)			
x_1	y_1	z_1	0
x_2	y_2	z_2	0
$\vdots$			
x_n	y_n	z_n	0

Figure 10: Packet Structure

3 The Coverage: Formulation and Algorithms

This section resumes a variety of known results in geometric optimization and in robotic coordination. Subsection 3.1 enunciates the notion of partitions and introduces the multicenter function as a way to define the optimal robots position in the environment. Subsections 3.3 and 3.4 describe two control algorithms for the agent motion coordination and for the environment partitioning based on the classic Lloyd method.

3.1 Voronoi Partitions, Centroids and Multicenter Function

Let $\mathbb{X}$ be a compact convex subset of $\mathbb{R}^2$ with non-empty interior. An **N-partition** of $\mathbb{X}$, denoted by $v = (v_i)_{i=1}^{N}$, is an ordered collection of N subsets of $\mathbb{X}$ with the following properties:

1. $\bigcup_{i \in \{i,...,N\}} v_i = \mathbb{X}$;
2. $int(v_i) \bigcap int(v_j)$ is empty for all $i, j \in \{i, ..., N\}$ with $i \neq j$; and
3. each set v_i, $i \in \{i, ..., N\}$ is closed and has non-empty interior.

Let $\mathbf{x} = (x_1, ..., x_N) \in \mathbb{X}^N$ denote the position of N agents in the environment $\mathbb{X}$. Given a group of N agents and an N-partition, each agent is one-to-one correspondence with a component of the partition.

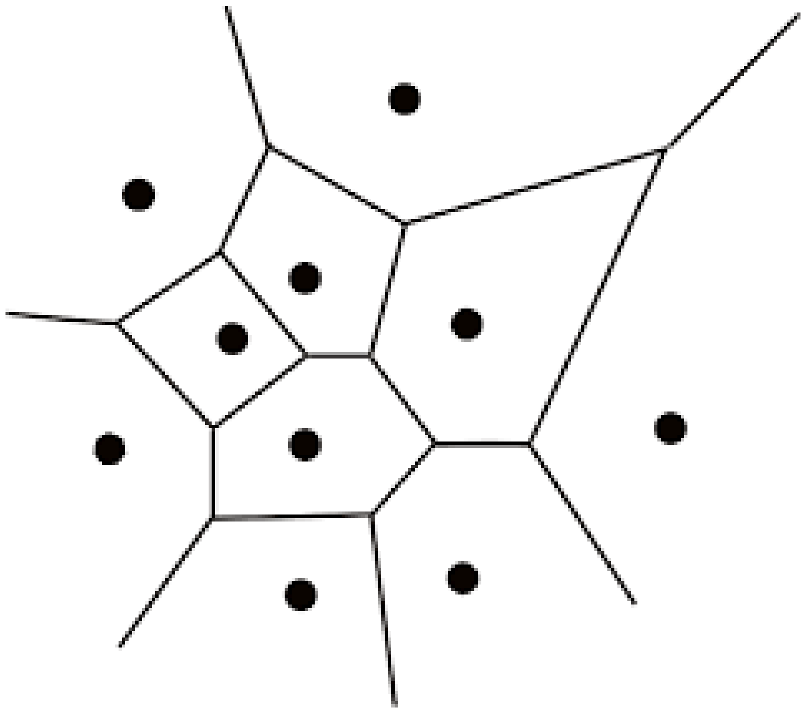


Figure 11: Voronoi Partition.

The **Voronoi partition** $W(\mathbf{x})$ of $\mathbb{X}$ generated by $\mathbf{x}$ is the ordered collection of the

Voronoi regions $(W_i(\mathbf{x}))_{i=1}^N$, defined by

$$W_i(\mathbf{x}) = \{q \in \mathbb{X} : \|q - x_i\| \leq \|q - x_j\|, \forall j \neq i\}\,. \tag{3.1}$$

Now, given two distinct points x_i and x_j in $\mathbb{X}$, define the $(x_i; x_j)$-*bisector half-space* by

$$H_{bisector}(x_i; x_j) = \{q \in \mathbb{X} : \|q - x_i\| \leq \|q - x_j\|\}\,. \tag{3.2}$$

The set $H_{bisector}(x_i; x_j)$ is the closed half-space containing x_i whose boundary is the plane bisected by the segment from x_i to x_j. Note that bisector subspaces satisfy $H_{bisector}(x_i; x_j) \neq H_{bisector}(x_j; x_i)$ and the Voronoi partition of $\mathbb{X}$ satisfies $W(\mathbf{x}) = \mathbb{X} \cap (\cap_{j \neq i} H_{bisector}(x_i; x_j))$.
Let $\mu : \mathbb{X} \to \mathbb{R}_{>0}$ be a distribution sensory function defined over $\mathbb{X}$. Given a generic partition v, for each region $v_i, i \in \{v_i, ..., v_N\}$, its **centroid** with respect to μ can be express as

$$C_i(v_i) = \left(\int_{v_i} \mu(q) dq \right)^{-1} \int_{v_i} q\mu(q) dq. \tag{3.3}$$

A partition $v = (v_1, ..., v_N)$ is said to be **Centroidal Voronoi partition** of the pair $(\mathbb{X}, \mu)$ if

$$v = W(\mathbf{C}(v)), \tag{3.4}$$

i.e., v coincides with Voronoi partition generated by $\mathbf{C}(v)$. At the end, with these notations, we want to introduce the **Multicer function** $H(v, \mathbf{x}, \mu)$ defined as

$$H(v, \mathbf{x}, \mu) = \sum_{i=1}^{N} \int_{v_i} \|q - x_i\|^2 \mu(q) dq. \tag{3.5}$$

It can be shown in [5] that, for a fixed sensory function μ, the set of local minima of $H(\cdot, \mathbf{C}(\cdot), \mu)$ coincides with the Centroidal Voronoi partitions of the pair $(\mathbb{X}, \mu)$.

3.2 Problem Formulation and Selected Approachs

This thesis wants to provide a correct partition of an area given a sensitive map and a group of N robots. The goodness of the algorithms is given by Multicenter Function that decribed how closest are the robots to the optimal locations. We decided to take some restrictions to implement the partitioning algorithms. First of all the robots can move only in a planar area represented by the convex set X. Additionally, each agent $i \in \{1, ..., N\}$ is assumed to have some computation and sensing capabilities:

- it always knows its locations $x_i(t)$ at time t
- it can send information either to a central server-base station or to the closely located agents

- it can move from its position $x_{i,k}$ to any desired location $x_{i,k+1}$ of convex set X

As seen in Section 1.2, there are many type of partitioning methods that have unique features. In order to analyse which is the best typology for Wheelphone robots, we decided to implement two algorithms that are as different from each other as possible. The first algorithm is based on a synchronous client-server communication architecture, instead the second one is a distribuited gossip algorithm that requires asynchronous and pairwise communication. The next Sections describes in detail the features of the implemented methods.

3.3 Server-based Algorithm

The first algorithm considered is a version of the classic Lloyd algorithm based on *centering* and *partitioning* for the computation of Centroidal Voronoi partitions.

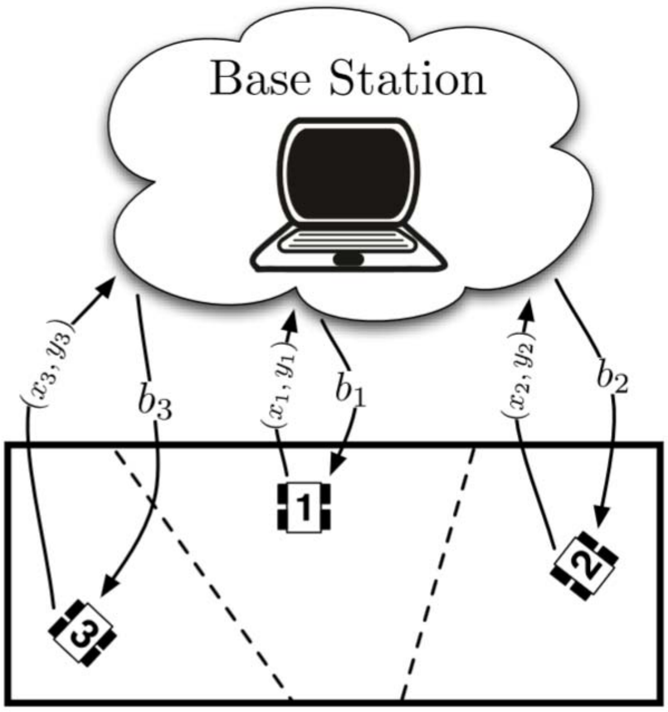


Figure 12: Client-Server Communication Architecture.

Let $\mathbb{X} \subset \mathbb{R}^2$ be a convex and closed polygon and let $\mu : \mathbb{X} \rightarrow \mathbb{R}$ be a sensory function. The coverage problem is equal to find the partition that minimize the Multicenter Function:

$$\min_v \{H(v, \mathbf{C}(v), \mu)\} . \tag{3.6}$$

The Lloyd's solution for this problem is to use an iterative algorithm that can be divided in four parts:

1. INITIALIZATION: it chooses an initial N-partition $v(0)$ of $\mathbb{X}$
2. CENTERING: it computes the centroids $\mathbf{C}(v(k))$ of the current partition k
3. PARTITIONING: it updates the new partition $v(k+1)$ using Voronoi on new centroids $v(k+1) = W(\mathbf{C}(v(k))$
4. TERMINATION: if the new partition and centroids meet some convergence criterium, like using the Multicenter function, it terminates; otherwise it returns to step 2.

It is easy to see that the final solution of this algorithm is not unique but it depends on the initial condition. Indeed to improve the convergence of the algorithm the best strategy is to choose wisely the initial partition $v(0)$ using the Monte Carlo method. It can be seen in detail in [9] that H is monotonically non-increasing along the solutions of Lloyd's algorithm and these solutions converge asymptotically to the set of Centroidal Voronoi partitions.
In this section we present a version of Lloyd's algorithm based on a synchronous client-server communication architecture (Figure 12). In this case the robots are allowed to communicate with a central server that:

- stores the map given by $\mu(\mathbb{X})$
- receives the position of all robots every T seconds
- computes centroids and Voronoi regions of all robots
- sends information periodically to robots every T seconds

Each robot i always knows its position and, when receives the new location, moves to this one. If the robot reaches the target point in time $t < T$, then it waits $T - t$ seconds to send the informations to the server; otherwise it stops in position $x_i(T)$ and immediately communicates with the base. This synchronous communication is necessary to implement the global Voronoi partitions and for this reason the setting of T value has an important role for the algorithm convergence time. Furthermore, in order to build a good algorithm, the packet dropouts and communications failures must be considered. The client-server architecture is naturally resilient to these problem. Indeed, in case that an input location is not received by the server, it only computes the partitioning with others locations. Similarly, if a robot does not receive the new target point, it just stands until the moment it receives the next control input.

Algorithm 1 Server-Based Algorithm

```
1: SERVER
   Require: The server stores in memory $\mu$ and has a clock that triggers an event every T seconds.
2: if $k = nT,\ n \in \mathbb{X}$ then
3:     Listen input locations
4:     $\mathbf{x}(k) = x_1(k), ..., x_N(k) \leftarrow ROBOTS$

5:     Partition and Centroids update:
6:     $v(k) = W(\mathbf{x})$
7:     $C_i = \left( \int_{v_i} \mu(q)dq \right)^{-1} \int_{v_i} q\mu(q)dq\ \forall i$

8:     Targets-Points trasmission:
9:     $x_i(k) = C_i\ \forall i$
10:    $x_i(k) \rightarrow ROBOT_i\ \forall i$
11: end if


12: ROBOTS
    Require: A clock with sample time T.
13: if $k = nT,\ n \in \mathbb{X}$ then
14:     if Robot is moving then
15:         STOP
16:     end if

17:     Location trasmission:
18:     $x_i(k) \rightarrow SERVER$

19:     Listen target-point:
20:     $x_i(k+1) \leftarrow SERVER$

21:     Move to the new target-point
22: end if
```

3.4 Distributed Gossip Algorithm

The coverage law, based upon the Lloyd algorithm and described in the previous section, has some important limitations:

- **Synchronized Communication**: every T seconds each robot communicates its position to the base station that calculates the new location,
- **Base Station Presence**: it must be positioned inside the range of comunication signal of each vehicle. It is easy to assume that this condition can not be verified in any situation, just think of the partitioning of a very large place.

For these reasons the aim of this chapter is to introduce an distributed coverage algorithm that reduces the communication requirements in terms of reliability, synchronization and topology.

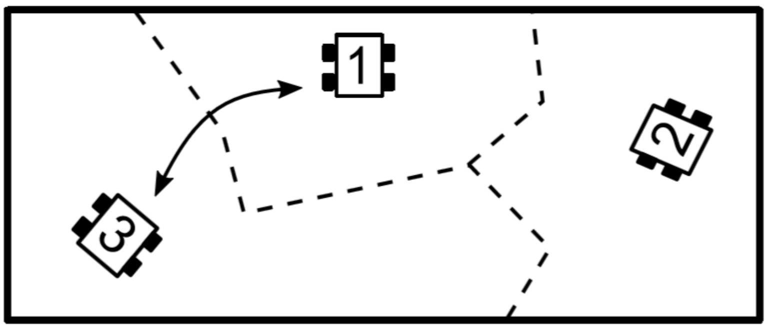


Figure 13: Peer-to-Peer Comunication.

The key idea is to implement the Lloyd strategy only on two adjacent regions. When two agents with distinct centroids communicate, their dominance regions evolve as follows:

1. the union **U** of the two dominance regions is created
2. **U** is divided into two new dominance regions using the *bisector half-space* on the two centroids

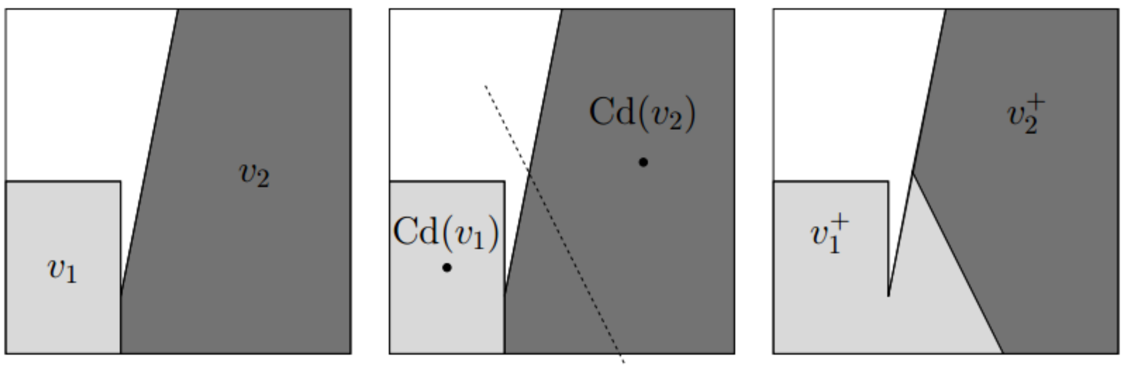


Figure 14: Partitioning Process of GD algorithm

This idea is well-posed in the sense that the sequence of collections $v(t)_{t \in \mathbb{N}}$ is an N-partition at all times t. Indeed it is immediate to see that the first two properties 3.1 are satisfied at all time if they are satisfied at initial time. Finally, at all times t, each component of v(t) is closed and has non-empty interior. Indeed it is impossible that exist a half-plane containing the interior of a region and not containing the centroid of the same region.
The gossip coverage algorithm takes advantage of what has been said in the following way. Let the collection $(v_1(0), ..., v_N(0))$ be an arbitrary poligonal N-partition of $\mathbb{X}$. For all $t \in \mathbb{N}$, each agent $i \in 1, ..., N$ mantains in memory the dominance region $v_i(t)$, the sensory function μ and its position $x_i(t)$. At each $t \in \mathbb{N}$ one or more pairs of distinct agents are selected by a random process. We define *pseudodist* between two closed region, v_a and v_b, with non-empty interior:

$$pseudodist(v_a, v_b) = inf\,\{|a - b| : (a, b) \in int(v_a) \times int(v_b)\} \tag{3.7}$$

Consider i and j the two agents of a pair, they perfom the following tasks:

1. agent i transmits to agent j its dominance region $v_i(t)$ and its position $x_i(t)$

2. agent j computes the $pseudodist(v_i, v_j)$

3. **if** $pseudodist(v_i, v_j) > 0$ **then**
   the dominance regions are not adjacent; the robots do not change their regions and positions:

   $x_{i,j}(t + 1) = x_{i,j}(t)$

   $v_{i,j}(t + 1) = v_{i,j}(t)$

4. **else**
   the dominance regions are adjacent; the robots region and position change in the following way:

   $v_i(t + 1) = (v_i(t) \bigcup v_j(t)) \bigcap H_{bisector}(x_i(t); x_j(t))$
   $v_j(t + 1) = (v_i(t) \bigcup v_j(t)) \bigcap H_{bisector}(x_j(t); x_i(t))$

   $x_i(t + 1) = C_i = \left(\int_{v_i(t+1)} \mu(q)dq\right)^{-1} \int_{v_i(t+1)} q\mu(q)dq$
   $x_j(t + 1) = C_j = \left(\int_{v_j(t+1)} \mu(q)dq\right)^{-1} \int_{v_j(t+1)} q\mu(q)dq$

5. agent j saves $\{v_j(t + 1), x_j(t + 1)\}$ and transmits $\{v_i(t + 1), x_i(t + 1)\}$ to agent i

6. agents i and j move to new positions $x_i(t + 1)$, $x_j(t + 1)$

In analysing this algorithm, we observed that there is a huge problem: it can generate non-convex partitions. As shown in Figure 14, that described an algorithm iteration, when the *bisector half-space* cuts the **U** region it can happen that the

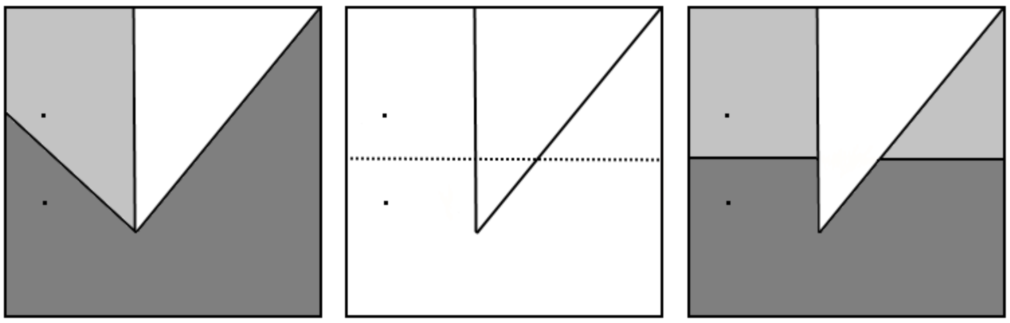


Figure 15: A Gossip Distributed Iteration

two new partitions become concave. This could implied, in the following iterations, the creation of partitions with disconnected regions (Fig. 15).
In this case there are two negative consequences:

- from the computational point of view, the algorithm performance become less efficient;
- from the area subdivision point of view, it is easy to see that this type of partition is not optimal. This is the reason why in almost any coverage problem is desirable to mantain the components connectivity.

We overcome this issue adding another constraint in process. When agent j verifies that the dominance regions are adjacent, it also checks that the two partitions, cut by the *bisector half-space*, are made up of connected components.

4 Wheelphone Robot: modeling and control

This section describes the process that must be considered to simulate, with *Simulink* software, the patitioning algorithms of Chapter 3.
First of all, we have to create the Wheelphone object in our simulator. Subsection 4.1 enunciates the kinematic and dynamic model of an unicycle robot type (that is the same type of Wheelphone). The second fase consists in designing the motion contol (Fig:16) that allows to drive the vehicle from one point to another.

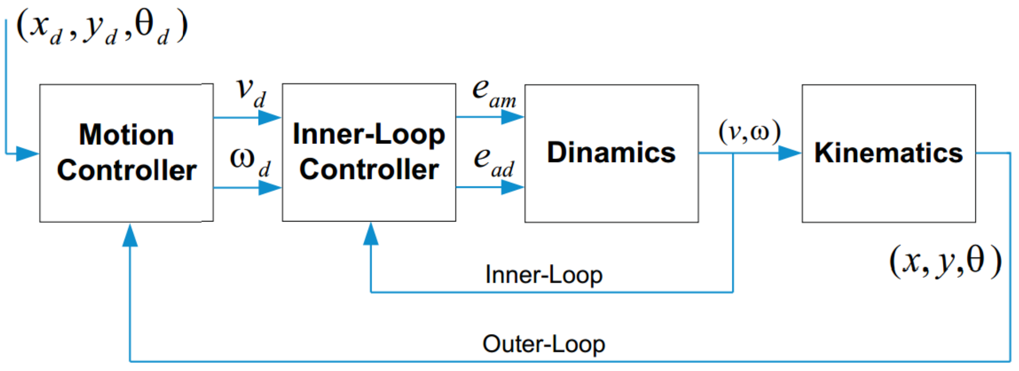


Figure 16: Motion Control.

4.1 Unicycle Model

An unicycle type robot is, in general, a robot moving in a 2D world, having some forward speed but zero instantaneous lateral motion. In other words, it is a non-holonomic system. Despite the unicycle name, it describes vehicles having usually two parallel driven wheels, each one mounted beside their center. The unicycle type robots modeling comprises their kinematics and dynamics study, as it is usual for most the physical systems.

4.1.1 Dynamic Modeling

The dynamic modeling concerned with the study of forces and torques and their effect on motion, defining the commanding speeds. The dynamic of the vehicle is

$$\begin{cases} M\frac{dv}{dt} &= -K_v v + K_m e_{am} - B_v v \\ J\frac{d\omega}{dt} &= -K_\omega \omega + K_d e_{ad} - B_\omega \omega \end{cases} \tag{4.1}$$

where v, ω are the linear and angular velocities of the vehicle, M and J are the mass and the inertia of the vehicle, respectively; e_{am} and e_{ad} are the average and differential voltages applied to the wheels. K_v, K_m, K_ω, K_d are constant which map forces, linear and angular velocities in forces and torques. Finally B_v and B_ω are the translational and rotational friction coefficients, respectively.

The voltages applied to the wheels, V_R e V_L, are connected to e_{am} e e_{ad} through

$$\begin{cases} e_{am} & = \frac{1}{2}(V_R + V_L) \\ e_{ad} & = V_R - V_L \end{cases} \tag{4.2}$$

In the end, the relation among the linear velocities of the wheels and the linear and angular velocities of the vehicle is

$$\begin{cases} v_R & = v + \omega d \\ v_L & = v - \omega d \end{cases} \tag{4.3}$$

The dynamic of the robot has been expressed through a state-space representation using as input the vector $[e_{am} \quad e_{ad}]^T$, as state $[v \quad \omega]$ and, as output, the vector of wheels velocities $[v_R \quad v_L]^T$

$$\begin{cases} \dot{x} & = Fx + Gu \\ y & = Hx \end{cases} \tag{4.4}$$

where

$$F = \begin{bmatrix} -\frac{K_v + B_v}{M} & 0 \\ 0 & -\frac{K_\omega + B_\omega}{J} \end{bmatrix}, \qquad G = \begin{bmatrix} \frac{K_m}{M} & 0 \\ 0 & \frac{K_d}{J} \end{bmatrix} \qquad e \qquad H = \begin{bmatrix} 1 & d \\ 1 & -d \end{bmatrix}$$

4.1.2 Kinematic Modeling

Kinematics modeling describes the trajectories that mobile robots follow when they are subject to commanding speeds. The kinematic of the vehicle is

$$\begin{cases} \dot{x} & = \frac{1}{2}(v_L + v_R)\cos(\theta) \\ \dot{y} & = \frac{1}{2}(v_L + v_R)\sin(\theta) \\ \dot{\theta} & = \frac{1}{2d}(v_R - v_L) \end{cases} \tag{4.5}$$

where x, y, θ is the pose of the vehicle, e.g. position and orientation in the plane, v_R and v_L are the linear speed provided by the right and left wheel, while d is the length of the semi-axis of the vehicle.
With the previous formula, it has been possible to estimate position changes over time. The diagram (Fig: 17) shows the odometry where the three integrators are used to convert velocities $\dot{x}, \dot{y}, \dot{\theta}$ in the vehicle pose x, y, θ.

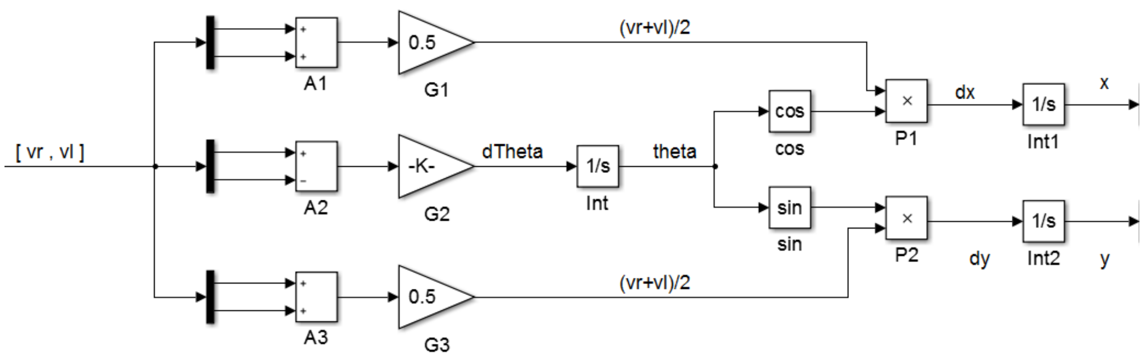


Figure 17: Odometry.

4.2 Motion Control

This section describes the motion control laws which allows to drive the vehicle from one point to another.
The proposed controller exhibits a inner-outer-loop structure (Fig:16). The inner-loop control law is responsible to compute the adequate electrical signals (voltage) that will tackle the wheels's motors to force the robot to move according to a desired linear and angular velocity. These desired velocities are the control signals generated by the outer-loop controller.

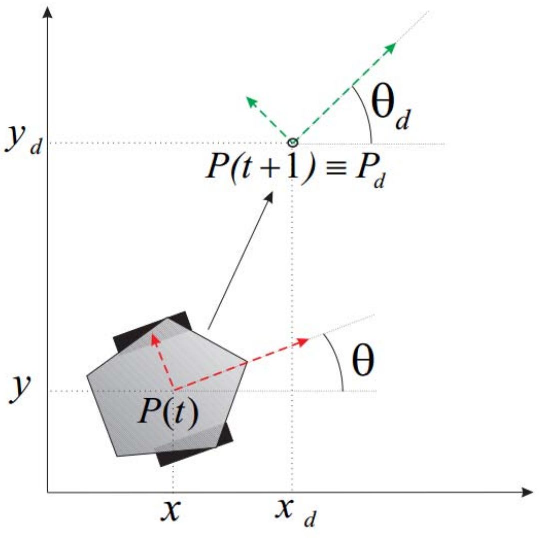


Figure 18: Tracking.

4.2.1 Inner-Loop

In order to drive robot to a desired linear velocity v and angular velocity ω, it is essential compute the error between the true velocities and the desired ones. Therefore, let e_v and e_ω be respectively the linear and angular velocity errors, we initially design a proportional control system

$$\begin{cases} e_{am} &= -K_{P_1} e_v \\ e_{ad} &= -K_{P_2} e_\omega \end{cases} \tag{4.6}$$

In this case, if the dynamic of the robot has small static gains, the velocities errors may remain significant. Then, adding an integral term, we can enforce the steady state error convergence to zero.

$$\begin{cases} e_{am} &= -K_{P_1} e_v - K_{I_1} \int_0^t e_v(\tau) d\tau \\ e_{ad} &= -K_{P_2} e_\omega - K_{I_2} \int_0^t e_\omega(\tau) d\tau \end{cases} \tag{4.7}$$

In Fig. 19 there is the diagram used to implement the Inner-Loop controller through Simulink software. The block *Sat*, placed before the state-space representation, has the objective to saturate the average and differential voltages and, consequently, to limitate the maximum speed of the wheels. Without any features regarding the behavior of the system in the transitional phase, we used an experimental calibration. The two PI controller parameters are shown in Table 1.

	Kp	Ki
PI 1	25	5
PI 2	35	5

Table 1: Experimental values for PI controllers.

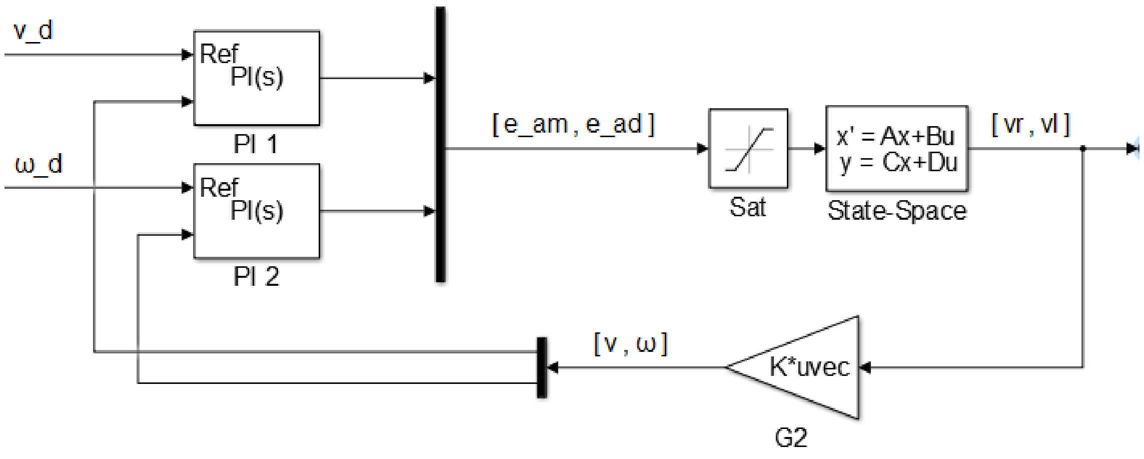


Figure 19: Inner-Loop Controller.

In the figures 20 and 21 there is the trend of v and ω speeds in response to a reference input step with amplitude 0,3m/s e 0,2rad/s respectively. The control manages to bring the speeds to the desired values without overshooting that would be harmful to the controls applied subsequently.

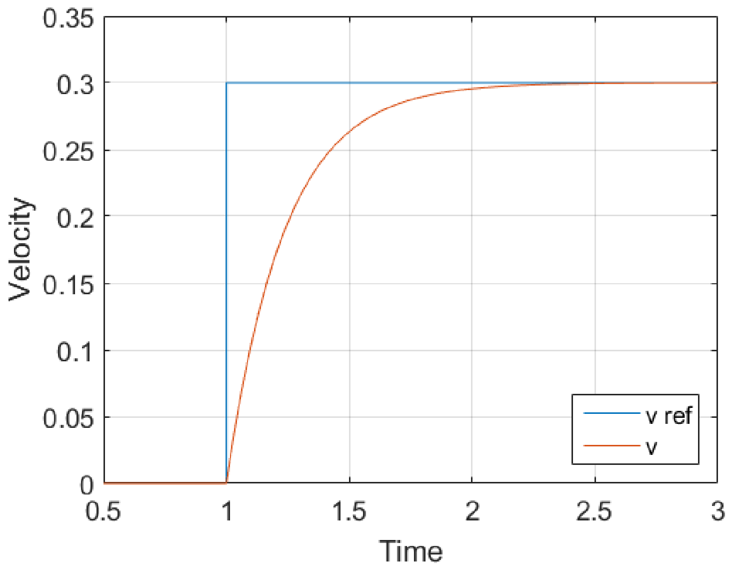


Figure 20: Linear Velocity Trend

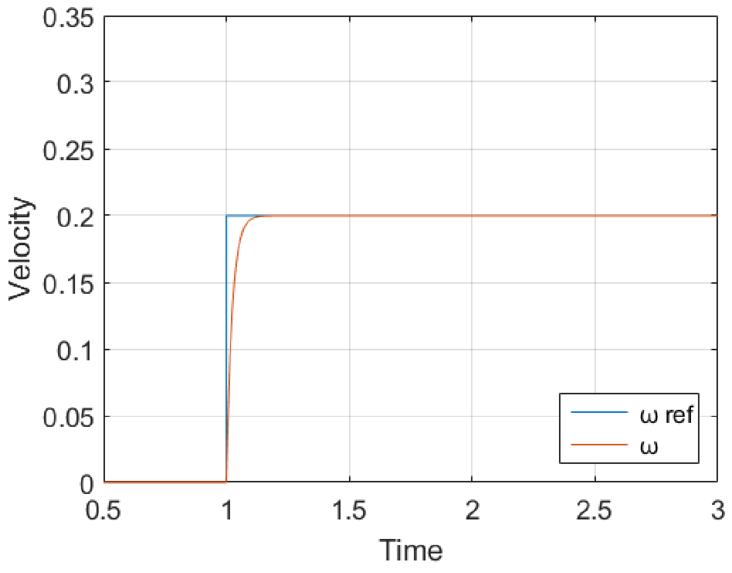


Figure 21: Angular Velocity trend

4.2.2 Outer-Loop

The Outer-Loop objective is to generate the adequate desired linear and angular velocity (v_d, ω_d) to force the position of the robot $P = (x,\ y)$ to converge to the reference position $P_r = (x_r,\ y_r)$. We decided to control, istant by istant, these speeds in proportion to the distance from the point of arrival and the missing angle to reach the desired orientation. Therefore let:

- $dist(t) = \sqrt{(x_r - x(t))^2 + (y_r - y(t))^2}$ be the Euclidean distance and
- $\theta_r(t) = atan2((y_r - y(t), (x_r - x(t))$ be the desired orientation,

the contro law is

$$\begin{aligned} v_d &= K_v dist(t) \\ \omega_d &= K_\omega(\theta_d(t) \ominus \theta(t)) \end{aligned} \tag{4.8}$$

where K_v and K_ω are the controller parameters and the operator $\ominus$ computes the difference between the angles expressed as a value between $-\pi$ and π.
For the same reason of paragraph 4.2.1, we used an experimental calibration to determine K_v and K_ω that are:

$$K_v = 4 \qquad K_\omega = 9.$$

The most important value that we have to define is K_ω. Indeed faster the robot finds the right direction less path it will take. The figure 22 exhibits the trajectory that the controller makes the robot do to reach the position (-0.5,0.5). As we can see, it starts from the origin with an initial orientation equal to the one of the x-axis untill it gets to the target.

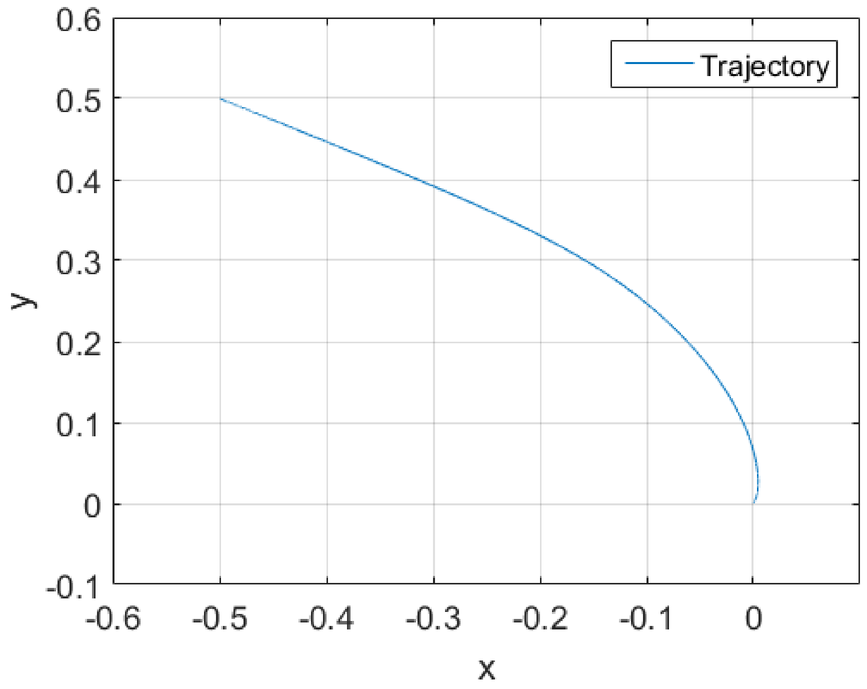


Figure 22: A Vehicle Trajectory

The diagram in Fig. 23 is the Outer-Loop implementation in Simulink software where the Inner-Loop and Odometry blocks are the same used in Fig. 19 and Fig. 17.

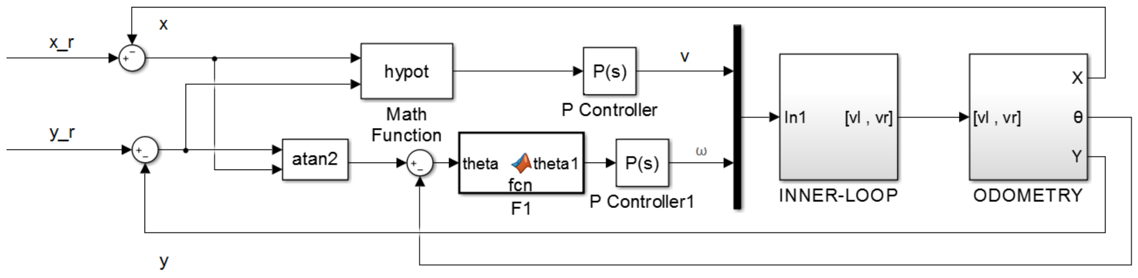


Figure 23: Outer-Loop.

The *hypot* and *atan*2 blocks are provided by *Simulink* performing the followyng math operation:

- $hypot(x, y) = \sqrt{x^2 + y^2}$
- $atan2(x, y) = arctan(\frac{x}{y})$

The F1 block, instead, is a function built by us, through MATLAB function block, to perfom the $\ominus$ operator. In the next lines it is shown the F1 code

Algorithm 2 F1 function

```
1: function theta1 = F1(theta)

2: while theta > $\pi$ do
3:     theta = theta$-2\pi$;
4: end

5: while theta < $-\pi$ do
6:     theta = theta$+2\pi$;
7: end

8: theta1 = theta;
```

The process until here described, composed by:

- unicycle dynamic
- unicycle kinematic
- inner-loop controller
- outer-loop controller,

has allowed us to create a robot simulator whereby we tested the partitioning algorithms in a simulated environment before implementing them on real devices. These tests have improved the debugging phase of the algorithms letting us be more efficient in the experimental stage.

5 Numerical and Experimental Results

This section presents in detail the steps performed in the MagicLab laboratory of Padova's engineering department to implement the partitioning algorithms on Wheelphone devices. It is organized in the following setting:

- we provide the features of the preliminary diagrams used to perform motion control and its results
- we introduce the algorithms created for the reconstruction of the vehicles poses having access to the data obtained from the Motion Capture System of Section 2.3
- we show how we implemented the Server-Based algorithm and the Distributed Gossip algorithm in MATLAB Simulink.
- finally, we show some real simulations of the two algorithms chosen in Section 3.

5.1 Preliminary Results

Before testing the validity of partitioning algorithms, we must make sure that the motion control, designed in a simulated environment, is also robust with real devices. Then we installed in our smartphone an Application that contained the Simulink diagram of Figure 24.
The *ODOMETRY* and the *MOTION CONTROLLER* blocks have the same structure as those presented respectively in sections 4.1.2 and 4.2.2 while we replaced the inner-loop block with *sfun_wheelphone* as it is no longer necessary to simulate the robot dynamics. Since the S-function inputs, v_L and v_R, i.e. left and right wheel

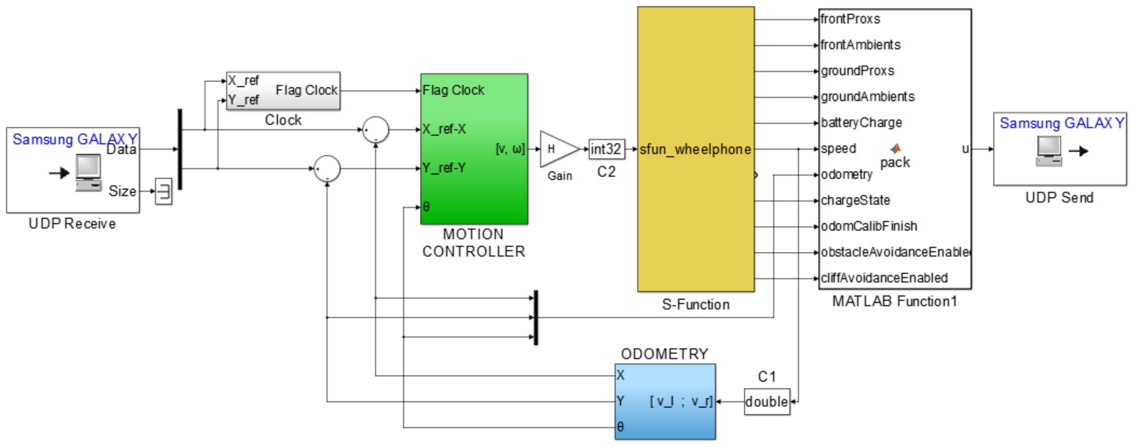


Figure 24: Application Simulink Scheme

speeds, are different from the Motion Controller outpus, v and ω, i.e. linear and angular robot speeds, we included in the control chain a matrix gain H to perform the conversion of the values:

$$H = \begin{bmatrix} 1 & 1 \\ d & -d \end{bmatrix}$$

where d is the length of the semi-axis of the vehicle.
The communication between the vehicles and the central base has been established through the *UDP Receive* and *UDP Send* blocks provided by MATLAB Support Package for Android Sensors. Even in this situation we defined two types of packet: the packet transmitted to the phone and the one received by it.

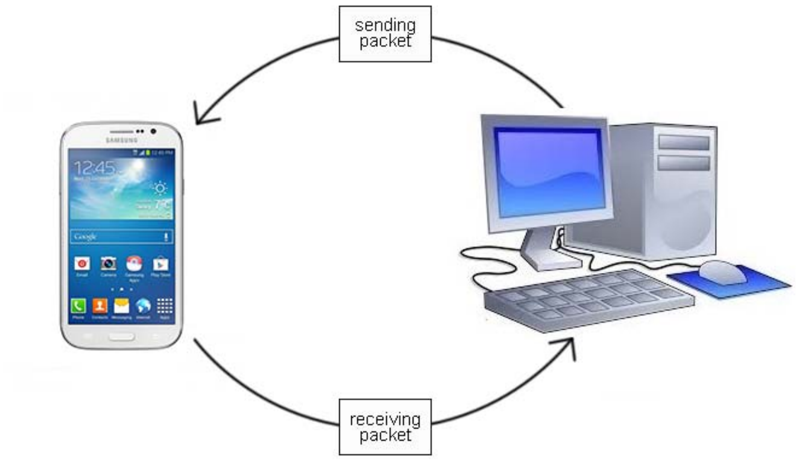


Figure 25: Phone-Computer Communication

The following tables show the structure of the packages that we used:

Byte	Sending Packet
1:8	Desired X
9:16	Desired Y

Byte	Receiving Packet
1:4	Proximity Values
5:8	Proximity Ambient Values
9:12	Ground Values
13:16	Ground Ambient Values
17	Battery State
18:21	Left and Right Wheel Speed
22:45	Odometry
46	Battery Charging State
47	Odom-Calibration State
48	Obstacle-Avoidance State
49	Cliff-Avoidance State

Finally, the *Clock* block of Fig.24 has the objective to make synchronous the movements of the vehicles. It activates a timer when a packet comes from the central base and, exceeding $T_{Clock} = 20$ seconds, it forces the linear and angular velocity equal to zero. We chose to make synchronous the movement of the robots and not their communication because creating independent clocks in the vehicles and in the base station could lead to asynchronous communications implyng control failure.

Applying the above scheme through a huge number of experimental tests, we calibrated the proportional constants, K_v and K_ω, of the motion controller, obtaining the following values:

$$K_v = 2 \qquad K_\omega = 5.$$

Nevertheless we faced that in same cases the vehicle did not reach the desired final position. In fact, placing the robot with a specific orientation and choosing as a final point a destination distant at least 2 meters and that is exactly in the opposite direction, the vehicle instead of steering mantains the initial orientation. This is caused by the presence of internal saturators in the two robot motors that limit the maximum wheels speed. In these situation, being our motion control proportional to the distance, the velocity of the left and the right wheels would exceed the saturators threshold and, not being possible, they are set equal to the threshold itself. To solve this problem, we primarly reduced the K_v value realizing that also the vehicle performance was lowered in the short-distance trips. For this reason we opted to insert a saturator in the Motion Controller with the scope to limit the linear velocity to the 80% of the threshold. In this way, even if the robot potentialities are under exploited, the motion control is working equally for all range of distance.

Moreover we got an additional test to verify the control strength. The vehicle in this case had to follow a sequence of points. Then, placing the robot in (0,0) position, we transmitted to it, in differt times, four packets that contained respectively the positions (0.5,0), (0.5,0.5), (0,0.5) and (0,0). As you can see by the Figure 26,

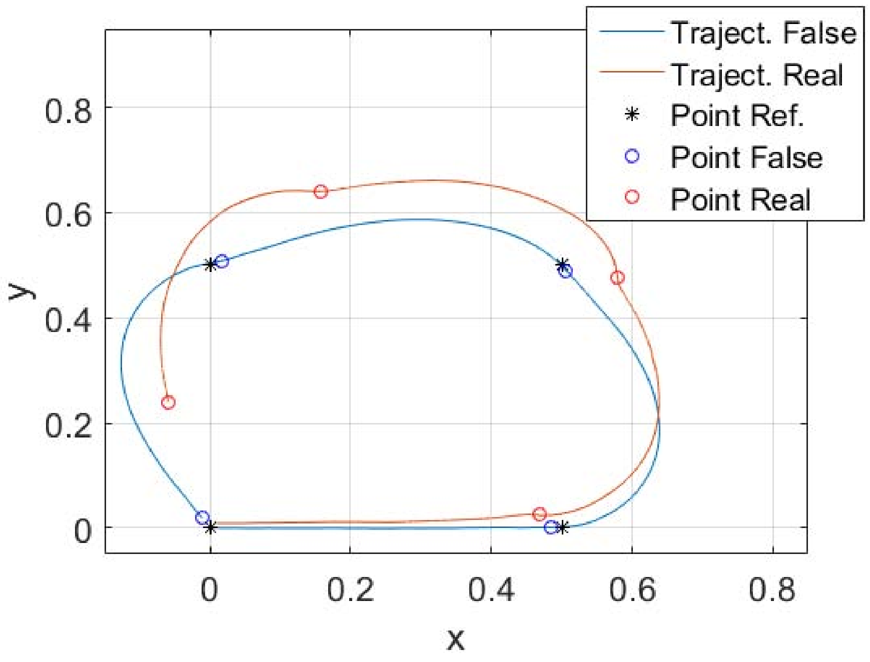


Figure 26: "Sequence of Points" Test1

the trajectory and the places where it stopped are different from those in which it thought to be. This anomaly could be caused by the lack of the encoders that measure the actual speed of the motors. In fact, using the counter-electromotive force, the engine speed in our device is only estimated. Being that an intrinsic problem, is not possible to solve it through a software reprogramming and for this reason we chose to adopt the Motion Capture System described in Section 2.3. In the following subsection is explain in more detail the use of the Motion Capture System in our application.

5.2 Pose Reconstruction with 3D Motion Capture System

In the section 2.3 we analized the modality which the Motion Capture System acquires the markers position. The purpose of this chapter, instead, is to obtain the Wheelphones pose, i.e. position and orientation in the three-dimensional space, from markers data.
We choose to link each robot to a pattern of 4 markers since that one single marker gives us only three degrees of freedom which represent its position but no information about the orientation of the reference frame attached to it. To do this, we removed the upper case installing a cardboard with the markers as you can see in Figure 27. Before explaining how the robots pose is rebuilt, it is necessary to understand how

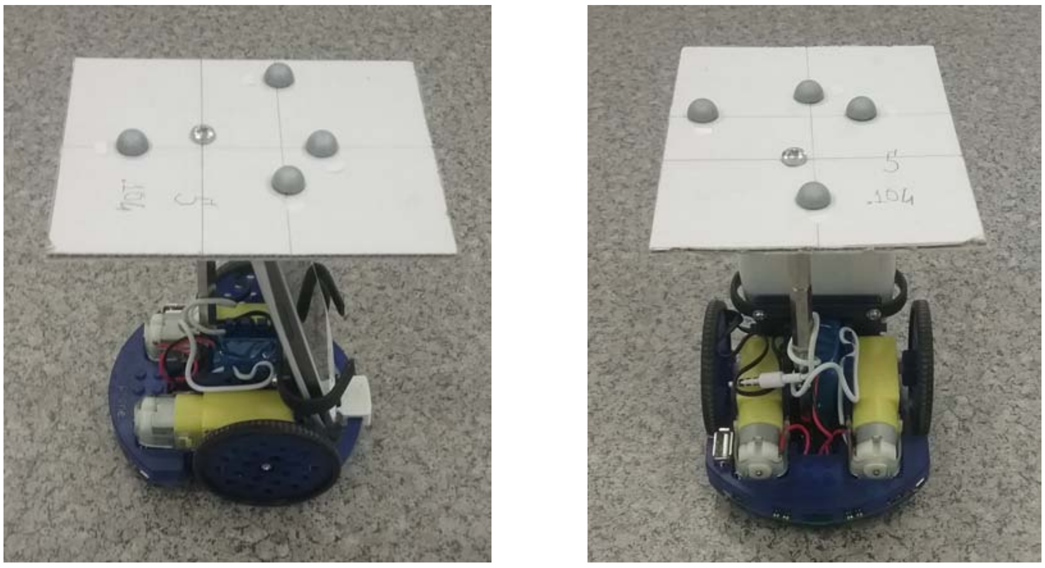


Figure 27: Wheelphone Robot with Markers

we make the markers labelling in order to connect each marker to the right pattern. The following subsection will describe the structure of the used algorithm.

5.2.1 Marker Labelling

Multiple markers are non distinguishable from a camera point of view, there are no distinctive parameters intrinsic to the marker such as colour or shape, they all look as bright elliptic blobs, whose position is later approximated by their centroid.

In this scenario it is clear that, when multiple markers are used, it is necessary to develop some strategies to be able to keep track of each marker individually in every set of frames. Our approach, to make marker labelling, takes advantage of the rigidity constraints which link every point of the solid object. We placed four markers on each Wheelphone robot in such a way that there were no equally spaced pairs. Furthermore to distinguish a vehicle from another, we choose different shape patterns for each robot (Figure 28).

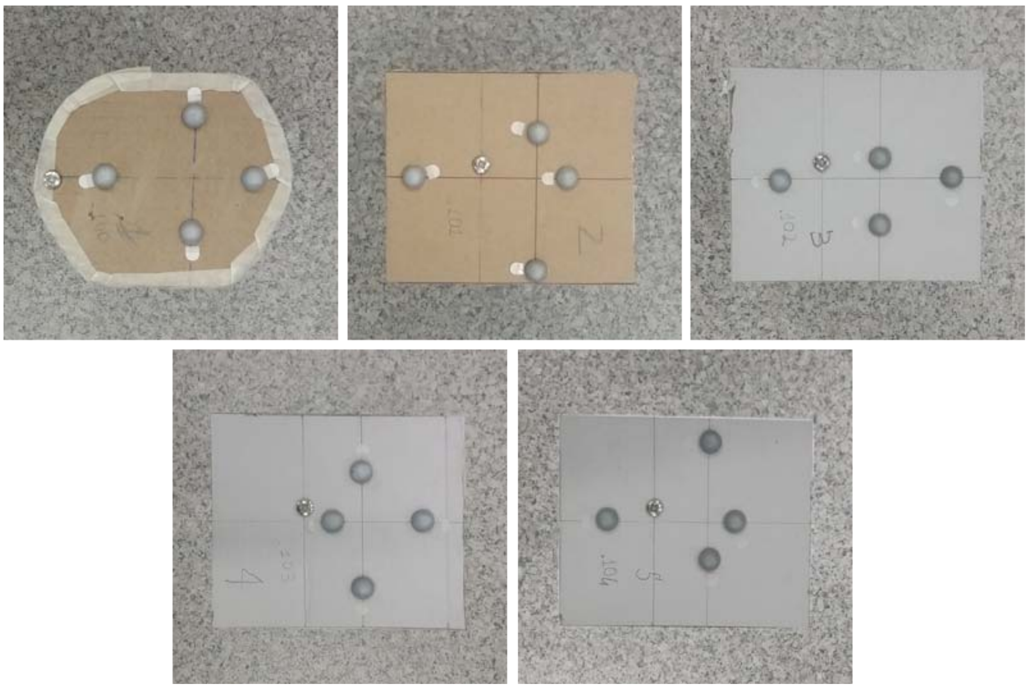


Figure 28: Robots Patterns

These configuration can be modelled with undirected connected graphs where the nodes represent each different marker and the weights of the edges represent the distance between each pair of nodes of its pattern.
Considering a specific pattern, the distances between each pair of markers are invariant for any rotation or translation of the robot due to the rigidity of the structure. With this reasoning every node is uniquely identified by the ordered triplet of the weights of the edges connecting its three adjacent nodes. To achieve our purpose we first need to create the reference models for the structures defined by the markers.

For any model we stores two matrices:

- $P_{model} \in \mathbb{R}^{3\times4}$ that contains the four markers absolute position at initial time

$$P_{model} = \begin{bmatrix} x_A & x_B & x_C & x_D \\ y_A & y_B & y_C & y_D \\ z_A & z_B & z_C & z_D \end{bmatrix} \tag{5.1}$$

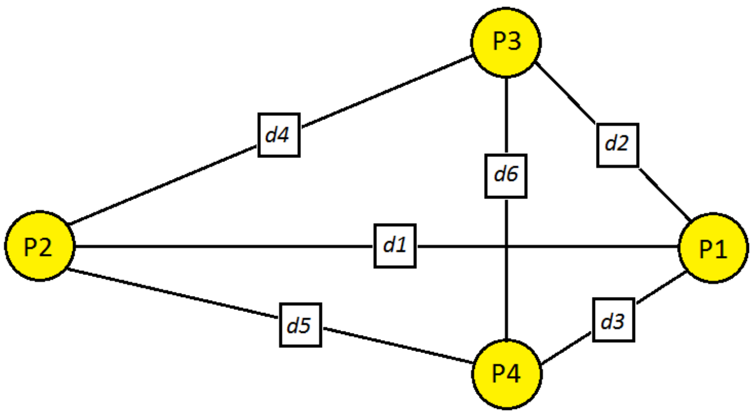


Figure 29: Reference Model

- $D_{model} \in \mathbb{R}^{4\times4}$ is the distance matrix where $d_{ij} = ||p_i - p_j||$ and p_k represents the k-th column of P_{model}

$$D_{model} = \begin{bmatrix} 0 & d_1 & d_2 & d_3 \\ d_1 & 0 & d_4 & d_5 \\ d_2 & d_4 & 0 & d_6 \\ d_3 & d_5 & d_6 & 0 \end{bmatrix} \tag{5.2}$$

Considering N robots and then N patterns, we define

$$P^{tot}_{model} = \begin{bmatrix} P^1_{model} & \dots & P^N_{model} \end{bmatrix}$$
$$D^{tot}_{model} = \begin{bmatrix} D^1_{model} & \dots & D^N_{model} \end{bmatrix}$$

where P^i_{model} and D^i_{model} represent respectively the P_{model} and D_{model} of i-th pattern. For any new marker set obtained at time $t > 0$ by cameras, we compute the matrices $P_{cam} \in \mathbb{R}^{3\times n}$ and $D_{cam} \in \mathbb{R}^{n\times n}$ with the same logic. Then we compared each column of D_{cam} with each column of D^{tot}_{model} trying three matches. In this way it was possible to rearrange the matrix P_{cam} so that the markers belonging to the same robot were neighbors and eliminating any extraneous markers to the models.
The algorithm structure is visible in the following pseudocode.

Algorithm 3 Marker Labelling

```
Require: $P_{cam}$, $D_{cam}$, $D^{tot}_{model}$

1: N = size($D^{tot}_{model}$)
2: for i=1 to N do

   Find the index of the column with three correspondences:
3:     $j = whoAmI(D^{tot}_{model}(:, i), D_{cam})$
4:     $P_{ordered}(:, i) = P_{cam}(:, j)$

5: end
```

5.2.2 Pose Reconstruction

Obtained the correspondence between the initial models and each set of points at $t > 0$, we want to implement an algorithm that reconstructs the Wheelphones poses. It is important to underline that, since we need just three points to compute the position and orientation of one vehicle in 3D space, using more markers improve the algorithm robustness in case of one or more misdetections.
The pose reconstruction of any robot was connected to a least squares problem. Indeed, let $P = \begin{bmatrix} p_1 & p_2 & ... & p_n \end{bmatrix}$ and $Q = \begin{bmatrix} q_1 & q_2 & ... & q_n \end{bmatrix}$ be two different sets of points, to find Wheelphone pose is equal to find the matrices $R \in SO(3)$, rotation matrix, and T, translation vector, that solve the following formula:

$$(R, T) = \arg\min_{R,T} \sum_{i=1}^{n} ||(Rp_i + T) - q_i||^2 \tag{5.3}$$

Following the reasoning described in [25], let $\hat{R}$ and $\hat{T}$ be the two solution of (5.3), then Q and $Q' = \hat{R}Q$ have the same centroid:

$$c_Q \equiv c_{Q'}$$

where c_Q and $c_{Q'}$ represent rispectively the controid of Q and Q'.
Therefore, it is possible to divide the least squares problem into two equations to derive the matrices $\hat{R}$ and $\hat{T}$:

- $\hat{R} = \arg\min_R \Sigma^2 = \arg\min_R \sum_{i=1}^{n} ||h'_i - Rh_i||^2$
- $\hat{T} = c_Q - c_P$

where $h'_i = q_i - c_Q$, $h_i = p_i - c_P$ and c_P is the centroid of P.
In our case we want the centroid coincides with the point equidistant from the wheels and belonging to their axis of rotation since in this way, as is easy to guess, it makes the motion control more performing. To calculate the value of centroids, see Fig A, is sufficient to find the intersection of the segments (P1-P3) and (P2-P4). However, this value is not always possible to obtain since, due to inaccuracies of the cameras, the two segments can not lie in the same plane. For this reason we have chosen as the centroid the intersection between the segment (P2-P4) and the plane perpendicular to it containing P1 and P3.
Finally to acquire $\hat{R}$, we resolve the problem using the *singular value decomposition* (SVD). Defining

$$H = \sum_{i=1}^{N} h_i h_i'^T,$$

the SVD of H is

$$H = U \Lambda V^T$$

where U and V are orthonormal matrices with appropriate size. If $det(UV^T)$ is equal to 1, then (UV^T) is our rotation matrix $\hat{R}$.
Once we have found the rotation matrix $\hat{R}$ for the reference frame attached to the object to track, it is easy to extract the Euler angles $[\phi, \theta, \psi]$, namely *roll*, *pitch* and

yaw, which express the rotation along the x, y and z axis. The rotation matrix can be rewritten as follows:

$$\hat{R} = \begin{bmatrix} c_\theta c_\psi - s_\phi s_\psi s_\theta & -s_\psi c_\phi & c_\theta s_\phi c_\psi + c_\psi s_\theta \\ c_\theta s_\psi + c_\psi s_\phi s_\theta & c_\psi c_\phi & s_\psi s_\theta - c_\psi s_\phi c_\theta \\ -c_\phi s_\theta & s_\phi & c_\theta c_\phi \end{bmatrix}$$

which leads to the following relations:

$$\psi = Atan2(-r_{12}, r_{22})$$

$$\phi = Atan2\Big(r_{32}, \sqrt{r_{31}^2 + r_{33}^2}\Big)$$

$$\theta = Atan2(-r_{31}, r_{33})$$

where ϕ belongs to the interval $(-\pi/2, \pi/2)$.
In our case, ϕ and ψ are always zero since the only rotation that the robot can perform is around its z axis. Indeed only θ describes the orientation of the vehicle and for this it is the only angle considered for our experiments.

5.2.3 Implementation

We configured the motion capture system to work with Simulink using the following block scheme:

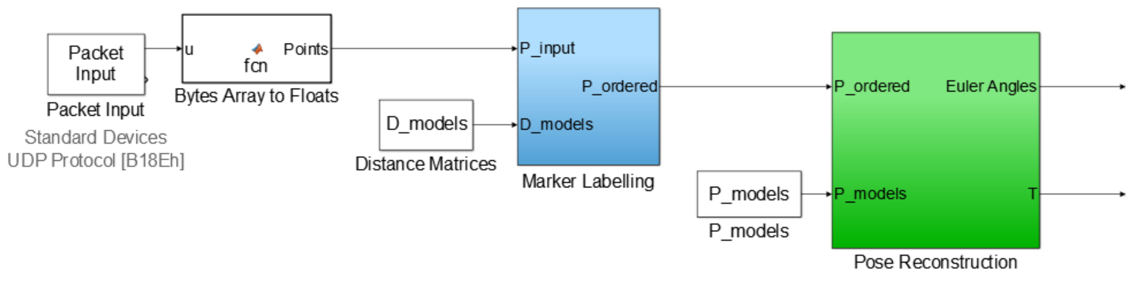


Figure 30: Simulink Motion Capture Block Scheme

The *Packet Input* block receives the packets with the structure of Figure 10 and, through *Bytes Array to Floats* function, derives the spatial coordinates of each marker. The blue and green blocks implement sequentially the algorithms of Markers Labelling and Pose Reconstruction described in the previous sections.
This scheme is performed by the Central-Base which must, in addition to send the reference positions, transmit the actual locations of vehicles moment for moment.
Moreover we repeated the test of Section 6.1 in which the robot tried to follow a sequence of points. In Figure 31 we compared the real trajectories and the real places where the vehicle stopped using two different motion control:

- **Trajectory1** and **Point1** are produced by the control utilizing the wheels odometry,

- **Trajectory2** and **Point2** refer to the control implemented through the Motion Capture System.

As we can see, the second one is much more accurate detecting a maximum positioning error equal to 2 cm.
For this reason, the experimental results which will be shown in section 5.4, are obtained using only the motion control implemented through the Motion Capture System.

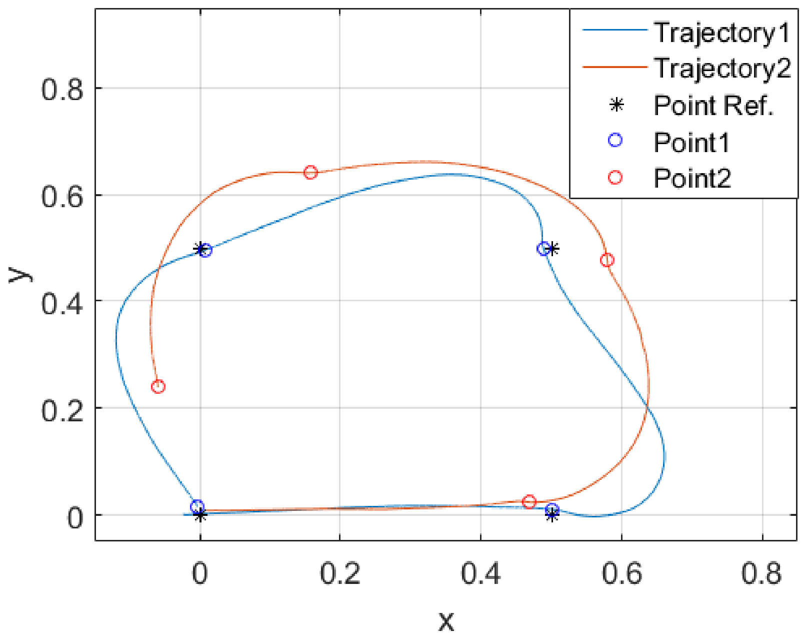


Figure 31: "Sequence of Points" Test2

5.3 Coverage Algorithms Simulations

In this section we want to exhibit how we implemented the partitioning algorithms in MATLAB sofware using the previously described motion control.
We consider a team of N robots and a squared domain $\mathbb{X} = [0,2] \times [0,2]$. The sensory function $\mu(\mathbf{x})$ is a combination of two bi-dimensional Gaussians:

$$\mu(x) = e^{-\frac{\|x-\mu_1\|^2}{0.0313}} + 2e^{-\frac{\|x-\mu_2\|^2}{0.125}} \tag{5.4}$$

where the Gaussian centers are located in position $\mu_1 = [0.675, 0.675]$ and $\mu_2 = [1.625, 1.425]$. This function is shown in Fig. 32 in three-dimensional space.

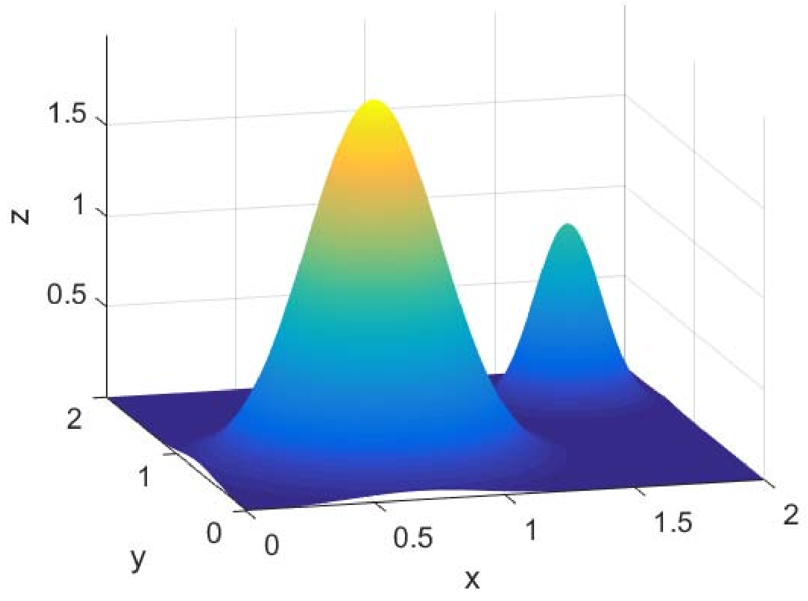


Figure 32: Gaussian Sensory Function

Successively, to use $\mu(\mathbf{x})$ in our algorithms, we represented it as a square matrix I of size equal to 500×500 where $I(i,j) = \mu([x_j, y_i])$. We define

$$k_{i2m} = \frac{2\ meters}{500\ index} = 0,004$$

as the costant that convert the indexs of I in meters

$$\begin{cases} y_i = k_{i2m} i \\ x_j = k_{i2m} j \end{cases} \tag{5.5}$$

The matrix size is an important value that has to be calibrated because a higher value implies that the Gaussians are more accurately discretized while a lower one involves an improvement of the computation time.
The figure 33 shows the matrix I in a grayscale image where the white color is associated with higher values of μ while black with the lower ones.
This matrix is the map on which we apply the two partitioning algorithms.

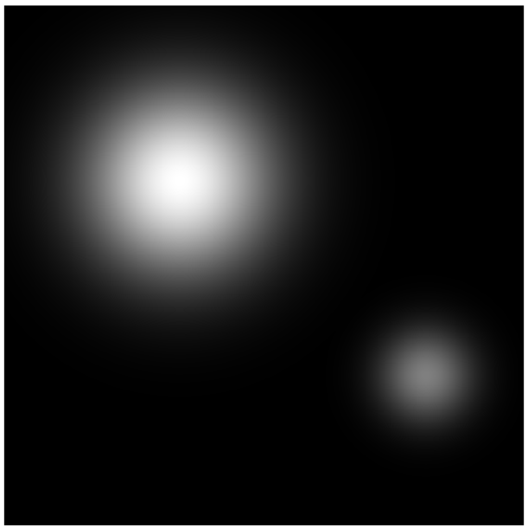


Figure 33: Gaussian Sensory Function

5.3.1 Server-Based Algorithm Implementation

In order to obtain Voronoi paritions and centroids, we create two specific functions: ***VoronoiBounded*** and ***Poly2Centroid***.

- *VoronoiBounded* requires as inputs the positions of the robots and the vertices of the region of interest $\mathbb{X}$. It calculates the partitions v_i through the MATLAB function *voronoin*, it surrounds these in our domain and provides as output the vertices of poligons inside $\mathbb{X}$.

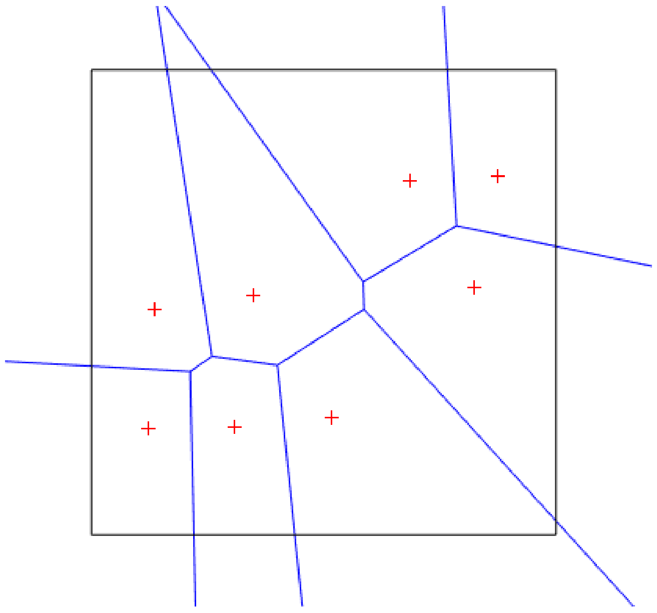


Figure 34: *VoronoiBounded* Output

The Figure 34 shows an example of the *VoronoiBounded* function using a team of 8 robots arranged in a random way. The black and the blue lines represent respectively the boundaries of our domain and those of *voronoin* function generated by the robots position (red cross).

- *Poly2Centroid* requires as inputs the vertices of a polygon and the matrix I that described the sensory function μ. It considers only the values of I inside the polygon, calculates the centroid throgh the formula 3.3 and provides as output the coordinates c_x and c_y of the point.

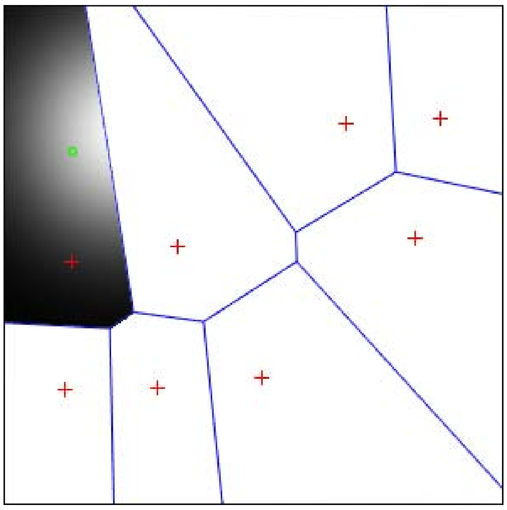


Figure 35: *Poly2Centroid* Output

The Figure 35 displays an example of *Poly2Centroid* function. Using the vertices of the upper left partition of Fig.34 it cuts out the sensory function computing its centroid (green circle).

To use both functions in MATLAB, we installed the Mapping Toolbox that provides algorithms for performing operations on polygons.
Setting the values of c_x and c_y as "*Desired X*" and "*Desired Y*", the server base trasmits them to phone moving the vehicle to the centroid of its partition. Finally this process "*partitioning*, *centering* and *moving*" was performed iteratively several times. The Server-Based results are exposed in section 5.4.

5.3.2 Gossip Distributed Algorithm Implementation

The Gossip Distributed algorithm requires that, through a communication network, the robots theirself store and calculate their partition without an external source which oversees the process. This idea is inconsistent with our situation because is the central base the one who calculates where the vehicles are located in the plane. Without it the motion control, that is the basis of our partitioning algorithm, would be impossible to implement. Furthermore, if we left to the vehicles deciding when and to whom to transmit, many communication issues, for example the packets collision, would be managed.
For these reasons we chose to simulate the distributed feature through a central base. The central server knows each robots partition but it considers only two of

them in any iteration of the algorithm.
We implemented the Gossip Distributed algorithm in MATLAB performing the following steps:

1. We randomly select two different agent i, j. For these two vehicles we know their positions $\mathbf{x}_i$, $\mathbf{x}_j$ and the vertices $\mathbf{v}_i$, $\mathbf{v}_j$ of their dominance regions.

2. We use a default MATLAB function, *polybool*, that perform the union between two polygon given their vertices, using the following script:
$$\mathbf{v}_U = \mathit{polybool}('or', \mathbf{v}_i, \mathbf{v}_j);$$
where $\mathbf{v}_U$ values are the vertices of resulting polygon. Furthermore this function handles the case in which the polygons are separated from each other. If this happens, the *polybool* output will contain the $\mathbf{v}_i$ vertices, a *Not a Number* (NaN) and the $\mathbf{v}_j$ vertices. In this way we can check the adjacency condition before continuing with the execution of the algorithm.

3. Let x_i, y_i, x_j, y_j be the spatial coordinates of the robots positions $\mathbf{x}_i$ and $\mathbf{x}_j$, the straight line passing through these points has the following equation:
$$\frac{y - y_i}{y_j - y_i} = \frac{x - x_i}{x_j - x_i}$$
Let $\mathbf{x}_M = (x_M, y_M)$ be the central point between $\mathbf{x}_i$ and $\mathbf{x}_j$,
$$\mathbf{x}_M = \left(\frac{x_i + x_j}{2}, \frac{y_i + y_j}{2}\right),$$
in order to implement $H_{bisector}$ (formula..), we consider the straight line perpendicular to the previous one, passing through $\mathbf{x}_M$. It has the following equation:
$$(y - y_M) = -\left(\frac{x_j - x_i}{y_j - y_i}\right)(x - x_M)$$
This line cuts the squared domain $\mathbb{X}$ in two regions whose vertices are store in different array $\mathbf{v}_1$ and $\mathbf{v}_2$.

4. We use again the *polybool* function but this time to perform two intersections between polygons. The first intersection is generated by the Union polygon and the region with $\mathbf{v}_1$ vertices while the second one derives from the same polygon and the other region of Step 3. We utilized the following scrip:
$$\mathbf{v}_{U1} = \mathit{polybool}('and', \mathbf{v}_U, \mathbf{v}_1);$$
$$\mathbf{v}_{U2} = \mathit{polybool}('and', \mathbf{v}_U, \mathbf{v}_2);$$
where $\mathbf{v}_{U1}$, $\mathbf{v}_{U2}$ values are the vertices of resulting polygons. Moreover in this function it is also handled the case in which the resulting polygons have disconnected areas. If this happens, the *polybool* output will contain at least a *Not a Number*. In this way we can check the connectivity condition before continuing with the execution of the algorithm. The polygons created by $\mathbf{v}_{U1}$ and $\mathbf{v}_{U2}$ vertices are the new partitions of i and j agents overwriting $\mathbf{v}_i$ and $\mathbf{v}_j$.

5. Using the function *Poly2Centroid*, described in the previous section, we are able to find the new centroids $\mathbf{x}_i$ and $\mathbf{x}_j$.

6. The central base transmits to the phones these informations and returns to Step 1.

The following figure show graphically the step performed by GB algorithm.

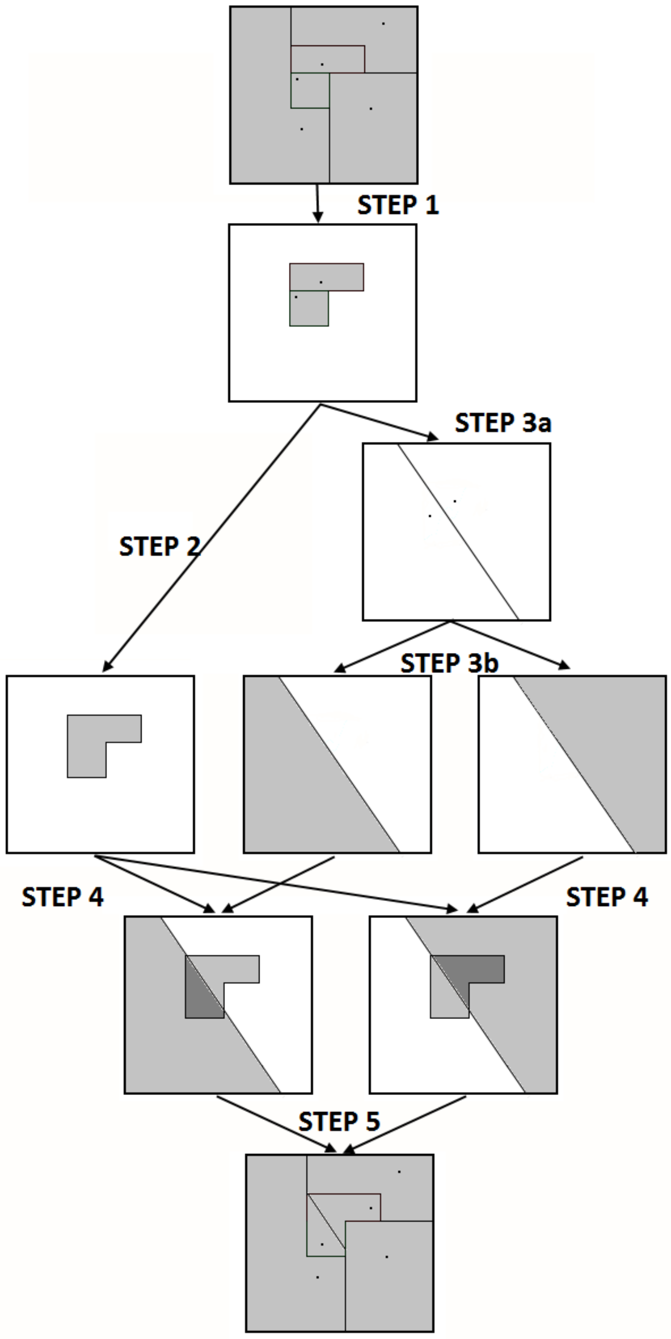


Figure 36: Evolution of the Partition in GB algorithm

5.4 Results

In this section we show the results obtained by the Server Based and Gossip Distributed algorithms. They are implemented both on ideal unicycles vehicles and wheelphones devices. In the first case we used the motion control based on the results given by the wheels odometry. In the other case the motion control designed was based on the information resulting from the Motion Capture System.
To compare the four experiments we decided to use the same initial conditions (Fig.37):

- **5 robots**: four vehicles positioned in the corners of the squared domain and one located in the central point
- **Partitions**: the dominance region of each agent is the Voronoid partition given by the initial positions of the robots
- **Map**: the sensory function μ described in Section 5.3.

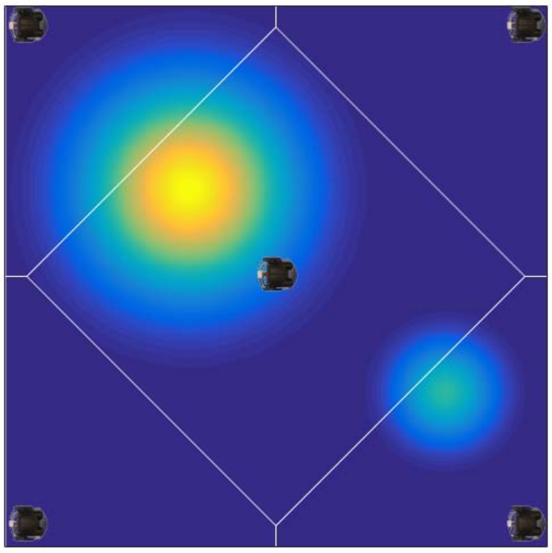


Figure 37: Initial conditions.

Each algorithm was processed 40 times and for each iteration the central server stored the values of the vehicles position and the boundary of their partitions. In the following figures are shown some of the 40 iterations, specifically the numbers 1, 5, 20 and 40. Since for each algorithm we made two experiments, the figures that we obtained are four. Figures 38 and 39 refer to the Server Based approach, while figures 40 and 41 relate to the Gossip Distributed proces.

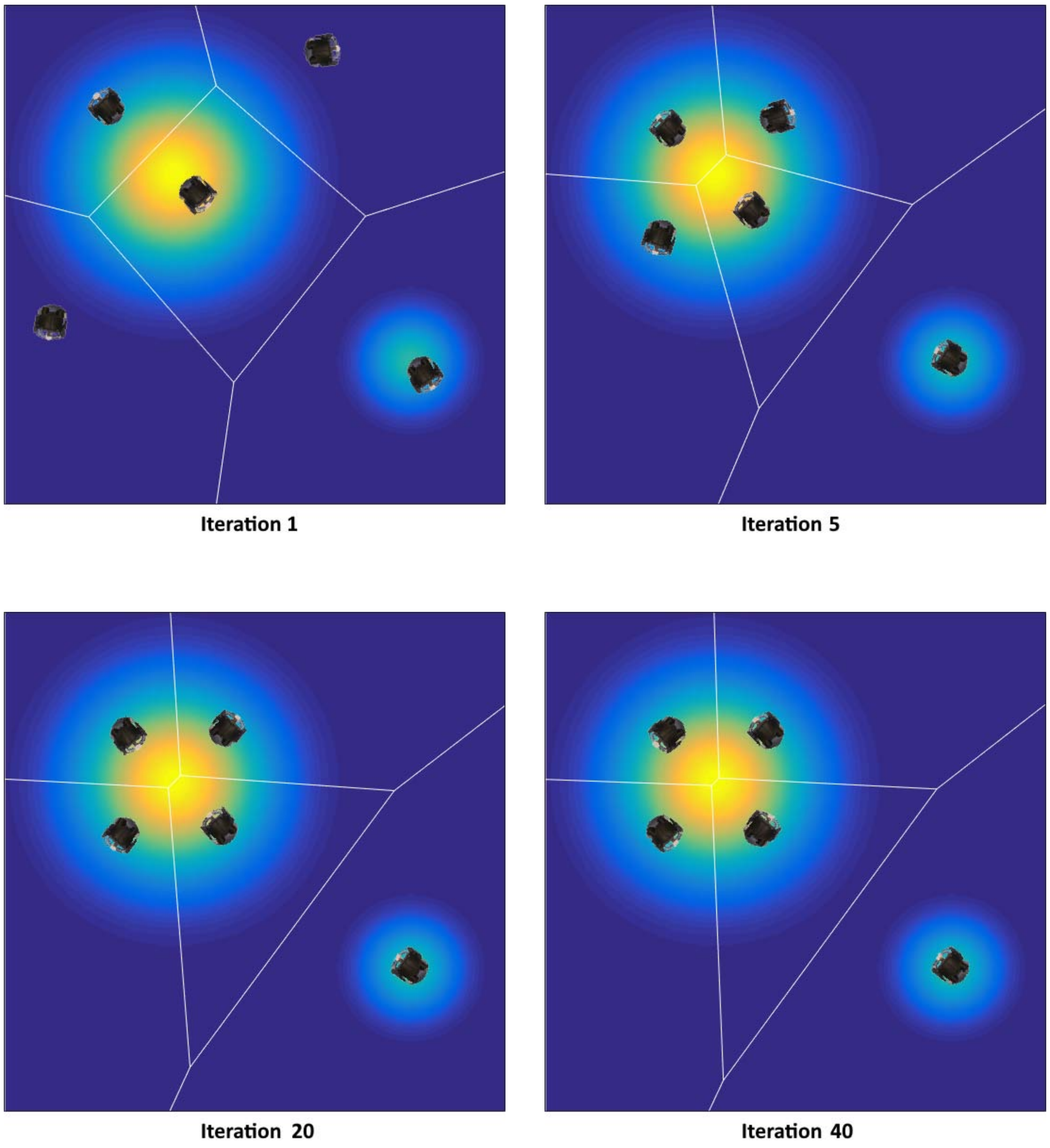


Figure 38: Evolution of Simulate Server Based Algorithm

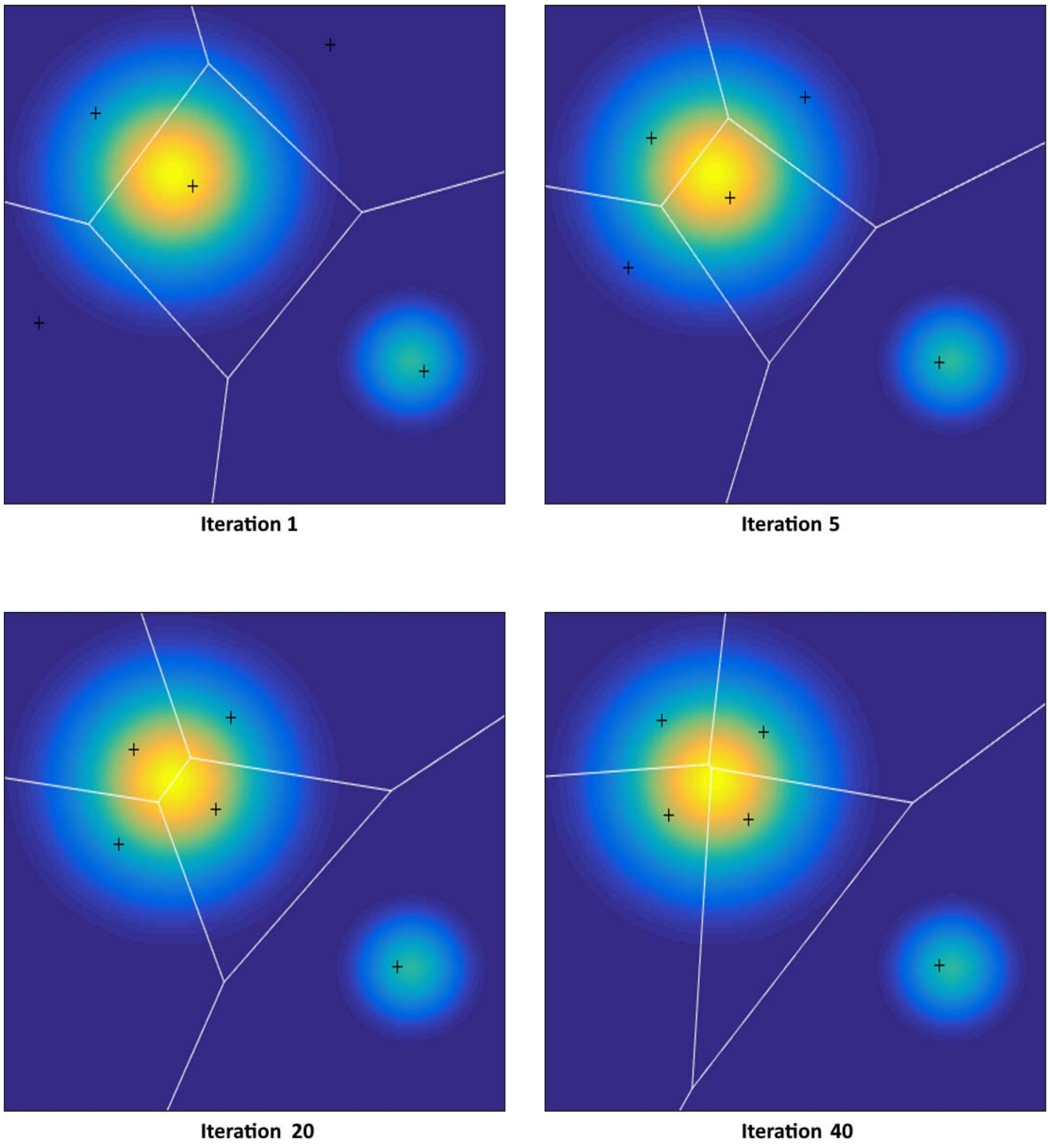


Figure 39: Evolution of Experimental Server Based Algorithm

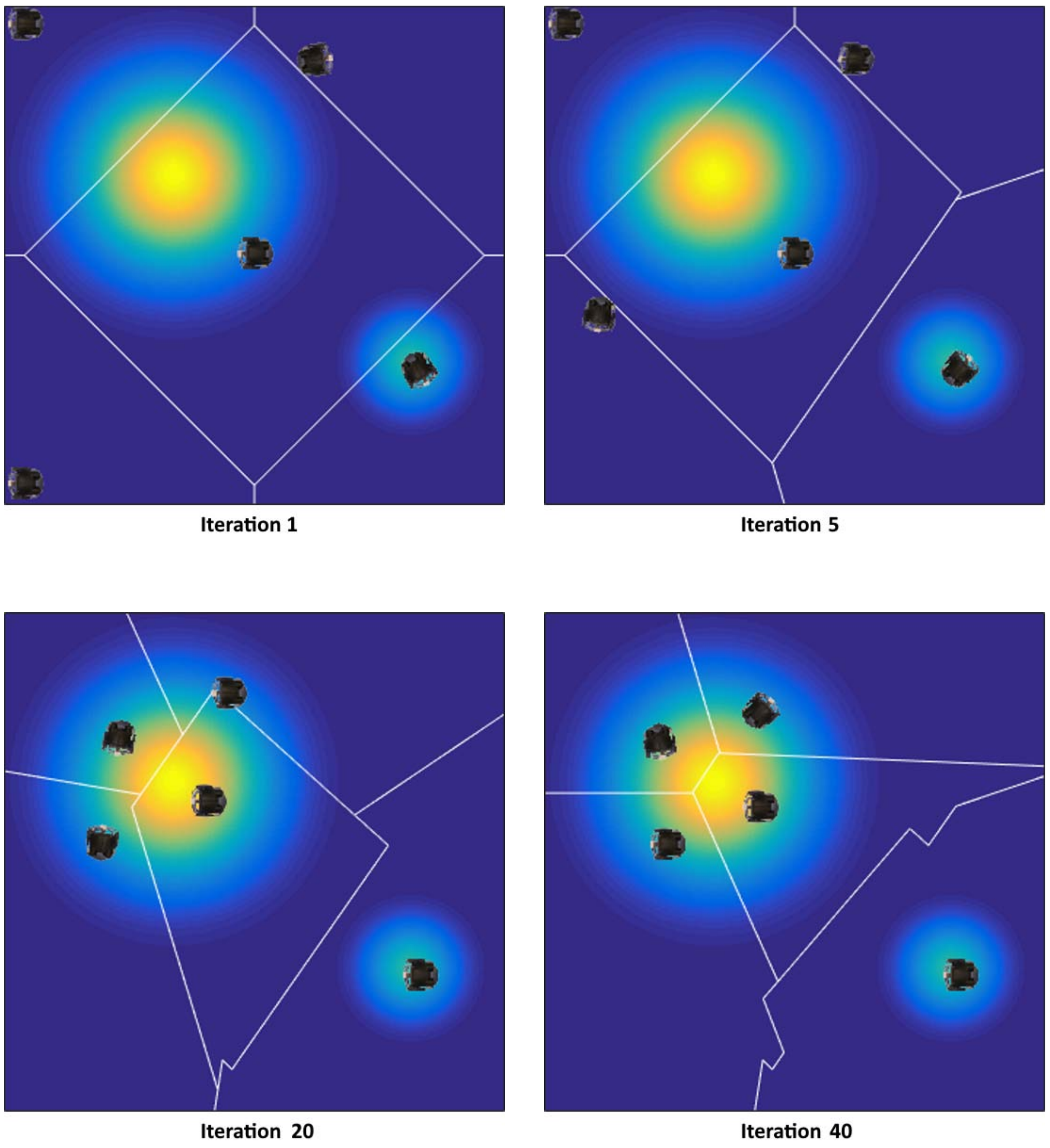


Figure 40: Evolution of Simulate Gossip Distributed Algorithm

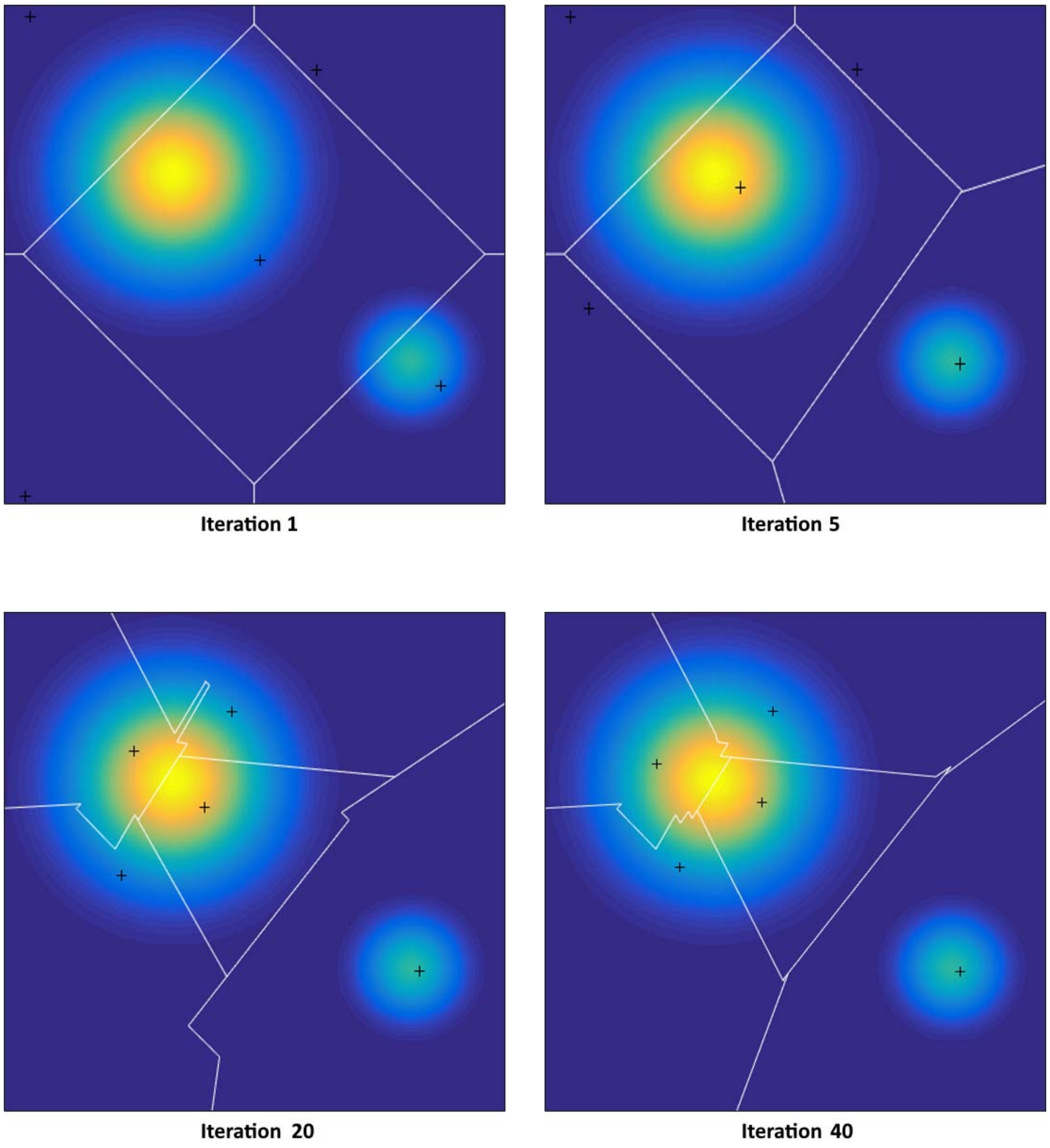


Figure 41: Evolution of Experimental Gossip Distributed Algorithm

As it is possible to see, for all the four experiments the final positions are almost the same: four robots surrond the most critical area (the yellow zone) while the remaining vehicle stand on the less critical one. This result reflects our expectations since the sensory function is generated by the union of two Gaussian that have one the twice amplitude respect to the other. Hence the most important zone have to be monitored by more robots than the other area. Moreover, considering that each iteration is depending on the previous one and the initial condition is the same for all the experiments, it was predictable that the final positions were similar.
Analysing the partitions, we can see that there is a difference between the Server Based approach and the Gossip Distributed one. In the first case there are always convex partitions with a low number of vertices and simply shapes. This is due to the global characteristic of the algorithm which obtain the dominance regions studying all the agents position. In the second case we can find concave partitions with complicated shapes. This is caused by pairwise comunications between agents, in fact if two robots do not interact for many interations, the boundary between their partitions can be a jagged line. In our case, since the number of robots used is small, this problem have not affected the algorithm performances. In situations with a huge number of agents, it is advisable to not choose randomly the two robots that have to communicate, but to select them following specific criteria like, for example, by increasing the communication possibility between vehicles with several vertices partition.

To analyse the goodness of the algorithms we have to evaluate the Multicenter function H. Figure 42 shows the relation of H (Eq.3.5) to the number of iterations.

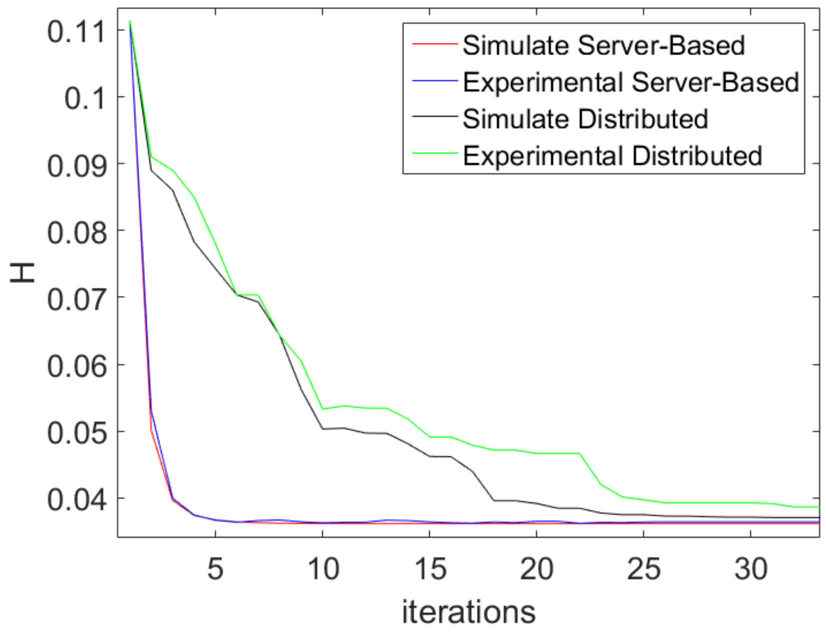


Figure 42: Evolution of the cost function H.

As expected, for all experiments the values of the cost function converge to the same level that is one third of the initial one. This imply that the final partitions coverage

the domain more efficiently than the initial conditions. Furthermore, since the SB and GD algorithms are based on the classical method of Lloyd, H is characterized by a non-increasing behavior.
Evaluating now the convergence rate of the function, we can see how the Server Based algorithm is faster than the Gossip Distributed one. This difference is attributed again to the different approach of the algorithms. In the first case all the vehicles are always involved in any iteration while in the second one only two of them.
Finally we can note a worsening performance in the experiments where we used real devices respect to ideal ones. This is caused by the different accuracy of the motion control. In the ideal case the robots are in the precise locations where the partitioning algorithms commanded to stand. Instead, implementing the control on real devices, although we utilized the Motion Capture System, there were always small positioning errors.

6 Conclusions

In this work two partitioning algorithms were implemented on Wheelphones devices in order to coverage optimally a specific area. We ran this process through an Application for smartphones enhancing the traditional coverage methods.
After a brief introduction on the importance that this technology is assuming in these years, we described the pre-existing hardware and software features of a Wheelphone robot. Thanks to them and the Support Package for Android of MATLAB, we were able to build a custom S-Function block in Simulink in order to handle the low-level communication between the robot and the smartphone. This step has been crucial for the success of our implementation as it allowed us to program with a high-level language enabling a faster addition or change of the control components.
Successively we exposed the geometric concepts of *Voroid partition* and *centroid* used by classic Lloyd method in order to find the optimal area partition. The algorithms proposed in this thesis were two versions of this method with different communication architecture. The first one employed a centralized approach where each agent have to communicate only its position to a central server in a synchronous manner. Afterwards, this server calculates the new partitions and positions retransmitting them to the vehicles. The second algorithm followed a distributed method where only two agents are involved in any iteration. In this case the vehicles themselves have to compute the information about their partitions and positions.
Later we provided the mathematical model for an unicycle vehicle that is the same type of Wheelphone robot. Studing its dynamic and kinematic we created a simple motion control that allowed us to drive the vehicle from one point to another.
Finally we showed the work made in laboratory on Wheelphone devices. Before that it was necessary to implement the *Marker Labelling* and *Pose Reconstraction* algorithms essential to estimate the positions and orientations of the robots.
On the basis of the results obtained, we verified the right functioning of the algorithms succeeding in minimizing the Multicenter Function H in 5 and 35 iterations respectively for the Server Based algorithm and the Gossip Distributed one.

Many future developments can be led at each level of the proposed work. From a theorical point of view can be tested more complex partitioning algorithms including a non convex enviroment with obstacles presence and time-varying density functions. Then it is possible to exploit the phone and Wheelphone sensors, like camera, I-R and ambient sensors. In this way the robots themselves can estimate the map that has to be covered. In this regard we suggest the implentation of the algorithms of [22] where the map estimation phase and the coverage one are simultaneoisly ran. Furthermore can be designed a more robust and reliable motion control. In addition to moving the robots from one point to another, it has to decide which trajectory they have to follow. In this way the vehicles can avoid the other agents or obstacles present in the area. Finally from the pratical point of view, can be installed on Wheelphone two motors encoders in order to know exactly the angular positions of the wheels. Managing the low-level communication, the Motion Control can be indipendent from the Motion Capture System of the laboratory making robots able to act in all type of environment.

7 APPENDIX

A Wheelphone Drivers

A.1 *sfun_wheelphone.c*

```
#define S_FUNCTION_NAME  sfun_wheelphone
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

#define NUM_INPUTS          1
#define NUM_OUTPUTS         11
#define NUM_PARAMS          0
#define NUM_CONT_STATES     0
#define NUM_DISC_STATES     0

#define SAMPLE_TIME_0       INHERITED_SAMPLE_TIME

/* Input Port  0 (Left/right speed ref)  */
#define INPUT_0_WIDTH       2
#define INPUT_0_DTYPE       SS_INT32
#define INPUT_0_COMPLEX     COMPLEX_NO
#define INPUT_0_FEEDTHROUGH 1

/* Output Port  0 (front proximity sensors)  */
#define OUTPUT_0_WIDTH      4
#define OUTPUT_0_DTYPE      SS_UINT8
#define OUTPUT_0_COMPLEX        COMPLEX_NO

/* Output Port  1 (front ambient sensors)  */
#define OUTPUT_1_WIDTH      4
#define OUTPUT_1_DTYPE      SS_UINT8
#define OUTPUT_1_COMPLEX        COMPLEX_NO

/* Output Port  2 (ground proximity sensors)  */
#define OUTPUT_2_WIDTH      4
#define OUTPUT_2_DTYPE      SS_UINT8
#define OUTPUT_2_COMPLEX        COMPLEX_NO

/* Output Port  3 (ground ambient sensors)  */
#define OUTPUT_3_WIDTH      4
#define OUTPUT_3_DTYPE      SS_UINT8
#define OUTPUT_3_COMPLEX        COMPLEX_NO

/* Output Port  4 (battery charge [%])  */
#define OUTPUT_4_WIDTH      1
#define OUTPUT_4_DTYPE      SS_UINT8
#define OUTPUT_4_COMPLEX        COMPLEX_NO

/* Output Port  5 (estimated left/right speed [mm/s])  */
#define OUTPUT_5_WIDTH      2
#define OUTPUT_5_DTYPE      SS_INT16
#define OUTPUT_5_COMPLEX        COMPLEX_NO
```

```
/* Output Port  6 (odometry - x [m], y [m], yaw [rad])  */
#define OUTPUT_6_WIDTH      3
#define OUTPUT_6_DTYPE      SS_DOUBLE
#define OUTPUT_6_COMPLEX        COMPLEX_NO

/* Output Port  7 (battery charging state)  */
#define OUTPUT_7_WIDTH      1
#define OUTPUT_7_DTYPE      SS_UINT8
#define OUTPUT_7_COMPLEX        COMPLEX_NO

/* Output Port  8 (odometry calibration flag)  */
#define OUTPUT_8_WIDTH      1
#define OUTPUT_8_DTYPE      SS_UINT8
#define OUTPUT_8_COMPLEX        COMPLEX_NO

/* Output Port  9 (obstacle avoidance flag)  */
#define OUTPUT_9_WIDTH      1
#define OUTPUT_9_DTYPE      SS_UINT8
#define OUTPUT_9_COMPLEX        COMPLEX_NO

/* Output Port  10 (cliff avoidance flag)  */
#define OUTPUT_10_WIDTH     1
#define OUTPUT_10_DTYPE     SS_UINT8
#define OUTPUT_10_COMPLEX       COMPLEX_NO


/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes
   ======================================== */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, NUM_PARAMS);  /* Number of expected
       parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        /* Return if number of expected != number of actual
           parameters */
        return;
    }

    ssSetNumContStates(S, NUM_CONT_STATES);
    ssSetNumDiscStates(S, NUM_DISC_STATES);

    if (!ssSetNumInputPorts(S, NUM_INPUTS)) return;

    ssSetInputPortWidth(S, 0, INPUT_0_WIDTH);
    ssSetInputPortDataType(S, 0, INPUT_0_DTYPE);
    ssSetInputPortComplexSignal(S, 0, INPUT_0_COMPLEX);
    ssSetInputPortDirectFeedThrough(S, 0, INPUT_0_FEEDTHROUGH);
    ssSetInputPortRequiredContiguous(S, 0, 1); /*direct input
       signal access*/

    if (!ssSetNumOutputPorts(S, NUM_OUTPUTS)) return;
```

```
ssSetOutputPortWidth(S, 0, OUTPUT_0_WIDTH);
ssSetOutputPortDataType(S, 0, OUTPUT_0_DTYPE);
ssSetOutputPortComplexSignal(S, 0, OUTPUT_0_COMPLEX);

ssSetOutputPortWidth(S, 1, OUTPUT_1_WIDTH);
ssSetOutputPortDataType(S, 1, OUTPUT_1_DTYPE);
ssSetOutputPortComplexSignal(S, 1, OUTPUT_1_COMPLEX);

ssSetOutputPortWidth(S, 2, OUTPUT_2_WIDTH);
ssSetOutputPortDataType(S, 2, OUTPUT_2_DTYPE);
ssSetOutputPortComplexSignal(S, 2, OUTPUT_2_COMPLEX);

ssSetOutputPortWidth(S, 3, OUTPUT_3_WIDTH);
ssSetOutputPortDataType(S, 3, OUTPUT_3_DTYPE);
ssSetOutputPortComplexSignal(S, 3, OUTPUT_3_COMPLEX);

ssSetOutputPortWidth(S, 4, OUTPUT_4_WIDTH);
ssSetOutputPortDataType(S, 4, OUTPUT_4_DTYPE);
ssSetOutputPortComplexSignal(S, 4, OUTPUT_4_COMPLEX);

ssSetOutputPortWidth(S, 5, OUTPUT_5_WIDTH);
ssSetOutputPortDataType(S, 5, OUTPUT_5_DTYPE);
ssSetOutputPortComplexSignal(S, 5, OUTPUT_5_COMPLEX);

ssSetOutputPortWidth(S, 6, OUTPUT_6_WIDTH);
ssSetOutputPortDataType(S, 6, OUTPUT_6_DTYPE);
ssSetOutputPortComplexSignal(S, 6, OUTPUT_6_COMPLEX);

ssSetOutputPortWidth(S, 7, OUTPUT_7_WIDTH);
ssSetOutputPortDataType(S, 7, OUTPUT_7_DTYPE);
ssSetOutputPortComplexSignal(S, 7, OUTPUT_7_COMPLEX);

ssSetOutputPortWidth(S, 8, OUTPUT_8_WIDTH);
ssSetOutputPortDataType(S, 8, OUTPUT_8_DTYPE);
ssSetOutputPortComplexSignal(S, 8, OUTPUT_8_COMPLEX);

ssSetOutputPortWidth(S, 9, OUTPUT_9_WIDTH);
ssSetOutputPortDataType(S, 9, OUTPUT_9_DTYPE);
ssSetOutputPortComplexSignal(S, 9, OUTPUT_9_COMPLEX);

ssSetOutputPortWidth(S, 10, OUTPUT_10_WIDTH);
ssSetOutputPortDataType(S, 10, OUTPUT_10_DTYPE);
ssSetOutputPortComplexSignal(S, 10, OUTPUT_10_COMPLEX);

ssSetNumSampleTimes(S, 1);
ssSetNumRWork(S, 0);
ssSetNumIWork(S, 0);
ssSetNumPWork(S, 0);
ssSetNumModes(S, 0);
ssSetNumNonsampledZCs(S, 0);

/* Specify the sim state compliance to be same as a built-in
    block */
ssSetSimStateCompliance(S, USE_DEFAULT_SIM_STATE);
```

```
    ssSetOptions(S, 0);
}

/* Function: mdlInitializeSampleTimes
   ================================== */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, SAMPLE_TIME_0);
    ssSetOffsetTime(S, 0, 0.0);

    //  notify need to access absolute time data
    ssSetNeedAbsoluteTime(S, 1);
}


#undef MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove
    function */
#if defined(MDL_INITIALIZE_CONDITIONS)
  /* Function: mdlInitializeConditions
     ================================= */
  static void mdlInitializeConditions(SimStruct *S)
  {
  }
#endif /* MDL_INITIALIZE_CONDITIONS */



#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart
     =============================================== */
  static void mdlStart(SimStruct *S)
  {
  }
#endif /*  MDL_START */



/* Function: mdlOutputs
   ================================================ */
static void mdlOutputs(SimStruct *S, int_T tid)
{
}



#undef MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  /* Function: mdlUpdate
     ============================================== */
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
  }
#endif /* MDL_UPDATE */
```

```
#undef MDL_DERIVATIVES  /* Change to #undef to remove function
   */
#if defined(MDL_DERIVATIVES)
  /* Function: mdlDerivatives
      ========================================== */
  static void mdlDerivatives(SimStruct *S)
  {
  }
#endif /* MDL_DERIVATIVES */



/* Function: mdlTerminate
   ============================================== */
static void mdlTerminate(SimStruct *S){
}


/*=============================*
 * Required S-function trailer *
 *=============================*/

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a
   MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration
   function */
#endif
```

A.2 *sfun_wheelphone.tlc*

```
%implements "sfun_wheelphone" "C"

%include "utillib.tlc"

%assign ::includeWheelphoneSupport = 1

%function BlockTypeSetup(block, system) void
    %% %<LibAddToCommonIncludes("driver_wheelphone.h")>
    %openfile buffer
    extern void initWheelphone(void);

    extern void getWheelphoneData(time_t, uint8_t *, uint8_t *,
       uint8_t *,
                    uint8_t *, uint8_t *, int16_t *, double *,
                       uint8_t *,
                    uint8_t *, uint8_t *, uint8_t *);

    extern void setWheelphoneSpeed(int32_t *, int32_t *);

    extern void terminateWheelphone(void);
    %closefile buffer
    %<LibCacheFunctionPrototype(buffer)>
    %<LibAddToModelSources("driver_wheelphone")>
%endfunction %% BlockTypeSetup


%function Start(block, system) Output
    initWheelphone();
%endfunction %% Start


%function Outputs(block, system) Output
    /* %<Type> Block: %<Name> */

    %assign pu0 = LibBlockInputSignalAddr(0, "", "", 0)
    %assign pu1 = LibBlockInputSignalAddr(0, "", "", 1)

    %assign py0 = LibBlockOutputSignalAddr(0, "", "", 0)
    %assign py1 = LibBlockOutputSignalAddr(1, "", "", 0)
    %assign py2 = LibBlockOutputSignalAddr(2, "", "", 0)
    %assign py3 = LibBlockOutputSignalAddr(3, "", "", 0)
    %assign py4 = LibBlockOutputSignalAddr(4, "", "", 0)
    %assign py5 = LibBlockOutputSignalAddr(5, "", "", 0)
    %assign py6 = LibBlockOutputSignalAddr(6, "", "", 0)
    %assign py7 = LibBlockOutputSignalAddr(7, "", "", 0)
    %assign py8 = LibBlockOutputSignalAddr(8, "", "", 0)
    %assign py9 = LibBlockOutputSignalAddr(9, "", "", 0)
    %assign py10 = LibBlockOutputSignalAddr(10, "", "", 0)

    getWheelphoneData(%<LibGetTaskTimeFromTID(block)>,
        %<py0>, %<py1>, %<py2>, %<py3>,
        %<py4>, %<py5>, %<py6>,
        %<py7>, %<py8>, %<py9>, %<py10>);
    setWheelphoneSpeed(%<pu0>, %<pu1>);
%endfunction %% Outputs


%function Terminate(block, system) Output
    terminateWheelphone();
%endfunction %% Terminate
```

A.3 *driver_wheelphone.c*

```
#include <jni.h>
#include <stdlib.h>
#include <math.h>

#define ENCODED_STATE_LEN    63
#define MAX_BATTERY_VALUE    152
#define SPEED_THR            3

#define NOT_CHARGING         0
#define CHARGING             1
#define CHARGED              2

#define LEFT_DIAM_COEFF      1.0
#define RIGHT_DIAM_COEFF         1.0
#define WHEEL_BASE           0.087

extern JavaVM *cachedJvm;
extern jobject cachedActivityObj;
extern jclass cachedMainActivityCls;
static jmethodID sgWheelphoneGetEncodedRobotStateID;
static jmethodID sgWheelphoneSetSpeedID;
static jmethodID sgWheelphoneSendCmdID;

static double leftDist = 0.0;
static double rightDist = 0.0;
static double leftDistPrev = 0.0;
static double rightDistPrev = 0.0;

static double timePrev = 0.0;

static double odometryX = 0.0;
static double odometryY = 0.0;
static double odometryTheta = 0.0;


void initWheelphone(void)
{
    JNIEnv *pEnv;
    (*cachedJvm)->AttachCurrentThread(cachedJvm, &pEnv, NULL);

    sgWheelphoneGetEncodedRobotStateID = (*pEnv)->GetMethodID(
       pEnv, cachedMainActivityCls,
            "getWheelphoneEncodedState","()[B");

    sgWheelphoneSetSpeedID = (*pEnv)->GetMethodID(pEnv,
       cachedMainActivityCls,
            "setWheelphoneSpeed", "(II)V");
    sgWheelphoneSendCmdID = (*pEnv)->GetMethodID(pEnv,
       cachedMainActivityCls,
            "sendWheelphoneCommands", "()V");
}
```

```
void getWheelphoneData(time_t time, uint8_t *frontProxs, uint8_t
    *frontAmbients,
        uint8_t *groundProxs, uint8_t *groundAmbients,
        uint8_t *batteryCharge,
        int16_t *estimatedSpeed,
        double *odometry,
        uint8_t *chargeState,
        uint8_t *odomCalibFinish, uint8_t *
           obstacleAvoidanceEnabled, uint8_t *
           cliffAvoidanceEnabled)
{
    if (sgWheelphoneGetEncodedRobotStateID != NULL)
    {
        JNIEnv *pEnv;
        jbyteArray ret;
        jbyte encodedState[ENCODED_STATE_LEN];
        int16_t leftMeasuredSpeed, rightMeasuredSpeed;
        uint8_t batteryRaw;
        float batteryVoltage;
        uint8_t flagRobotToPhone;
        time_t totalTime;
        double deltaDist;


        (*cachedJvm)->AttachCurrentThread(cachedJvm, &pEnv, NULL
           );

        ret = (jintArray)(*pEnv)->CallObjectMethod(pEnv,
           cachedActivityObj, sgWheelphoneGetEncodedRobotStateID
           );
        if ((*pEnv)->ExceptionCheck(pEnv))
            return; /* Exception during execution of
               sgWheelphoneGetEncodedRobotStateID */

        (*pEnv)->GetByteArrayRegion(pEnv, ret, 0,
           ENCODED_STATE_LEN, (jbyte *)encodedState);
        if ((*pEnv)->ExceptionCheck(pEnv))
        {
            (*pEnv)->DeleteLocalRef(pEnv, ret);
            return; /* ArrayIndexOutOfBoundsException */
        }

        frontProxs[0] = (uint8_t)encodedState[1];
        frontProxs[1] = (uint8_t)encodedState[2];
        frontProxs[2] = (uint8_t)encodedState[3];
        frontProxs[3] = (uint8_t)encodedState[4];

        frontAmbients[0] = (uint8_t)encodedState[5];
        frontAmbients[1] = (uint8_t)encodedState[6];
        frontAmbients[2] = (uint8_t)encodedState[7];
        frontAmbients[3] = (uint8_t)encodedState[8];

        groundProxs[0] = (uint8_t)encodedState[9];
        groundProxs[1] = (uint8_t)encodedState[10];
        groundProxs[2] = (uint8_t)encodedState[11];
        groundProxs[3] = (uint8_t)encodedState[12];
```

```
groundAmbients[0] = (uint8_t)encodedState[13];
groundAmbients[1] = (uint8_t)encodedState[14];
groundAmbients[2] = (uint8_t)encodedState[15];
groundAmbients[3] = (uint8_t)encodedState[16];

// battery level (from 0 to maxBatteryValue=152)
batteryRaw = (uint8_t)encodedState[17];

//      battery voltage (from 3.5 to 4.2 volts)
//  915 is the ADC out at 4.2 volts
//  763 is the ADC out at 3.5 volts
//  the "battery" variable actually contains the "
    sampled value - 763"
batteryVoltage = 4.2*(float)((batteryRaw + 763)/915.0);

//  remaining battery charge (from 0% to 100%)
*batteryCharge = 100*batteryRaw/MAX_BATTERY_VALUE;

flagRobotToPhone = (uint8_t)encodedState[18];

//  estimated speed (from back EMF) in [mm/s]
leftMeasuredSpeed = (uint8_t)encodedState[19] + 256*(
    uint8_t)encodedState[20];
rightMeasuredSpeed = (uint8_t)encodedState[21] + 256*(
    uint8_t)encodedState[22];

if (abs(leftMeasuredSpeed) < SPEED_THR) {
     leftMeasuredSpeed = 0;
}
if (abs(rightMeasuredSpeed) < SPEED_THR) {
     rightMeasuredSpeed = 0;
}

estimatedSpeed[0] = leftMeasuredSpeed;
estimatedSpeed[1] = rightMeasuredSpeed;

//  estimated travelled distance [m]
leftDistPrev = leftDist;
rightDistPrev = rightDist;
totalTime = time - timePrev;
leftDist += (leftMeasuredSpeed/1000.0 * totalTime) *
    LEFT_DIAM_COEFF;
rightDist += (rightMeasuredSpeed/1000.0 * totalTime) *
    RIGHT_DIAM_COEFF;
deltaDist = ((rightDist - rightDistPrev) + (leftDist -
    leftDistPrev)) / 2.0;

//  odometry [m], [rad]
odometryX += cos(odometryTheta) * deltaDist;
odometryY += sin(odometryTheta) * deltaDist;
odometryTheta = ((rightDist - leftDist) / WHEEL_BASE) /
    1000.0;

odometry[0] = odometryX;
odometry[1] = odometryY;
```

```
        odometry[2] = odometryTheta;

        //  battery charging state
        if ((flagRobotToPhone & 0x20) == 0x20) {
            if ((flagRobotToPhone & 0x40) == 0x40) {
                *chargeState = CHARGED;
            } else {
                *chargeState = CHARGING;
            }
        } else {
            *chargeState = NOT_CHARGING;
        }

        //  odometry calibration flag
        if ((flagRobotToPhone & 0x80) == 0x80) {
            *odomCalibFinish = 1;
        } else {
            *odomCalibFinish = 0;
        }

        //  obstacle avoidance flag
        if ((flagRobotToPhone & 0x01) == 0x01) {
            *obstacleAvoidanceEnabled = 1;
        } else {
            *obstacleAvoidanceEnabled = 0;
        }

        //  cliff avoidance flag
        if ((flagRobotToPhone & 0x02) == 0x02) {
            *cliffAvoidanceEnabled = 1;
        } else {
            *cliffAvoidanceEnabled = 0;
        }

        (*pEnv)->DeleteLocalRef(pEnv, ret);
    }

}


void setWheelphoneSpeed(int32_t *lSpeed, int32_t *rSpeed)
{
    if ( (sgWheelphoneSetSpeedID != NULL) && (
        sgWheelphoneSendCmdID != NULL) )
    {
        JNIEnv *pEnv;
        jint jlSpeed = (jint)*lSpeed;
        jint jrSpeed = (jint)*rSpeed;

        (*cachedJvm)->AttachCurrentThread(cachedJvm, &pEnv, NULL
           );

        (*pEnv)->CallVoidMethod(pEnv, cachedActivityObj,
           sgWheelphoneSetSpeedID,
                jlSpeed, jrSpeed);
        if ((*pEnv)->ExceptionCheck(pEnv))
```

```
                return; /* Exception during execution of
                    sgWheelphoneSetSpeedID */

            (*pEnv)->CallVoidMethod(pEnv, cachedActivityObj,
                sgWheelphoneSendCmdID);
            if ((*pEnv)->ExceptionCheck(pEnv))
                return; /* Exception during execution of
                    sgWheelphoneSendCmdID */
    }
}


void terminateWheelphone(void){
}
```

A.4 *wheelphonelib.tlc*

```
%function FcnGenWheelphoneLib() void
    %assign tgtData     = FEVAL("get_param", CompiledModel.Name,
        "TargetExtensionData")
    %assign packageName = "%<tgtData.packagename>.%<
       CompiledModel.Name>"
    %openfile wheelphoneLibFile = "WheelphoneRobot.java"
    %selectfile wheelphoneLibFile
package %<packageName>;

import android.content.Context;
import android.content.Intent;
import android.os.Handler;
import android.os.Message;


public class WheelphoneRobot {

    // USB communication
    private final static int packetLengthRecv = 63;
    private final static int packetLengthSend = 63;
    private final static int USBAccessoryWhat = 0;
    private static final int APP_CONNECT = (int) 0xFE;
    private static final int APP_DISCONNECT = (int) 0xFF;
    private static final int UPDATE_STATE = 4;
    private boolean deviceAttached = false;
    private boolean isConnected = false;

    private int firmwareVersion = 0;
    private USBAccessoryManager accessoryManager;

    // Packet received from PIC with encoded robot state
    private byte[] encodedRobotState = new byte[packetLengthRecv
       ];

    // Robot control (phone => robot)
    private int lSpeed = 0, rSpeed = 0;
    private static final int MIN_SPEED_RAW = -127;
    private static final int MAX_SPEED_RAW = 127;
    private static final int MIN_SPEED_REAL = -350;
    private static final int MAX_SPEED_REAL = 350;
    private static final double MM_S_TO_BYTE = 2.8;
    private byte flagPhoneToRobot = 1;

    // bit 0 => controller On/Off
    // bit 1 => soft acceleration On/Off
    // bit 2 => obstacle avoidance On/Off
    // bit 3 => cliff avoidance On/Off
    // others bits not used


    private Context context;
    private Intent activityIntent;
```

```
/*
 * Interface that should be implemented by classes that
    would like to
 * be notified by when the WheelphoneRobot state is updated.
 */
public interface WheelphoneRobotListener {
    void onWheelphoneUpdate();
}

private WheelphoneRobotListener mEventListener;



// Handler for receiving messages from the USB Manager
   thread
private Handler handler = new Handler() {


    public void handleMessage(Message msg) {
        byte[] commandPacket1 = new byte[2];
        byte[] commandPacket2 = new byte[packetLengthSend];

        switch (msg.what) {

            case USBAccessoryWhat:

                switch (((USBAccessoryManagerMessage) msg.
                   obj).type) {

                    case CONNECTED:
                        break;

                    case DISCONNECTED:
                        isConnected = false;

                        //Notify listener of a disconnection
                        if(mEventListener!=null) {
                            mEventListener.
                                onWheelphoneUpdate();
                        }
                        break;

                    case READY:

                        String version = ((
                           USBAccessoryManagerMessage) msg.
                           obj).accessory.getVersion();
                        firmwareVersion = getFirmwareVersion
                           (version);

                        switch (firmwareVersion) {
                            case 1:
                                deviceAttached = true;
                                break;
                    case 3:
                        //  send information PIC
                        commandPacket1[0] = (byte)
                           APP\_CONNECT;
                        commandPacket1[1] = 0;
                        accessoryManager.write(
                           commandPacket1);
                        deviceAttached = true;

                        //   Wheelphone state
                        commandPacket2[0] = (byte)
                           UPDATE\_STATE;
                        commandPacket2[1] = (byte)
                           0;
                        commandPacket2[2] = (byte)
                           0;
                        commandPacket2[3] =
                           flagPhoneToRobot;
                        accessoryManager.write(
                           commandPacket2);
                        break;

                    default:
                        break;



                isConnected = true;
                break;

            case READ:
                if (accessoryManager.isConnected()
                    == false) {
                    return;
                }

                while (true) {
                    if (accessoryManager.available()
                        < packetLengthRecv) {
                        break;
                    }

                    accessoryManager.read(
                        encodedRobotState);

                    switch(encodedRobotState[0]) {
                        case UPDATE\_STATE:
                            //Notify listener of an
                               update
                            if(mEventListener!=null)
                                {
                                mEventListener.
                                   onWheelphoneUpdate
                                   ();
                            }
```

```
                                    break;
                                }

                            }

                            break;

                        case ERROR:
                            break;

                        default:
                            break;

                    }
            }

        }

    };


    /**
     * brief Class constructor
     * param a pass the main activity instance (this)
     * return WheelphoneRobot instance
     */
    public WheelphoneRobot(Context c, Intent i) {
        context = c;
        activityIntent = i;
        mEventListener = null;
    }


    public void createUSBCommunication() {
        accessoryManager = new USBAccessoryManager(handler,
            USBAccessoryWhat);
    }


    /**
     * brief To be inserted into the "onResume" function of the
         main activity class.
     * return none
     */
    public void openUSBCommunication() {
        accessoryManager.enable(context, activityIntent);
    }


    /**
     * brief To be inserted into the "onPause" function of the
         main activity class.
     * return none
     */
    public void closeUSBCommunication() {
        accessoryManager.disable(context);

        switch (firmwareVersion) {
            case 2:
            case 3:
                byte[] commandPacket = new byte[2];
                commandPacket[0] = (byte) APP\_DISCONNECT;
                commandPacket[1] = 0;
                accessoryManager.write(commandPacket);
                break;
        }

        try {
            while (accessoryManager.isClosed() == false) {
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        isConnected = false;
    }


    /**
     * brief Set listener to be notified by when the
        WheelphoneRobot state is updated. .
     * return none
     */
    public void setWheelphoneRobotListener(
       WheelphoneRobotListener eventListener) {
        mEventListener = eventListener;
    }


    /**
     * brief Remove listener to be notified by when the
        WheelphoneRobot state is updated. .
     * return none
     */
    public void removeWheelphoneRobotListener() {
        mEventListener = null;
    }


    /**
     * brief Send the next packet to the robot containing the
        last left and right speeds and flag data.
     * Use the speed references set with the setSpeed methods.
     * return none
     */
    public void sendCommandsToRobot() {
        if (accessoryManager.isConnected() == false) {
            return;
        }
        byte[] commandPacket = new byte[packetLengthSend];
        commandPacket[0] = (byte) UPDATE\_STATE;
        commandPacket[1] = (byte) lSpeed;           //  left
            speed ref
        commandPacket[2] = (byte) rSpeed;           //  right
            speed ref
        commandPacket[3] = flagPhoneToRobot;        //  (control
             enable)
        accessoryManager.write(commandPacket);
    }


    /**
     * brief Turn Wheelphone motors off.
     * return none
     */
    public void turnMotorsOff() {
        if (accessoryManager.isConnected() == false) {
            return;
        }
        byte[] commandPacket = new byte[packetLengthSend];
        commandPacket[0] = (byte) UPDATE\_STATE;
        commandPacket[1] = 0;                       //  left
            speed ref
        commandPacket[2] = 0;                       //  right
            speed ref
        commandPacket[3] = flagPhoneToRobot;        //  (control
             enable)
        accessoryManager.write(commandPacket);
    }


    /**
     * brief Retrieve firmware version from USB version
     * param string with USB version
     * return firmware version installed on Wheelphone PIC24F
     */
    private int getFirmwareVersion(String version) {
        String major = "0";
        int positionOfDot;

        positionOfDot = version.indexOf('.');
        if (positionOfDot != -1) {
            major = version.substring(0, positionOfDot);
        }

        return new Integer(major).intValue();
    }


    /**
     * brief Return version of the firmware running on the robot
         . This is useful to know whether an update is available
         or not.
     * return firmware version
     */
    public int getFirmwareVersion() {
        return firmwareVersion;
    }


    /**
     * brief Return the encoded robot state (i.e. packet
        received from onboard PIC)
     * return encoded robot state
     */
    public byte[] getEncodedRobotState() {
        return encodedRobotState;
    }


    /**
     * brief Indicate whether the robot is connected (and
        exchanging packets) with the phone or not.
     * return true (if robot connected), false otherwise
     */
    public boolean isRobotConnected() {
        return isConnected;
    }


    /**
     * brief Set the new left and right speeds for the robot.
        The new data
     * will be actually sent to the robot when "
        sendCommandsToRobot" is
     * called the next time within the timer communication task
        (50 ms cadence).
     * This means that the robot speed will be updated after
     * at most 50 ms (if the task isn't delayed by the system).
     * param l left speed given in mm/s
     * param r right speed given in mm/s
     * return none
     */
    public void setSpeed(int l, int r) {         // speed given
       in mm/s
        if(l < MIN\_SPEED\_REAL) {
            l = MIN\_SPEED\_REAL;
        }
        if(l > MAX\_SPEED\_REAL) {
            l = MAX\_SPEED\_REAL;
        }
        if(r < MIN\_SPEED\_REAL) {
            r = MIN\_SPEED\_REAL;
        }
        if(r > MAX\_SPEED\_REAL) {
            r = MAX\_SPEED\_REAL;
        }
        lSpeed = (int) (l/MM\_S\_TO\_BYTE);
        rSpeed = (int) (r/MM\_S\_TO\_BYTE);
    }
    /**
     * brief Set the new left and right speeds for the robot.
        For more details refer to "setSpeed".
     * param l left speed (range is from -127 to 127)
     * param r right speed (range is from -127 to 127)
     * return none
     */
    public void setRawSpeed(int l, int r) {
        if(l < MIN\_SPEED\_RAW) {
            l = MIN\_SPEED\_RAW;
        }
        if(l > MAX\_SPEED\_RAW) {
            l = MAX\_SPEED\_RAW;
        }
        if(r < MIN\_SPEED\_RAW) {
            r = MIN\_SPEED\_RAW;
        }
        if(r > MAX\_SPEED\_RAW) {
            r = MAX\_SPEED\_RAW;
        }
        lSpeed = l;
        rSpeed = r;
    }



    %closefile wheelphoneLibFile
%endfunction
```

References

[1] S. Bhattacharya, R. Ghrist, and V. Kumar. Multi-robot Coverage and Exploration on Riemannian Manifolds with Boundary. *International Journal of Robotics Research*, 33(1):113–137, January 2014.

[2] F. Bullo, R. Carli, and P. Frasca. Gossip Coverage Control for Robotic Networks: Dynamical Systems on the Space of Partitions. *SIAM Journal on Control and Optimization*, 50(1):419–447, 2012.

[3] J. Choi, S. Oh, and R. Horowitz. Distributed learning and cooperative control for multi-agent systems. *Automatica*, 45(12):2802–2814, 2009.

[4] J. Cortés and F. Bullo. Coordination and geometric optimization via distributed dynamical systems. *SIAM Journal on Control and Optimization*, 44(5):1543–1574, 2005.

[5] J.Cortes,S.Martinez,T.Karatas,andF.Bullo. Coverage control for mobile sensing networks. *Automatica*, 20(2):243–255, 2004.

[6] J. Cortes, S. Martinez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. *Robotics and Automation, IEEE Transactions on*, 20(2):243–255, April 2004.

[7] P. Davison, M. Schwemmer, and N.E. Leonard. Distributed nonuniform coverage with limited scalar measurements. *In Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference* on, pages 1455–1460. IEEE, 2012.

[8] A.A. de Menezes Pereira, H.K. Heidarsson, C. Oberg, D.A. Caron, B.H. Jones, and G.S. Sukhatme. A CommunicationFrameworkforCost-effectiveOperationof AUVs in Coastal Regions. *In The 7th International Conference on Field and Service Robots, Cambridge, Massachusetts*, 2009.

[9] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: applications and algorithms. *SIAM review*, 41(4):637–676, 1999.

[10] J. W. Durham, R. Carli, P. Frasca, and F. Bullo. Discrete Partitioning and Coverage Control for Gossiping Robots. *Robotics, IEEE Transactions on*, 28(2):364– 378, 2012.

[11] S.G. Lee, Y. Diaz-Mercado, and M. Egerstedt. Multirobot Control Using Time-Varying Density Functions. *Robotics, IEEE Transactions on*, 31(2):489–493, April 2015.

[12] N.E. Leonard and A. Olshevsky. Nonuniform coverage control on the line. *In Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pages 753–758. IEEE, 2011.

[13] N.E. Leonard, D.A. Paley, F. Lekien, R. Sepulchre, D.M. Fratantoni, and R.E. Davis. Collective Motion, Sensor Networks, and Ocean Sampling. *Proceedings of the IEEE*, 95(1):48–74, Jan 2007.

[14] S. Lloyd. Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137, Mar 1982.

[15] K.M. Lynch, I.B. Schwartz, P. Yang, and R.A. Freeman. Decentralized environmental modeling by mobile sensor networks. *Robotics, IEEE Transactions on*, 24(3):710–724, 2008.

[16] R. Patel, P. Frasca, J. W. Durham, R. Carli, and F. Bullo. Dynamic Partitioning and Coverage Control with Asynchronous One-To-Base-StationCommunication. *Control of Network Systems, IEEE Transaction on*, 2015. To appear.

[17] M. Schwager, D. Rus, and J-J. Slotine. Decentralized, adaptive coverage control for networked robots. *The International Journal of Robotics Research*, 28(3):357–375, 2009.

[18] M.Schwager,M.P.Vitus,S.Powers,D.Rus,andC.J. Tomlin. Robust adaptive coverage control for robotic sensor networks. *IEEE Transactions on Control of Network Systems*, 2014.

[19] R.C. Shah, S. Roy, S. Jain, and W. Brunette. Data MULEs: modeling a three-tier architecture for sparse sensornetworks. In Sensor Network Protocols and Applications, 2003. *Proceedings of the First IEEE. 2003 IEEE International Workshop on*, pages 30–41, May 2003.

[20] M. Todescato, A. Carron, R. Carli, G. Pillonetto, and L. Schenato. Video of SB algorithm available at
http://automatica.dei.unipd.it/people/todescato/publications.html.

[21] Y. Xu, J. Choi, S. Dass, and T. Maiti. Efficient Bayesian spatial prediction with mobile sensor networks using Gaussian Markov random fields, *Automatica*, 49(12):3520–3530, 2013.

[22] M. Todescato, A. Carron, R. Carli,G. Pillonetto, L. Schenato Multi-Robots Gaussian Estimation and Coverage Control: from Client-Server to Peer-to-Peer Architectures, *Automatica*, 2016

[23] Y. Xu, J. Choi, andS. Oh. Mobile sensor network navigation using Gaussian processes with truncated observations. *IEEE Transactions on Robotics*, 27(6):1118–1131, 2011.

[24] http://www.btsbioengineering.com/it/

[25] K.S.Arun,T.S.Huang,S.D.Blostein, Least-Square Fitting of two 3D Point Set