UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

TESI DI LAUREA

IMPLEMENTATION OF DISTRIBUTED MAPPING ALGORITHMS USING MOBILE WHEELPHONES

LAUREANDO: Andrea Benazzato
RELATORE: Luca Schenato

Corso di Laurea Magistrale in Ingegneria dell' Automazione

ANNO ACCADEMICO 2016 / 2017

Padova, 4 Aprile 2017

Ai miei genitori per la loro infinita pazienza.
Alle mie sorelle Anna e Chiara.
A Matteo, Nicola e Davide per la sincera amicizia.
A Guido e alle sue skills da videomaker.
A Marco che tenta di tenermi a dieta (con scarsi risultati).
Ad Andrea che finalmente mi vedrá lavorare.
A Simone con cui condivido una grande passione (cibo o musica?).
Ad Enrico per le cene salutari.
A me stesso.

Contents

List of Figures

List of Tables

Abstract

In this thesis we consider the problem of multi-robot mapping of an unknown event of interest in an indoor planar region where the communication between the agents is provided by a Wi-Fi network. Various exploration algorithms, most of which based on a nonparametric Gaussian regression approach to estimate an unknown sensory field, will be described and tested in both computer simulation and real-world scenario using various Wheelphone robot devices. A significant attention will also be given to the driver building process which has been done in order to establish a working interface between the Wheelphone device and the MATLAB / Simulink environment.

Chapter 1

INTRODUCTION

SLAM (Simultaneous Localization And Mapping) is the procedure that allows robots to successfully map the environment and, at the same time, localize themselves on that map. This problem has been well studied in the last thirty years because of the impact robotics has in many different fields such as: civil, industrial and military. The concepts of mapping and localization in the robotic framework most of the times appear together because they are strictly connected, in fact, the knowledge of the position of a robot is of primary necessity to build a map, but at the same time the knowledge of the map is useful to improve the localization estimation. In the eighties and nineties the SLAM problem development was quite limited from technological constraints however nowadays, thanks to the appearance of cheap devices capable of good computation, the trend is shifted to the extension of the SLAM problem into a multi-agent framework.
What we want to do in this thesis is to map an unknown sensory function distributed in a planar area using a swarm of mobile robots in order to maximize the likelihood of detecting a specific event. Once the exploration has been done a partitioning and coverage action can be started.

Figure 1.1: Example of a robotics swarm.

This task may have some interesting real applications. Imagine to have robots which can use light for recharging their batteries. After they have explored the environment mapping the light distribution with their sensors they can eventually take position in the optimal place to accomplish recharge. Another example could be the usage of wheeled devices to prevent and detect a blaze. In fact the probability of a fire is proportional to the temperature in a certain location and the agents should cover more the areas where the temperature is higher.
Three mapping algorithms based on a nonparametric estimation in the Gaussian regression framework will be presented and simulated along with a demo with true agents.
The model of the robots used is called ”Wheelphone” and natively it can be programmed utilizing the android language. Even though android and java are widely used, in our case (where we want to test the control of the agent via rapid prototyping in an academic context) we prefer to be able to use MATLAB and Simulink because they are flexible tools well known and used in our department.
This led to the development of some driver interface files which description will be explained.

1.1 Mapping Literature Review

In modern robotics, the mapping task is the creation of a model of a real environment by robots using sensors to acquire topological information from the environment around them. The sensing technology commonly used today are: sonar, cameras, lasers and infrared. The sensors can be used singly or in a combination of them. Before of 2000, the majority of the relevant documents about exploration and mapping were focused on systems composed of a single robot [1, 2]; in the following years, thanks to the better performance and to the low cost of sensors and embedded computing architectures, and under the pressure of competitions and private companies (including military), the documentation about multi-robot systems able to communicate and exchange information, has increased significantly [3].
To solve the problem of map building and exploration over the years have been published different approaches to the problem, each with pros and cons and more suited to certain operating scenarios than others. Some of them are listed below.
Yamauchi’s work is a milestones [4], which show a decentralised approach where each robot shares part of the information while keeping separate personal maps (also introducing the fundamental concept of ”frontier”). With this approach each robot automatically decides the points of interest on the border where to go (frontier based exploration). However, this autonomous navigation method lacks the coordination between robots, this implies the possibility that more robots have the same objective, or that some of them explore areas already explored previously by others, this could cause a loss of efficiency.
Simmons [5] has presented a centralised method in which robots merge the respective personal maps in a common central map, from which a cost minimisation algorithm coordinates the movements of the explorers. Burgard [6] has shown that this coordinated approach has better performance of a non-coordinated one in the task resolution.
Some multi-robot exploration approaches subdivide the surface to be explored thanks to a Voronoi diagram [7] in order to optimally divide the territory to be mapped between the various explor-

ers, this avoids that two robots explore the same surface area and thus this means that we can minimize the cost function time/energy.
Another interesting field which an application is shown in [8] is the mapping and estimation from noisy measurements of a sensory field used to approximate the density function of an event appearance. Most of the classically used techniques for identification are based on parametric estimation (maximum likelihood and prediction error methods) [9, 10] but they require persistent excitation to guarantee convergence of the parameter and the usage a fixed number of basis function requires the collection of at least the same number of samples to output a good estimate. These drawbacks make this methods slow. That's why nonparametric estimation have been developed: the main idea is to exploit black box models to estimate a function from examples-collected on input locations drawn from a fixed probability function [11, 12, 13]. Even if this nonparametric approach in the worst case performs as good as the parametric one, its computational complexity grows unbounded as the cube of collected samples. Efficient solutions based on measurements truncation [14, 15] or Gaussian Markov [16] random fields have been proposed to overcome this problems. More recently the concept of stochastic gradient based on noisy measurements of an unknown sensory function to perform an adaptive deployment of a group of robots have been used [17]. In [18] a Kalman filtering procedure for non parametric estimation is explained. The authors propose a two stage algorithm in which, based on information on the posterior variance, the robot are spread throughout the space in order to achieve a good estimate of the sensory function; once achieved a predefined threshold, the robots are forced to cover the monitored area.

1.2 Contributions

In this work the main focus is to develop some mapping algorithms based on the estimation approach used in [19] and to test them in a real world scenario with true robots.
In particular we'll give attention to:

1. the interface procedure between the robot and the MATLAB/Simulink software. That point is fundamental because we'll be able to design the control and to apply our algorithms in the same flexible and well known environment. This will be precious in the case some one else in the future will expand our work;

2. the algorithms performance in terms of parameters in a real application.

1.3 Outline

The remainder of the thesis is organized as follows:

- Chapter 2 presents the hardware and software tools we used to obtain our results.

- Chapter 3 describes the driver building procedure which has been done for making the communication between the Wheelphone device and the MATLAB / Simulink platform possible.

- Chapter 4 deals with explaining some mathematical notions in order to understand how the devices are controlled and how we can estimate the position of the robots in the area we are trying to map.
- Chapter 5 shows the nonparametric estimation theory and our mapping / estimation algorithms from the pseudo-code to the simulated performances.
- In Chapter 6 a demonstration with true robots with the corresponding results is shown. A description of the motion capture system we used to know the position of the Wheelphones in the arena will also be explained.
- Chapter 7 concludes the thesis and describes directions for future works.

Chapter 2

HARDWARE AND SOFTWARE SETUP

In this section we are going to illustrate and describe some of the main hardware and software resources which has been used in order to achieve our results. In the first place the algorithms have been implemented and tested using a PC running MATLAB: the software can simulate the entire process from the robots movement to the mapping procedure. Afterwards the algorithms have been tested with real two wheeled devices called ”Wheelphone”. The point to point control is executed in real time on the smartphones which works as controller for the robots. The target points for the agents are calculated from a base station thanks to the particular mapping algorithm chosen and then sent to the robots thanks to a Wi-Fi network. In fact all the mobile vehicles and the main server are connected to the network and they can exchange information via UDP communication protocol. The agents knows their exact position in the environment thanks to a motion capture system.

2.1 Wheelphone

The robot is shown if Fig. 2.1 and it’s composed of two main parts: an Android based smartphone and a mobile platform.

1. The **Android phone** is used as a controller for the mobile platform. It communicates with the mobile dock via USB. We can design the control with Simulink and thanks to the Simulink Coder and to the driver we implemented we are able to generate and run in real time on the smartphone the C and C++ code generated from our model.
   The particular model chosen is the ”Samsung Galaxy S III mini” which comes with a small size factor, a 1 GHz dual core processor and 1 GB of RAM.

2. The two wheeled **mobile dock** is made up of a chassis (92 mm width, 102 mm length, 68 mm height, with a weight of 200 g) containing:

   - two DC motors, speed controlled with back EMF; [1]

[1] In motor control and robotics, the term ”Back-EMF” often refers most specifically to actually using the voltage

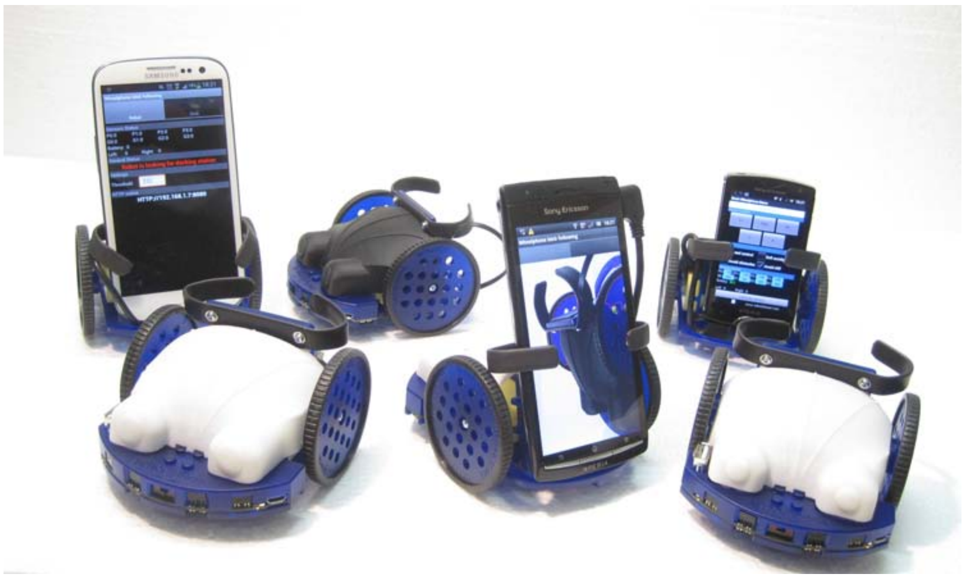


Figure 2.1: Various Wheelphone devices

- four frontal infra-red sensors measuring ambient light and proximity of objects up to 6 cm and other four ground sensors (placed on the bottom-front-side of the robot) detecting cliffs or color differences;
- LiPo rechargeable battery (1600 mAh, 3.7 V). They provide an autonomy of 3.5 hours and they can be recharged in approximately 1.45 hours through a docking station (1000 mA) or a micro USB (500 mA);
- an on board 16 bit microcontroller with a clock speed of 16MHz (8 MIPS) used to execute some predisposed actions such as: speed control, cliff avoidance, smooth acceleration and sensor and odometry calibration;
- an adaptable phone holder with a male micro USB connector in order to make the communication between the phone and the robot possible.

As being said the data transfer between the two components is achieved by USB communication. There are two main data packages: the first one is called *receiving packet* (63 bytes long even if actually only 23 bytes are used) and it is sent from the mobile station to the phone and it contains the information about the robot states (see table 2.1).

generated by a spinning motor to infer the speed of the motor's rotation for use in better controlling the motor in specific ways.

Receiving packet	
BYTE	DESCRIPTION
1	update state
2	frontal proximity sensor 1 value
3	frontal proximity sensor 2 value
4	frontal proximity sensor 3 value
5	frontal proximity sensor 4 value
6	frontal ambient light sensor 1 value
7	frontal ambient light sensor 2 value
8	frontal ambient light sensor 3 value
9	frontal ambient light sensor 4 value
10	ground proximity sensor 1 value
11	ground proximity sensor 2 value
12	ground proximity sensor 3 value
13	ground proximity sensor 4 value
14	ground ambient light sensor 1 value
15	ground ambient light sensor 2 value
16	ground ambient light sensor 3 value
17	ground ambient light sensor 4 value
18	battery level value
19	flagRobotToPhone
20	used to compute the measured velocity generated from the left motor
21	used to compute the measured velocity generated from the left motor
22	used to compute the measured velocity generated from the right motor
23	used to compute the measured velocity generated from the right motor

Table 2.1: Receiving packet (sent by the mobile platform to the phone) description.

The 19th byte is a flag byte called *flagRobotToPhone* and it contains some information about the native functions the onboard PIC is capable of running. If the first bit read of this byte is equal to 1 the obstacle avoidance feature is enabled, if it is equal to 0 the obstacle avoidance feature is disabled; the second bit shows if the cliff avoidance is enabled (1) or not (0). the third, fourth and fifth bits are not used. The sixth bit reports if the robot is charging (1) or not (0) his battery, the seventh if the battery is fully charged (1) or not (0). Finally if the eighth bit read is equal to 1 it means that the odometry calibration is finished, if it is equal to 0 the odometry calibration is still in process.

flagRobotToPhone	
BIT	DESCRIPTION
1	obstacle avoidance enabled?
2	cliff avoidance enabled?
3	not used
4	not used
5	not used
6	is charging?
7	battery charged?
8	odometry finished?

Table 2.2: flagRobotToPhone byte description.

The second data package is called *sending packet* (63 bytes long but only 4 bytes are used) and it is dispatched from the phone to the dock. It is used for setting the wheels speed and other flag bits which enable some prebuilt (see table 2.3) functions.

Sending packet	
BYTE	DESCRIPTION
1	update state
2	left wheel speed (mm/s)
3	right wheel speed (mm/s)
4	flagPhoneToRobot

Table 2.3: Sending packet (sent by the phone to the mobile platform) description.

Like for the reciving packet also in this case we have a flag byte called *flagPhoneToRobot* which can be written for turning on (setting the corresponding bit to 1) or off (setting the corresponding bit to 0) some features:

flagPhoneToRobot	
BIT	DESCRIPTION
1	speed controller
2	soft acceleration
3	obstacle avoidance
4	cliff avoidance
5	calibrate sensors
6	calibrate odometry
7	not used
8	not used

Table 2.4: flagPhoneToRobot byte description.

2.2 Motion Capture System

A motion capture system is a device capable of tracking and recording the movements of an object in a tridimensional space. The one located in our laboratory is an optical motion capture system (Fig. 2.2) which utilizes the data coming from two or more cameras to reconstruct the position of the agent in the 3D space thanks to a triangulation algorithm. Triangulation accuracy is strongly related to the positions and orientations of the cameras. Thus, the configuration of the camera network has a critical impact on performance. The most common approach is based on infrared marker recognition. The markers are small objects capable of reflecting the infrared

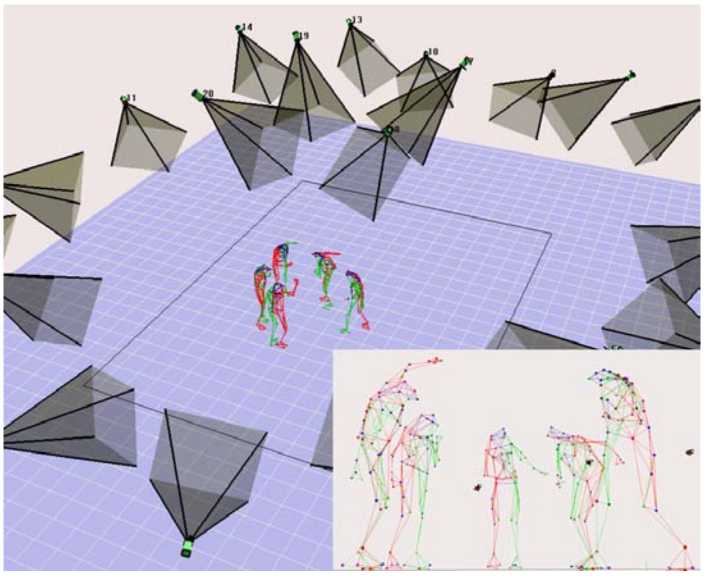


Figure 2.2: An optical motion capture system representation

light emitted from sources near the cameras. The cameras are equipped with an IR-pass filter and then they are capable of capture in the image only the reflecting markers. The utilized markers in our laboratory are shown in Fig. 2.3. The communication between the cameras and the central

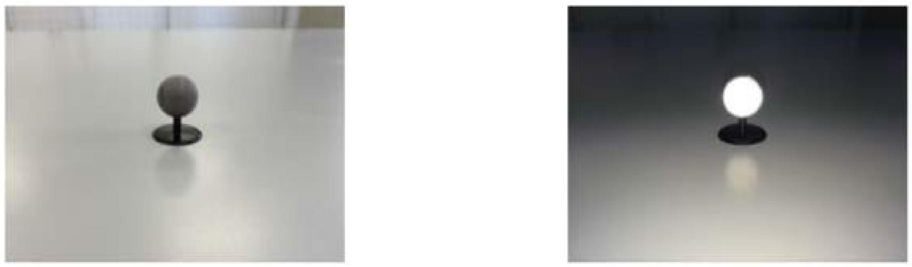
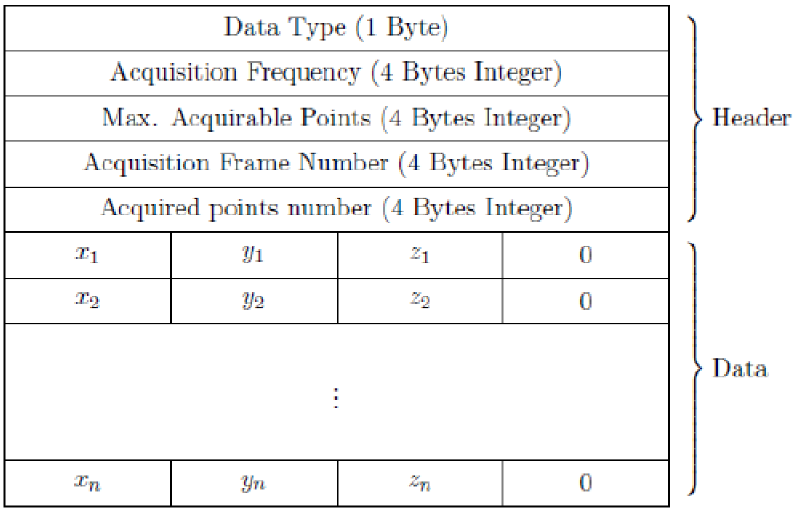


Figure 2.3: the left image shows a marker exposed to the ambient light. To the right we can see a marker exposed to a digital camera flash

unit which elaborates the data and returns the markers position in the 3D plan is based on the following TCP/IP protocol:

1. every camera acquires the markers 2D position in their own image plane and sends this data to the central unit through a TCP packet;
2. the central unit with the information sent by the cameras computes the 3D position of the markers;
3. the x, y and z position of every marker are stored as four-byte values;
4. the elaborated data is sent through an UDP packet wich structure is shown in Fig.2.4.



Data Type (1 Byte)				Header
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	Data
x_2	y_2	z_2	0	
$\vdots$				
x_n	y_n	z_n	0	

Figure 2.4: UDP sending packet.

2.3 MATLAB and Simulink

MATLAB (**mat**rix **lab**oratory) is a proprietary programming language developed by MathWorks.
It allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages. The MATLAB platform is optimized for solving engineering and scientific problems. The MATALB strength is the vast library of prebuilt toolboxes which lets you get started right away with algorithms essential to your domain. All the MATLAB tools and capabilities are all rigorously tested and designed to work together. All the algorithms presented on this thesis have been implemented and simulated on a computer using ”MATLAB 2015b”.

Simulink is a block diagram environment for model design. It supports simulation, automatic code generation, and continuous test and verification of embedded systems. Thanks to a graphical editor you can build and test your own model, eventually customizing and creating your own coded block.
Its great power is that Simulink is integrated with MATLAB. This means that you can incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis.
In this case Simulink has been used to implement a real time application which runs on the smartphone connected to the Wheelphone in order to execute the point to point driving control. This has been possible tanks to the ability of Simulink (through the Simulink Coder and Target Language Compiler tools) to be open to custom code building and custom block design.

Chapter 3

DRIVER DEVELOPMENT PROCESS

Once that the MATLAB / Simulink platform was chosen to control the Wheelphone an effort had to be made to interface the robot with the software. We want our control to be executed from the android phone attached to the mobile dock. This can be made thanks to Simulink and to the Simulink Coder (formerly Real-Time Workshop) which generates and executes C and C++ code from Simulink diagrams, Stateflow charts, and MATLAB functions. The generated source code can be used for real-time and non-real-time applications, including simulation acceleration, rapid prototyping (this is our case: rapid prototyping provides a fast and inexpensive way for control and signal processing engineers to verify designs early and evaluate design tradeoffs), and hardware-in-the-loop testing. In this way we can tune and monitor the generated code using Simulink or run and interact with the code outside MATLAB and Simulink. Thus, after we create a Simulink block for the Wheelphone thanks to an S-function, we must instruct the Simulink Coder on how to build code from this custom block. This can be achieved with the help of the Target Language Compiler (TLC) which provides a great deal of freedom for altering, optimizing, and enhancing the generated code. One of the most important TLC features is that it lets you inline S-functions that you wrote to add your own algorithms, device drivers, and custom blocks to a Simulink model.

3.1 S-function Design

An S-function is a computer lenguage description of a Simulink block written in MATLAB, C, C++ or Fortran.
We decided to write the Wheelphone S-function, called ”sfun_wheelphone.c”, in C code. C written S-functions are complied as MEX files using the command: `mex sfun_wheelphone.c`.
S-functions use a special calling syntax called ”the S-function API” that enables you to interact with the Simulink engine. To understand how the S-function we wrote works we first need to know how the Simulink engine simulates a model.

3.1.1 Simulation Stages

A Simulink block is defined by its set of inputs, set of states and its set of outputs (where the outputs are a function of the simulation time, inputs and of the states).

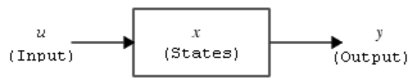


Figure 3.1: Representation of a Simulink block

The mathematical relationships between the inputs, outputs, states, and simulation time are expressed by the following equations:

$$\begin{aligned} y &= f_0(t, x, u) \\ \dot{x} &= f_d(t, x, u) \qquad \text{where } x = \begin{bmatrix} x_c \\ x_d \end{bmatrix} \\ x_{d_{k+1}} &= f_u(t, x_c, x_{d_k}, u) \end{aligned}$$

The Execution of a Simulink model is organized in stages. The first one is the initialization phase In this phase the Simulink engine incorporates library blocks into the model, propagates signal widths, data types, and sample times, evaluates block parameters, determines block execution order and allocates memory. The engine then enters in a simulation loop, where each pass through the loop is referred to as a simulation step. During each simulation step, the engine executes each block in the model in the order determined during initialization. For each block, the engine invokes functions that compute the block states, derivatives, and outputs for the current sample time.
The following figure (Fig. 3.2) illustrates the stages of a simulation. The inner integration loop takes place only if the model contains continuous states.

3.1.2 S-Function Callback Methods

Every S-function must implement a set of methods, called callback methods, which perform the tasks required at each simulation stage when a Simulink model is executed. Tasks performed by S-function callback methods include:

- Initialization.
- Calculation of next sample hit.
- Calculation of outputs in the major time step.
- Update of discrete states in the major time step.
- Integration.

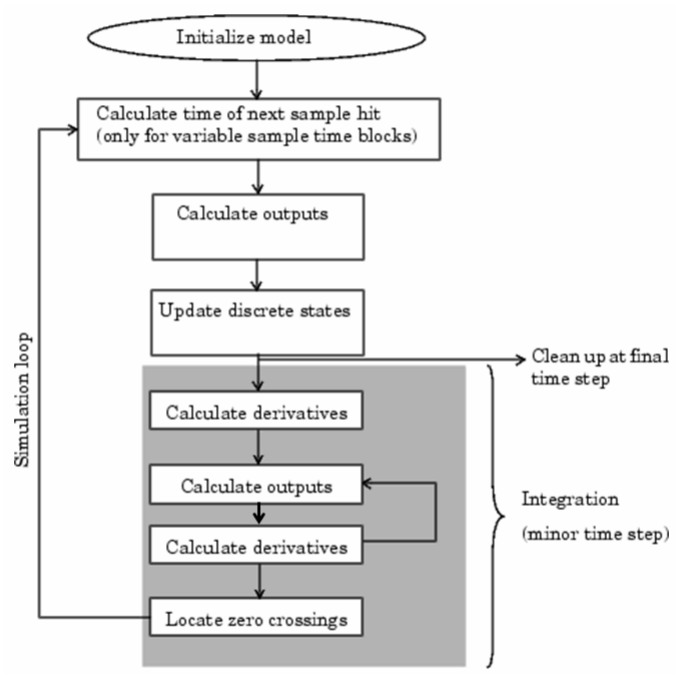


Figure 3.2: How the Simulink engine performs simulation

In a C MEX S-function structure the following callback methods must be all implemented to have a correct execution of a Simulink model:

- `mdlInitializeSizes`: Specifies the sizes of various parameters in the `SimStruct`, such as the number of output ports for the block.
- `mdlInitializeSampleTimes`: Specifies the sample time(s) of the block.
- `mdlOutputs`: Calculates the output of the block.
- `mdlTerminate`: Performs any actions required at the termination of the simulation. If no actions are required, this function can be implemented as a stub.

As we have just saw our S-function ”sfun_wheelphone.c” (for the whole code see Appendix C), after declaring the number of outputs, inputs and their data type, must implement these mandatory methods. Actually most of this methods like `mdlOutputs` and `mdlTerminate` are empty. In fact we don't need to specify how the output will be computed because we won't run the Simulink model from the computer but instead the chart will be coded and executed in real time on the Wheelphone. Therefore the Target Language Compiler (for more information about the Target Language Compile see Appendix A) will instruct the Simulink Coder on how to build the code to compute the outputs for our Wheelphone and how to terminate every instance of the Simulink robot block when we want the simulation to be stopped.

3.2 TLC Files Setup and Design

To generate code from our wheelphone custom block during a simulation we have to inline our S-function. This is done by writing a TLC file called ”sfun_wheelphone.tlc” (Appendix D). Because we want to take advantage of the fact that the Wheelphone manufacturer provides us a list of java library for the robot, it’s better to write a wrapper inlined S-function. This means that we’ll not rewrite inside the TLC file all the java methods to initialize and use our robot but instead we will invoke them from a C file called ”driver_wheelphone.c” (Appendix E). By default, for a wrapped S-function, the methods we want to call from the TLC file ”sfun_wheelphone.tlc” must be declared in a C file. To invoke the true Java methods from an Android running application we have to implement in the right way the methods call in the ”driver_wheelphone.c” file. To do this we must first understand how the robot control android application is generated. The Android application is created by two TLC files called ”srmainandroid.tlc” and ”srmainsamsung_galaxy_s4.tlc”. The file ”srmainandroid.tlc” takes as input some TLC files which provide an android library and others java libraries used to build the control application successfully. The file ”srmainsamsung_galaxy_s4.tlc” makes sure that the generated code is compatible with Samsung Galaxy devices. Note that when the main program is executed two tasks will run at the same time. The first one is related to the Simulink charts: it simulates the Simulink diagram within the Android target environment computing the input and output values for each block. The second one is an application related to the libraries used to make the first task possible: whenever a method is called within the code of the first process the application searches for its declaration in the android library present in the second process. To make sure that the ”driver_wheelphone.c” file, during the first task, can utilize the java methods declared in the java libraries we must first define new methods in the android library in order to invoke the true java methods we need. The only way to call the native java methods for the Wheelphone device through an Android application generated thanks to the Simulink Coder is then to call from ”driver_wheelphone.c” the corresponding methods present in ”srmainandroid.tlc” which in turn, inside its methods body, calls the native methods present inside the true java library (which has been lightened by the unnecessary methods to improve performances) contained in a TLC file called ”wheelphonelib.tlc” (Appendix F). The whole process is illustrated in Fig.3.3.
As we have just saw every block has a target file that determines what code should be generated for the block and the code can vary depending on the exact parameters of the block or the types of connections to it. Within each block target file, **block functions** specify the code to be output for the block in the model’s `start` function, `output` function, `update` function and so on. For our ”sfun_wheelphone.tlc” the first method we wrote is `BlockTypeSetup(block,system)` which doesn’t output code. It executes once per block type before code generation begins. We use it to declare the main methods for our Wheelphone device such as:

- `initWheelphone(void)` which initializes our robot crating a new wheelphone java object and making sure that the robot is communicating with the smartphone via USB;
- `getWheelphoneData(u)` which returns the states of our robot such as the front and ground proximity sensors values, the battery battery charge level, the estimated speed and so on;

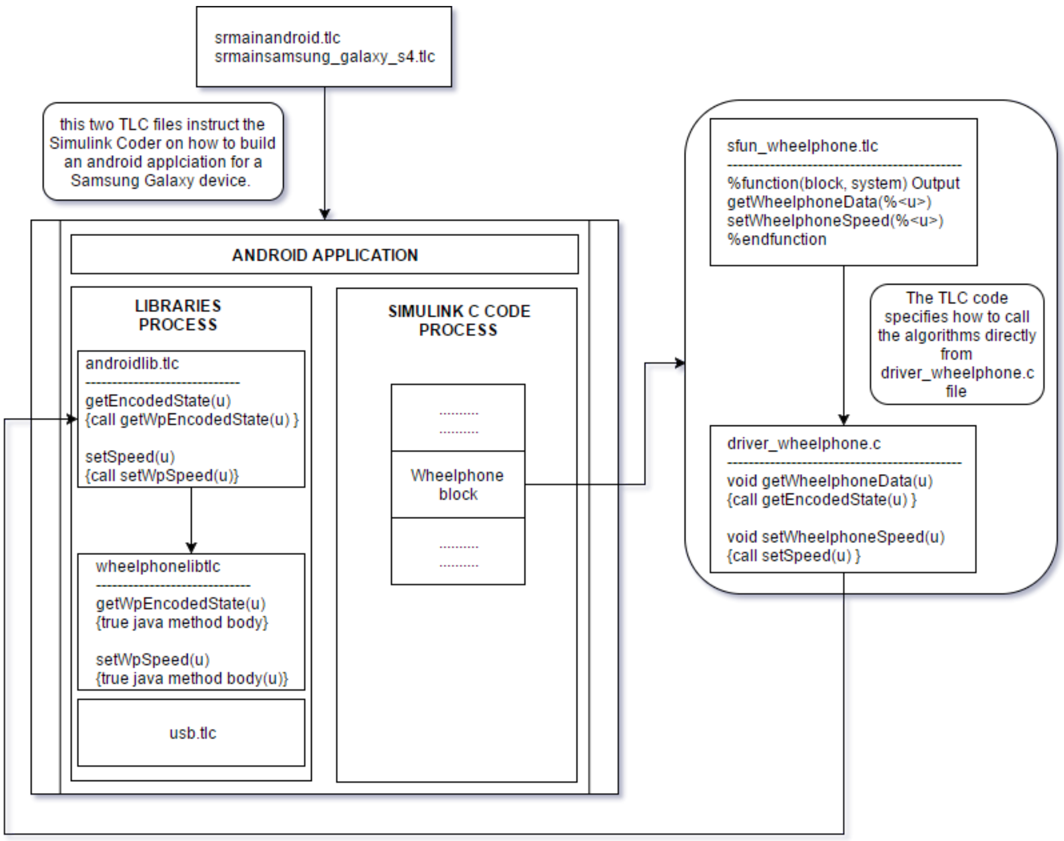

srmainandroid.tlc
srmainsamsung_galaxy_s4.tlc
this two TLC files instruct the Simulink Coder on how to build an android applciation for a Samsung Galaxy device.
ANDROID APPLICATION
LIBRARIES PROCESS
androidlib.tlc
getEncodedState(u)
{call getWpEncodedState(u) }
setSpeed(u)
{call setWpSpeed(u)}
wheelphonelibtlc
getWpEncodedState(u)
{true java method body}
setWpSpeed(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 3.3: The wrapped inlined S-function sfun_wheelphone.tlc calls methods from the C file sfun_wheelphone.c wich in turn calls methods from the native Wheelphone java library. This library is accessible in real time during the execution of the android app thanks to a TLC file called wheelphonelib.tlc

- `setWheelphoneSpeed(l,r)` which set the left and right wheel speed in mm/s.

The second function is `Start(block, system)`. The code inside the `Start` function executes once and only once. We included a Start function to initialize the Wheelphone (calling the method `initWheelphone`) at the beginning of the simulation.
Every block should then include an `Outputs(block, system)` function which of course specifies how to compute the outputs for our S-function. In this case, after defining the input and output variables, we just invoke the methods `getWheelphoneData` and `setWheelphoneSpeed`.

Chapter 4

MATHEMATICAL PRELIMINARIES

4.1 Mobile Robot Kinematics

Mobile Robot Kinematics is the dynamic model of how a mobile robot behaves. In mobile robotics, we need to understand the mechanical behavior of the robot both in order to design and to control a mobile agent hardware. More specifically the Mobile Robot Kinematics is a useful tool when it comes to position and motion estimation. The process of understanding the motions of a robot begins with the process of describing the contribution each wheel provides for motion. Each wheel provides motion but also imposes some constraints such as refusing to skid laterally.

4.1.1 Representing Robot Position

We define a global reference frame $R_0 = (O, X_0, Y_0)$ attached to the plane on which the robot moves and a local reference frame $R_M = (M, X_M, Y_M)$ attached to the robot platform. In order to specify the position of the robot on the plane we establish a relationship between the global reference frame of the plane and the local reference frame of the robot (Fig. 4.1).
The axes X_0 and Y_0 define an arbitrary inertial basis on the plane as the global reference frame from some origin O. To specify the position of the robot, we choose a point M on the robot chassis (often the wheels axis midpoint is chosen) as its position reference point. The basis $\{X_M, Y_M\}$ defines two axes relative to M on the robot chassis and it is thus the robots local reference frame.
The position of M in the global reference frame is specified by coordinates x and y, and the angular difference between the global and local reference frames is given by θ. The pose of the robot with respect to the global reference frame can be specified as a vector:

$$q_0 = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{4.1}$$

The twist (velocity and rotation speed, or posture speed) of the platform with respect to frame R_0 is the time derivative of the posture q $(\dot{q} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{\theta} \end{bmatrix}^T)$. The rotational speed $\dot{\theta}$ is always around the z axis common to all frames, so it does not depend on the frame chosen to express it but the components of the velocity of point M do vary depending on the frame. We will need to express the twist of the robot either in R_0 or in R_M. The orthogonal rotation matrix $R(\theta)$ is used to map motion in the robot local reference frame $\{X_M, Y_M\}$ to motion in the global reference frame $\{X_0, Y_0\}$:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

$$\dot{q}_0 = R(\theta)\dot{q}_M \tag{4.3}$$

Of course we also have that:

$$R(\theta)^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

$$\dot{q}_M = R(\theta)^{-1}\dot{q}_0 \tag{4.5}$$

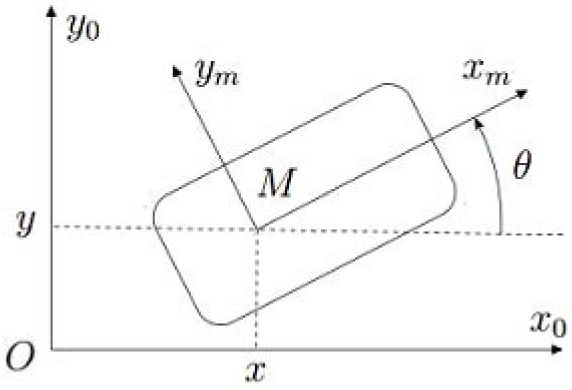


Figure 4.1: The global reference frame and the robot local reference frame

4.1.2 Kinematic Constraints

The motion of a differential-drive mobile robot is limited by two non-holonomic constraint equations, which are obtained by two main assumptions:

- **No lateral slip motion**: in the robot frame this condition means that the velocity of the center-point M is zero along the lateral axis: $\dot{y}_M = 0$. Thus, remembering that $\dot{q}_M = R(\theta)^{-1}\dot{q}_0$, this translates in the global reference frame as:

$$\dot{y}_M = \begin{bmatrix} -\sin\theta & \cos\theta & 0 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = 0 \qquad \Longrightarrow \qquad -\dot{x}\sin\theta + \dot{y}\cos\theta = 0$$

- **Pure rolling constraint**: each wheel maintains a single contact point P with the ground (Fig. 4.2). There is no slipping of the wheel in its longitudinal axis and no skidding in its orthogonal axis so we can write: $\begin{cases} v_r = R\dot{\varphi}_r \\ v_l = R\dot{\varphi}_l \end{cases}$

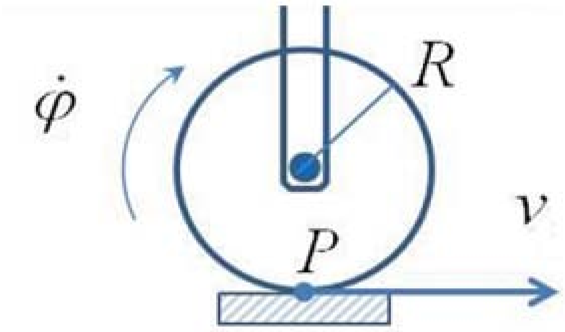


Figure 4.2: Pure Rolling Motion Constraint.

4.1.3 Kinematic Model for Differential Drive Robot

Many mobile robots use a drive mechanism known as **differential drive**. It consists of two drive wheels (each with radius R) mounted on a common axis; each wheel can independently being driven either forward or backward.
The **Forward Kinematics** provides an estimate of the robots position given its geometry and the speeds of its wheels.
Given a point M positioned in the midpoint of the wheels axis, each wheel is at distance L from M. Given R, L, and the spinning speed of each wheel, $\dot{\varphi}_r$ and $\dot{\varphi}_l$, a forward kinematic model can predict the robots overall speed in the global reference frame:

$$\dot{q}_0 = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = f(L, R, \theta, \dot{\varphi}_r, \dot{\varphi}_l) \tag{4.6}$$

Suppose (as shown in Fig. 4.3) that the robots local reference frame is aligned such that the robot moves forward along $+X_M$. First consider the contribution of each wheels spinning speed to the translation speed at M in the direction of $+X_M$. To find the linear velocity in the direction of $+X_M$ each wheel contributes one half of the total speed (if only one wheel is spinning M is halfway between the two wheels and it will move instantaneously with half the speed):
$\dot{x}_{Mr} = \frac{1}{2}R\dot{\varphi}_r$ and $\dot{x}_{Ml} = \frac{1}{2}R\dot{\varphi}_l$.
Consider now a differential robot in which each wheel spins with equal speed but in opposite directions. The angular velocity about θ is again calculated from the contribution of the two

wheels. If only the right wheel is spinning forward it contributes to a counterclockwise rotation along the arc of a circle of radius 2L with a rotation velocity w_r. In the same way If only the left wheel is spinning forward it contributes to a clockwise rotation along the arc of a circle of radius 2L with a rotation velocity w_L. This leads to the two single angular velocity contributes:

$$\begin{aligned} w_r &= \frac{R\dot{\varphi}_r}{2L} \\ w_l &= -\frac{R\dot{\varphi}_l}{2L} \end{aligned} \tag{4.7}$$

Note that the sign minus on the w_l expression is due to the clockwise rotation of the left wheel.

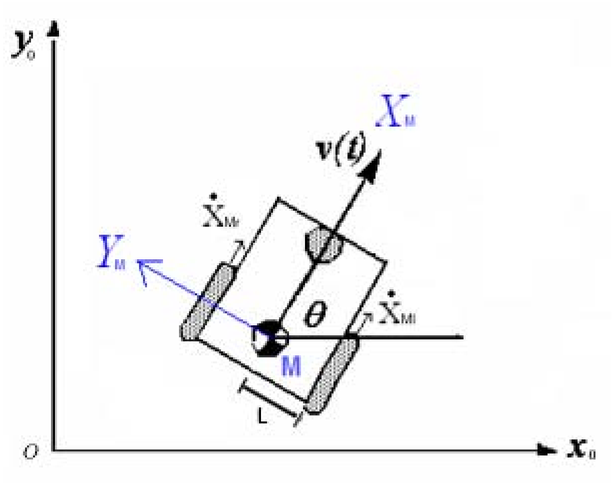


Figure 4.3: A differential drive robot in its global reference frame

The Differential Drive Mobile Robot velocities in the robot frame can now be represented in terms of the center-point M velocities combining these individual contribution expressions:

$$\begin{bmatrix} \dot{x}_M \\ \dot{y}_M \\ \dot{\theta}_M \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ 0 & 0 \\ \frac{R}{2L} & -\frac{R}{2L} \end{bmatrix} \begin{bmatrix} \dot{\varphi}_r \\ \dot{\varphi}_l \end{bmatrix} \tag{4.8}$$

In the global reference frame we have:

$$\dot{q}_0 = R(\theta)\dot{q}_M = R(\theta)\begin{bmatrix} \dot{x}_M \\ \dot{y}_M \\ \dot{\theta}_M \end{bmatrix} = R(\theta)\begin{bmatrix} \frac{R\dot{\varphi}_r}{2} + \frac{R\dot{\varphi}_l}{2} \\ 0 \\ \frac{R\dot{\varphi}_r}{2L} - \frac{R\dot{\varphi}_l}{2L} \end{bmatrix} = \begin{bmatrix} \frac{R(\dot{\varphi}_r+\dot{\varphi}_l)}{2}\cos\theta \\ \frac{R(\dot{\varphi}_r+\dot{\varphi}_l)}{2}\sin\theta \\ \frac{R\dot{\varphi}_r}{2L} - \frac{R\dot{\varphi}_l}{2L} \end{bmatrix} \tag{4.9}$$

4.1.4 Odometry

In a robotic environment the usage of internal sensors to estimate the robot pose is called **odometry**. The classical technique for a wheeled robot to calculate its position is to track its location through a series of measurements of the rotations of the robots wheels. Odometry requires a method for accurately counting the rotation of the robot and, with this information, estimate the left and right wheel velocities. Therefore relative position estimation is extremely dependent on the measurement of the robots velocity. Once we have our left and right spinning velocities we can compute the equation 4.9 and obtain our pose integrating $\dot{x}$ $\dot{y}$ and $\dot{\theta}$:

$$\begin{aligned} x &= \frac{1}{2}\int_0^t [v_r(t) + v_l(t)]\cos(\theta(t))\,dx \\ y &= \frac{1}{2}\int_0^t [v_r(t) + v_l(t)]\sin(\theta(t))\,dx \\ \theta &= \frac{1}{2L}\int_0^t [v_r(t) - v_l(t)]\,dx \end{aligned} \tag{4.10}$$

4.2 Point to Point Control

The algorithms we will see in Chapter 5 return some points in the plane the robots must reach to take a noisy measurement of the sensory function we want to map. To make these movements possible a driving control method must be designed.
Therefore consider the problem of moving toward a goal point (x_r, y_r) in the plane. We will control the robots velocity to be proportional to its distance from the goal:

$$v = K_v \cdot e_d \tag{4.11}$$

where e_d is the distance:

$$e_d = \sqrt{(x_r - x)^2 + (y_r - y)^2}$$

Let's recall that the x and y coordinate of the robot are represented by the position in space of the wheels axis midpoint M.
To steer toward the goal we must compute the angle between the x axis of the global reference frame and the straight line passing through the goal point and the origin of the robot reference frame. This vehicle-relative angle is:

$$\theta_r = \tan^{-1}\left(\frac{y_r - y}{x_r - x}\right)$$

To make the robot face the right direction we impose an angular velocity proportional to the angular error $e_\theta = \theta_r - \theta$:

$$w = K_w \cdot e_\theta \tag{4.12}$$

Note that x, y and θ are estimated thanks to the formulas (4.10) from the robot actual velocity. Since the Wheelphone takes as input the left and right velocities we want to convert this angular velocity into linear velocities. We also want that the robot may be able to turn itself around his wheels axis midpoint M; that's why we choose the left and right linear velocities as $v_l = -v_r$:

$$\begin{aligned} v_l &= -w \cdot L \\ v_r &= w \cdot L \end{aligned} \tag{4.13}$$

Finally, to make the robot moves faster toward the goal point, we sum for each wheel the velocity component calculated from the distance e_d (4.11) and the one calculated from the angular velocity (4.13):

$$\begin{aligned} v_l &= K_v \cdot e_d - K_w \cdot e_\theta \cdot L \\ v_r &= K_v \cdot e_d + K_w \cdot e_\theta \cdot L \end{aligned} \tag{4.14}$$

This control procedure is schematized in Fig.4.4.

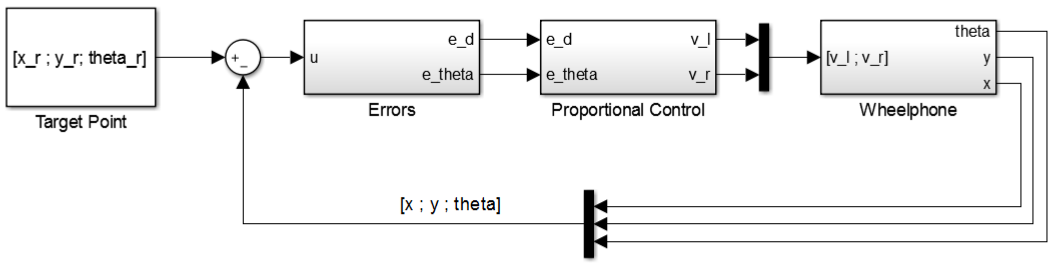


Figure 4.4: Point to point control chart

4.2.1 Point to Point Control Simulink Model With Odometry

The point to point driving control explained in Section 4.2 must be implemented on the robot controller (in this case the mobile android phone connected to the mobile dock). The code generated from the Simulink model illustrated in Fig. 4.5 allows an android app to be run in real time on the smartphone. The robot moves over a planar surface and its configuration is entirely described by the three generalized coordinates $q = (x, y, \theta) \in \mathbb{C}$ with $\mathbb{C} \subset \mathbb{R}^2 \times \mathbb{S}$. In differential drive robotics the θ domain is often chosen as $\mathbb{S} = [-\pi, \pi)$ so the function "angdiff" (which computes the difference between two angles and returns a difference in the interval $[-\pi, \pi)$) is used when we want to calculate e_θ.

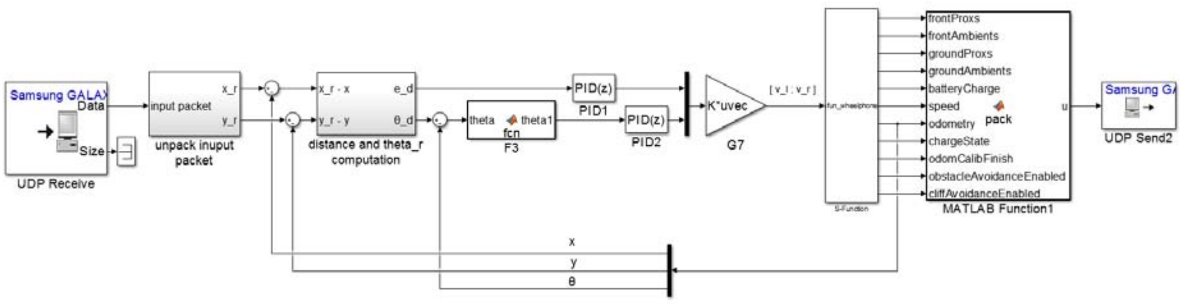


Figure 4.5: Simulink model of the point to point control

Due to the fact that the wheels of the Wheelphone robot are not equipped with true encoders the actual velocity estimation of the device is not accurate. If we consider also the fact we need to integrate this velocity in order to gain access to the robot pose (4.14) the result is that the estimated position of the wheelphone over time becomes unreliable and the control becomes also inefficient. To have a good pose estimation during the real implementation of the mapping algorithms we will use a motion capture system.

Chapter 5

MAPPING ALGORITHMS

In this section we present three algorithms based on nonparametric estimation.

5.1 Nonparametric Estimation

Let $f\colon \mathcal{A} \longrightarrow \mathbb{R}$ denote an unknown deterministic function defined on the compact set $\mathcal{A} \subset \mathbb{R}^n$. Assume to have a set of $S \in \mathbb{N}$ noisy measurements taken on the input locations a_i, i.e. $\{a_i, y_i\}_{i=1}^{S}$, of the form:

$$y_i = f(a_i) + v_i, \qquad i = 1, \ldots\ldots, S \tag{5.1}$$

where v_i is a zero-mean Gaussian noise with variance σ^2, i.e. $v_i \sim \mathcal{N}(0,\, \sigma^2)$, independent of the unknown function. Given the data set $\{a_i, y_i\}_{i=1}^{S}$, one of the most used approaches to estimate f relies upon the Tikhonov regularization theory (see Appendix B) [20].
Let's suppose that f is a zero-mean Gaussian field with covariance, also called kernel K:

$$K\colon \mathcal{A} \times \mathcal{A} \longrightarrow \mathbb{R}.$$

Defining the set containing all the y_i measurements taken at the a_i input location as the information set I:

$$I = \{(a_i, y_i) \quad | \quad i \in \{1, \ldots\ldots, S\}\}$$

and exploiting known results on estimation of Gaussian random fields, one obtains that the minimum variance estimate of f given I is a linear combination of the kernel sections $K(a_i, \cdot)$:

$$\hat{f}(x) = \mathbf{E}[f(a)|I] = \sum_{i=1}^{S} c_i K(a_i, a), \qquad x \in \mathcal{A} \tag{5.2}$$

where the expansion coefficients c_i are obtained as:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_S \end{bmatrix} = (\bar{K} + \sigma^2 \mathbb{I})^{-1} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_S \end{bmatrix} \tag{5.3}$$

while the matrix $\bar{K}$ is obtained by evaluating the kernel at the input-locations, i.e.

$$\bar{K} = \begin{bmatrix} K(a_1, a_1) & \cdots & K(a_1, a_S) \\ \vdots & \ddots & \vdots \\ K(a_S, a_1) & \cdots & K(a_S, a_S) \end{bmatrix}. \tag{5.4}$$

Moreover, the a posteriori variance of the estimate, in a generic location of the plane $a \in \mathcal{A}$ is:

$$V(a) = Var[f(a)|I] =$$

$$= K(a, a) - \begin{bmatrix} K(a_1, a) & \cdots & K(a_S, a) \end{bmatrix} (\bar{K} + \sigma^2 \mathbb{I})^{-1} \begin{bmatrix} K(a_1, a) \\ \vdots \\ K(a_S, a) \end{bmatrix} \tag{5.5}$$

The main problem with this approach is that we need to store more and more input-locations and noisy measurements as time increases to compute the estimate. Indeed, the most expensive operation is the inversion of the $S \times S$ matrix $(\bar{K} + \sigma^2 \mathbb{I})$ which requires $O(S^3)$ operations. This will eventually lead to a memory and computational consuming process which grows unbounded with S.

5.2 Problem Description

We consider a swarm of N robots allowed to move in a planar region represented by the convex set $\mathcal{X}$. The goal is to design algorithms to map and estimate an unknown sensory function $\mu \colon \mathcal{X} \longrightarrow \mathbb{R}$. We assume to take μ as a realization of a Gaussian random field with covariance $K \colon \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$.
We will consider the Gaussian kernel:

$$K(x, x') = \lambda e^{-\frac{\|x - x'\|^2}{2\xi^2}}, \quad \lambda = 1, \quad \xi = 0.2. \tag{5.6}$$

We also assume, for a computer simulation purpose, that every agent $i \in \{1, \dots, N\}$ can:

- take noisy measurements of μ in the form:

  $$y(x_i) = \mu(x_i) + v_i, \tag{5.7}$$

  where $v_i \sim \mathcal{N}(0, \sigma^2)$ is independent from μ and from all the noise measurements v_j;

- communicate with a server/base station or to closely located agents;
- move to a certain target-point b_i respecting the following update law:
  $x_{i,k+1} = x_{i,k} + u_{i,k}, \quad \forall i \in \{1, \dots, N\}$,
  which says that every robot can move from location $x_{i,k}$ at time $t = kT$ to any desired location $x_{i,k+1} = b_i$ at time $t = (k+1)T$.

To make simulations faster and suitable for for a numerical implementation we decided to discretize the working area $\mathcal{X}$. The idea is to constrain the robots to collect measurements only from a set of predetermined finite number of input locations which are obtained thanks to a spatial discretization of the continuous convex domain $\mathcal{X}$:

Definition 5.1. Consider the finite set of m input locations $\mathcal{X}_{grid} = \{x_{grid,1}, \dots, x_{grid,m}\} \subset \mathcal{X}$ where $\mathcal{X} \subset \mathbb{R}^2$ is a convex closed polygon. Given the scalar $\Delta > 0$, we say that the set $\mathcal{X}_{grid}$ forms a **sampled space** of resolution Δ if:

$$\min_{i=1,\dots,m} \|x_{grid,i} - x\| \leq \Delta \quad \forall x \in \mathcal{X} \tag{5.8}$$

5.3 Server-Based Algorithms

The following algorithms are based on a client-server communication architecture where the robots are allowed to communicate with a central server which can:

- store all the measurements taken by the robots;
- store the last position of the agents within the the convex set $\mathcal{X}$ in a matrix called "Robot_positions" where in each row "i" is stored: in position $(i,1)$ the robot id number, in position $(i,2)$ the x robot coordinate and in position $(i,3)$ the y robot coordinate;
- compute the target points to visit and send information to the robots every T seconds;
- compute and store an estimate $\hat{\mu}$ of the function μ and its posterior variance V.

We also suppose that every robot $i \in \{1, \dots, N\}$ can take only one measurement in the from (5.7) within the time window $t \in (kT, (K+1)T)$, $k \in \mathbb{N}$. Once the measurement has been collected it is immediately transmitted to the server which stores it in memory. Then the server computes,depending on the algorithm used, the next target locations for each robot. By defining with J_k the set of measurements received by the server at iteration k, i.e.

$$J_k := \{(x_{i,k}, y_{i,k}) | i = 1, \dots, N\}, \quad x_{i,k} \in \mathcal{X}_{grid} \subset \mathcal{X}$$

the total information set I_k available at server at iteration k can be computed as:

$$I_k = I_{k-1} \cup J_k, \quad \forall k \geq 1, \quad I_0 \neq \emptyset.$$

Thanks to the information set I_k the server can store in his memory the estimate $\hat{\mu}_k(x)$ of the function $\mu(x)$ and its posterior variance $V_k(x)$ computed according to equations (5.2) and (5.5).

In Algorithm 1 is shown the whole Server-Based communication process where the procedure have been divided in two parts. The first describes the operations executed by the server, the second those executed by the robots.
We developed three algorithms in order to calculate in different ways the target points for the agents which are:

1. a randomized selection algorithm;
2. a minimum distance algorithm based on the maximum posterior variance;
3. a distance weighted algorithm based on the posterior variance.

The algorithms return a matrix called ”Target_points” where for each row ”i” is stored: in position $(i, 1)$ the number id of the robot we want to send the target point to, in position $(i, 2)$ the x target point coordinate and in position $(i, 3)$ the y target point coordinate.
All these algorithms will be explained in the next subsections.

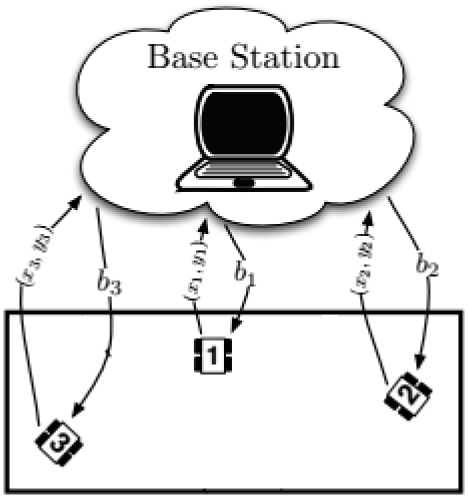


Figure 5.1: The client-server communication architecture considered: the robots can send their input locations to the server/base station while the server, after performing all the computation, can send to the robots the target points.

Algorithm 1 Server-Based Communication

```
1: SERVER
2: if $k = nT, \quad n \in \mathbf{N}$ then
3:     Listen (input location reception)
4:     $J_k = \emptyset$
5:     for $i = 1, \dots, N$ do
6:         $J_k = J_k \cup \{x_{i,k}, y_{i,k}\}$
7:     end for
8:     $I_k = I_{k-1} \cup J_k$

9:     Estimate Update:
10:    $\hat{\mu}(x) = \mathbf{E}[\mu(x)|I_k]$
11:    $V_k(x) = Var[\mu(x)|I_k]$

12:    Target Points Computation:
13:    the target points $b_{i,k}$ are computed differently depending on the mapping algorithm chosen

14:    Target Points Transmission
15:    $x_{i,k} = b_{i,k}, \quad \forall i$
16: end if

17: ROBOTS
18: if $k = nT, \quad n \in \mathbf{N}$ then
19:    Measurements Collection
20:    $y_{i,k} = \mu(x_{i,k}) + v_{i,k}$

21:    Measurements Transmission
22:    $(x_{i,k}, y_{i,k}) \longrightarrow Server$

23:    Listen (target points reception)
24:    $x_{i,k} = b_{i,k}$

25:    Move To The New Target Point
26: end if
```

5.3.1 Randomized Selection Algorithm

The algorithm `randomalg`$(\mathcal{X}_{grid}, N)$ takes as inputs a list containing all the possible grid points $x_{grid,i} \in \mathcal{X}_{grid} \subset \mathcal{X}$ and the number N of robots we are using.
The main idea is easy: for each robot the target point is chosen randomly among the list of all possible input locations. Every time a grid point is selected it is also deleted from the list of the input points. This is done in order to prevent two robot to move towards the same target point. The pseudocode is shown in Algorithm 5.

Algorithm 2 Randomized Selection Algorithm

```
 1: randomalg(list,N)
 2: %for each robot
 3: for $i = 1, \dots, N$ do
 4:     $b_{i,k} = (x_{rand}, y_{rand}) \in$ list          %the point is chosen randomly
 5:     Target_points(i,1) = i                  %robot id
 6:     Target_points(i,2) = $x_{rand}$              %target point random x coordinate
 7:     Target_points(i,3) = $y_{rand}$              %target point random y coordinate
 8:     delete $b_{i,k}$ from the list
 9: end for
10: return Target_points
11: end randomalg
```

5.3.2 Minimum Distance Algorithm

The algorithm `mindistalg(Robot_positions,V_x_tab,N)` takes as inputs a list of the robots positions, a list called V_x_tab and the number N of robots we are using.
V_x_tab is a matrix which has a number of rows equal to the number of grid points in $\mathcal{X}_{grid}$. The information about each grid point $x_{grid,i}$ is saved on the "i-th" row of V_x_tab. In fact in each "i-th" row of V_x_tab is stored: in position $(i, 1)$ the x coordinate of $x_{grid,i}$, in position $(i, 2)$ the y coordinate of $x_{grid,i}$ and in position $(i, 3)$ the $x_{grid,i}$ point posterior variance computed thanks to the previous measurements.
This algorithm detects and saves in a list called "highest_var_points" N target points among V_x_tab with the highest posterior variance (for each "i-th" row of highest_var_points is stored: in position $(i, 1)$ and $(i, 2)$ the x and y coordinates of the "i-th" grid point with the highest posterior variance and in position $(i, 3)$ its posterior variance value). Then `mindistalg` computes the distances between these points and each robot. For every robot the target point is chosen thanks to a minimum distance criteria: every robot will move to the closest target point. Every time a target point is assigned it is also deleted from the list "highest_var_points" in order to avoid multiple assignments. The pseudocode is shown in Algorithm 3.

Algorithm 3 Minimum Distance Algorithm

```
1:  mindistalg(Robot_positions,V_x_tab,N)
2:  for $i = 1, \dots, N$ do
3:      save $N$ highest posterior variance points $\{x_{M1}, \dots, x_{MN}\} \in$ V_x_tab in highest_var_points
4:  end for
5:  %for each robot
6:  for $i = 1, \dots, N$ do
7:      compute the distances between the i-th robot and all the grid points saved in
        highest_var_points
8:      $b_{i,k} = (x_j, y_j)$ is the closest point $x_{Mj}$ saved in highest_var_points to the i-th robot
9:      Target_points(i,1) = i                %robot id
10:     Target_points(i,2) = $x_j$            %target point x coordinate
11:     Target_points(i,3) = $y_j$            %target point y coordinate
12:     delete $b_{i,k}$ from highest_var_points
13: end for
14: return Target_points
15: end mindistalg
```

5.3.3 Distance Weighted Algorithm

The main idea behind this algorithm is to move the agents to the target points which have the highest posterior variance - distance ratio:

$$r_{i,j} = \frac{Var[f(x_{grid,j})|I]}{d_{ij}}; \tag{5.9}$$

where d_{ij} is the Euclidean distance between the robot "i" and the point $x_{grid,j}$.
In a real case scenario this can save fuel or robot energy: every agent will in fact prefer to move towards a target point which has a good ratio between the posterior variance (or lack of information) and its distance from the robot instead of moving to a point which have the highest posterior variance but it is also really far away from our agent.
As the minimum distance algorithm, `weightedtalg(Robot_positions,V_x_tab,N)` takes as input a list of the robots positions, V_x_tab and the number N of robots we are using.
This algorithm thanks to V_x_tab computes for every robot and for every grid point $x_{grid,j}$, $j \in \{1, \dots, M\}$, the ratio $r_{i,j}$ (eq. 5.9). Then each robot picks the target point with the highest ratio $r_{i,j}$ and again, in order to avoid multiple point assignments, the target point chosen is deleted from V_x_tab. The pseudocode is shown in Algorithm 4.

Algorithm 4 Distance Weighted Algorithm

```
1: weightedtalg(Robot_positions,V_x_tab,N)
2: %for each robot
3: for $i = 1, \dots, N$ do
4:     %and for each grid poin
5:     for $j = 1, \dots, M$ do
6:         compute and save the ratio $r_{i,j} = \frac{Var[f(x_{grid,j})|I]}{V\_x\_tab(j,3)}$
7:     end for
8:     $b_{i,k} = (x_j, y_j)$ is the point with the highest ratio $r_{i,j}$
9:     Target_points(i,1) = i            %robot id
10:    Target_points(i,2) = $x_j$         %target point x coordinate
11:    Target_points(i,3) = $y_j$         %target point y coordinate
12:    delete $b_{j,k}$ from highest_var_points
13: end for
14: return Target_points
15: end weightedtalg
```

5.4 Server-Based Algorithms With Non-Growing Information Set

The non-parametric estimation used by our algorithms and explained in Section 5.1 suffers from computational problems caused by dimensionality. Due to the matrix inversion $(\bar{K} + \sigma^2 \mathbb{I})^{-1}$ the computational cost at iteration k is $\mathcal{O}(k^3 m_k^3)$ which grows cubically with the number of measurements m_k. In fact $\bar{K}$ is a $m_k \times m_k$ matrix (m_k is the number of measurements taken by the robots up to iteration k) and the computational complexity for the inversion of an $n \times n$ matrix is approximately $\mathcal{O}(n^3)$. Thus, as presented in [19], we want to find a way to make our algorithms lighter and faster.

Since the grid is composed by a finite number of locations, new measurements can fall exactly over the same input location. Let $n \leq m$ be the number of distinct input locations on the grid visited at least once up to instant k, denoted by $x_{grid,1}, \dots, x_{grid,n}$ (if $n = m$ they coincide with $\mathcal{X}_{grid}$). Let also y_i be the l_i-th measurement taken in the input location $x_i \in \mathcal{X}_{grid}$ and define the virtual measurement w_{i,l_i} as the average of the first l_i measurements associated to the input location x_i. To avoid storing all the measurements up to y_i, it is possible to compute w_{i,l_i} in a recursive way exploiting only the current measurement y_i and the previous virtual measurements w_{i,l_i-1}.

In particular, one has:

$$w_{i,l_i} = \frac{l_i - 1}{l_i} w_{i,l_i-1} + \frac{1}{l_i} y_i \tag{5.10}$$

Accordingly, we define the variance associated to the virtual measurement w_{i,l_i} as $\sigma_i^2 = \frac{\sigma^2}{l_i}$ and with Σ a diagonal matrix collecting these noise variances, i.e $\Sigma = \text{diag}\left(\{\sigma_i^2\}_{i=1}^n\right)$.

Once we collect two or more measurements in the same input location, the size of the matrix

$\bar{K}$ does not vary and it increases only if an input location $x_{grid,i} \in \mathcal{X}$ is visited for the first time. When all the grid points have been visited at least one time the sampled Kernel $\bar{K}$ does not change size anymore.
Instead, every time a new measurement is achieved, one has to update the variance matrix Σ and the vector with the virtual measurements $w = [w_{1,l_1}, \dots, w_{m,l_n}]$ with $n \leq m$. After having updated these elements we can compute as usual the function estimate and the posterior variance using equations (5.2) and (5.5).

5.5 Simulations

In this Section we provide some simulations showing the performances of our algorithms. We run all the simulations in MATLAB on a desktop computer with a processor Intel Core i7-4670k and 16Gb of RAM. In order to test the algorithms in a context similar to the arena we use in our laboratory in our simulations we use a team of $N = 5$ robots spread in a square domain $\mathcal{X} = [-1, 1] \times [-1, 1]$ and the kernel used is the Gaussian kernel (5.6).
The unknown sensory function μ is a combination of two bi-dimensional Gaussians:

$$\mu(x) = 5 \sum_{i=1}^{2} e^{\frac{\|x-\mu_i\|^2}{0.01}}, \quad \mu_1 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \quad \mu_2 = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}; \tag{5.11}$$

which can be seen in Fig.5.2.

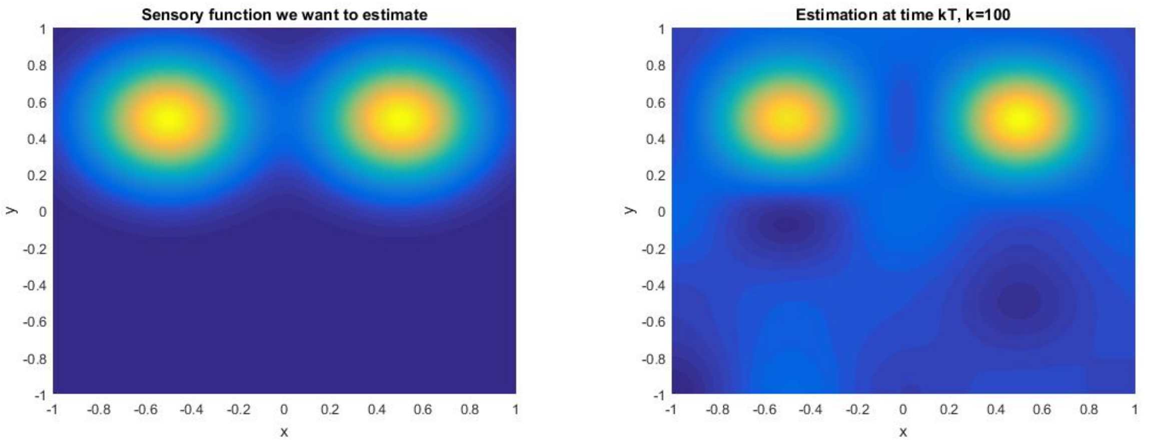


Figure 5.2: Representation of $\mu(x)$ on the domain $\mathcal{X}$ and an example of estimation $\hat{\mu}(x)$ after $k = 100$ iteration using the Distance Weighted Algorithm.

5.5.1 Comparsion Between Sever Based Algorithms

Here we compare our Randomized Selection Algorithm (RSA), Minimum Distance Algorithm (MDA) and Distance Weighted Algorithm (WDA) algorithms. In order to determine the quality of the algorithms the a posterior variance of the points of the set $\mathcal{X}_{grid}$ and of the set $\mathcal{X}$ will be analyzed. For computational reasons when we compute the a posterior variance of a point $x \in \mathcal{X}$ we consider a grid of 30 equidistant points per side (then the total number of points is $p^2 = 900$).

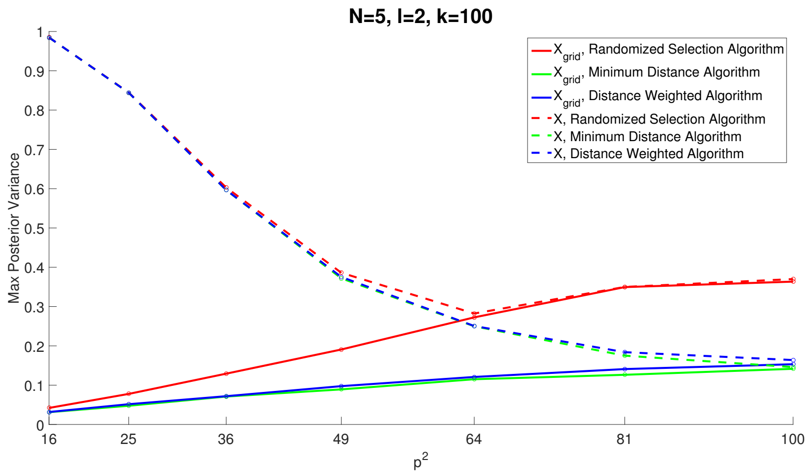


Figure 5.3: Comparsion between RSA, MDA and DWA for different total numbers of grid points p_g^2: evolution of the max of the posterior variance averaged over 100 indipendent simulations.

In Fig.5.3 is shown the evolution of the posterior variance for different values of the grid points p_g^2 after a number of iterations $k = 100$. The values of the number of robots used and of the side of the squared map are fixed to $N = 5$ and $l = 2$ ($\mathcal{X} = [-1, 1] \times [-1, 1]$). As we can see the worst algorithm is the Randomized Selection Algorithm due to its lack of mapping strategy which performs way worst than the Minimum Distance Algorithm and the Distance Weighted Algorithm. The best algorithm is the Minimum Distance Algorithm: every robot in fact will explore the points with the highest posterior variance. This helps, in a long run, to keep the average posterior variance as low as possible. The Distance Weighted Algorithm performs slightly worst than MDA but it can potentially save a lot of energy/fuel thanks to its conservative strategy. For every mapping algorithm the maximum posterior variance found in the set $\mathcal{X}$ converges to the one found within the set $\mathcal{X}_{grid}$ when the value of p_g^2 increases.

In Fig. 5.4 is again shown the evolution of the posterior variance (from 0 to 100 iterations) but this time also the number of grid points is fixed to $p_g^2 = 25$. Even if the maximum posterior variance values found over time for a generic point $x_i \in \mathcal{X}$ are in general high, the maximum posterior variance values for our grid points $x_{grid,i} \in \mathcal{X}_{grid}$ after 100 iterations are really low (under the 0.1 threshold). This means that for our grid points we have enough information to estimate a value which is really close to the one computed by evaluating the sensory function in these points and if the number of grid points is reasonable this provides a good estimation of our sensory function (see Fig.5.2).

Note that the curves of Figure 5.4 scale with the number of robots. That is increasing the number of robots, the max of the posterior variance tends to zero more rapidly. In particular the curves scale as $\frac{1}{\sqrt{N}}$. Again, the RSA is the worst of the three algorithms, the MDA is the best and the DWA preforms similar to the MDA. As we can see from this figure for a low number of iterations

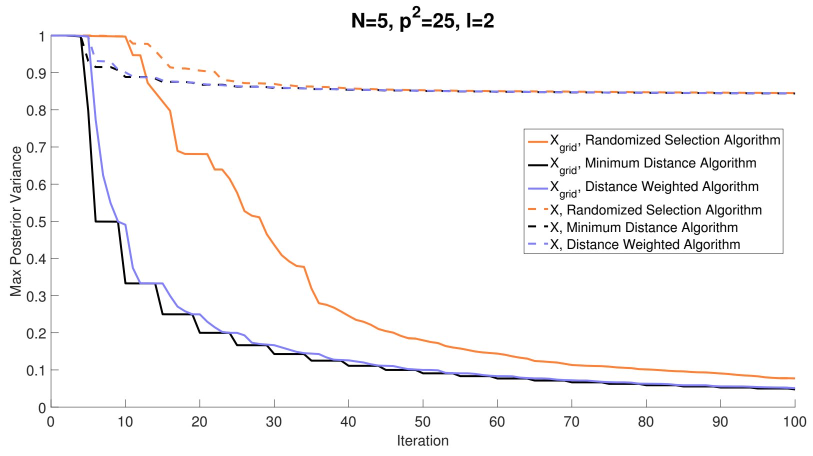


Figure 5.4: Comparsion between RSA, MDA and DWA: evolution of the max of the posterior variance from 0 to 100 iterations averaged over 100 indipendent simulations.

we want to avoid the Randomized Selection Algorithm, but for number of iteration higher than 100 it may be preferable if we don't have a good computational power since it is light and easy to implement and it provides a good maximum posterior variance for the grid points. The Distance Weighted Algorithm is instead preferable if we want to save energy resources like in an emergency or in a military scenario. Lastly if we don't have any limitations, the Minimum Distance Algorithm is surely the best one.
Similar considerations can be made observing Fig. 5.5 where the maximum posterior for some $x_i \in \mathcal{X}$ and for some $x_{grid,i} \in \mathcal{X}_{grid}$ is computed for different values of N and with a fixed number of grid points $p_g^2 = 25$ and $l = 2$.

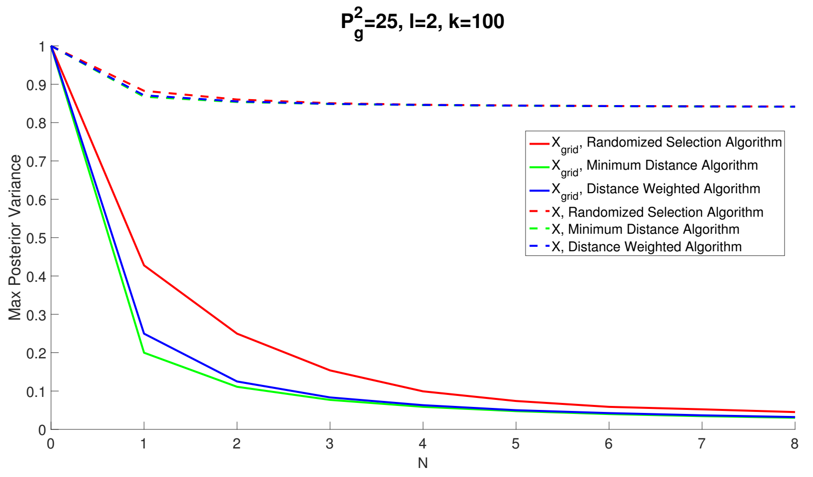


Figure 5.5: Comparsion between RSA, MDA and DWA for different total numbers of robots N: evolution of the max of the posterior variance averaged over 100 indipendent simulations.

5.5.2 Comparsion Between Sever Based Algorithms With Non-Growing Information Set

In this Section we compare our RSA, MDA and WDA algorithms using an Information set computed recursively as shown in Section 5.4. In particular we are going to compare the execution time of our algorithms with a non growing information set and the same algorithms with a normal stacked information set. In Fig. 5.6 is shown the execution time in seconds of our algorithms for a number of iteration $k \in [0, 100]$. As we can see the usage of a non growing information

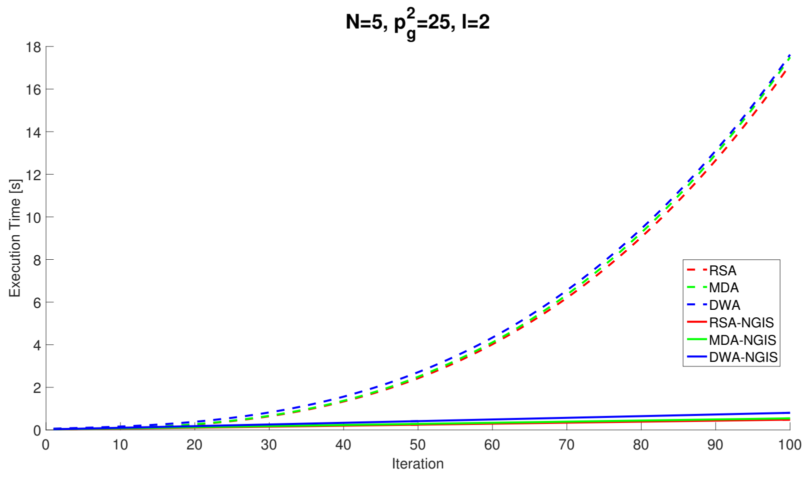


Figure 5.6: Comparsion between RSA, MDA, DWA and the same version of these algorithms using a non-growing information set: evolution of the execution time averaged over 100 indipendent simulations.

set makes the evolution of the execution time for a varying value of iterations k almost linear compared to the evolution of the execution time of our algorithms with a "normal" information set which has an exponential behavior. Therefore for a real time application the usage of a non growing information set (NGIS) is strongly recommended.
This last statement is reinforced by what we can see in Fig. 5.7 where the evolution of the maximum posterior variance is printed for different iterations values and for both versions of the algorithms. This image proves that the NGIS version of our algorithms performs similar to the normal version of the same algorithms but, using a non growing information set, we have a much faster execution time.
This faster execution time allows us to test our algorithms for a higher value of iterations and grid points. In Fig. 5.8 is again shown the evolution of the posterior variance (from 0 to 100 iterations) but with a number of grid points fixed to $p_g^2 = 64$. From this picture we can observe that the maximum posterior found for a generic $x_i \in \mathcal{X}$ is decreasing when also the number of iteration is increasing. This means that with a more dense grid and with an high number of iterations we will have a more reliable knowledge of our sensory function for a point x_i outside the grid set $\mathcal{X}_{grid}$.

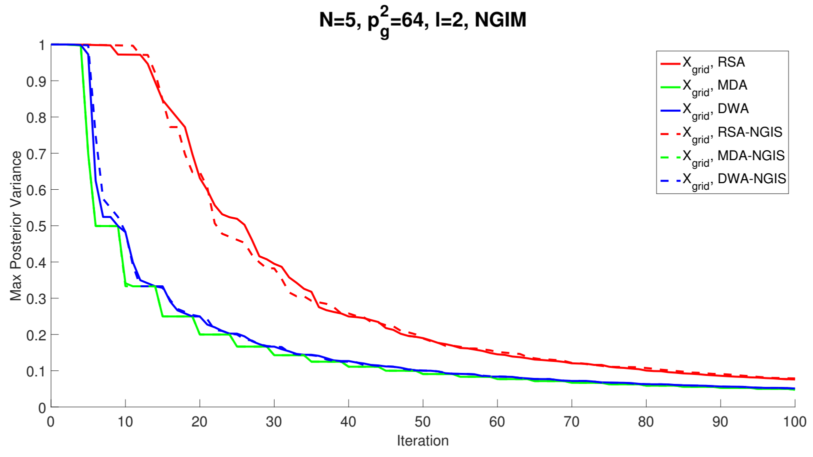


Figure 5.7: Comparsion between RSA, MDA, DWA and the same version of these algorithms using a non-growing information set: evolution of the max of the posterior variance from 0 to 100 iterations averaged over 100 indipendent simulations.

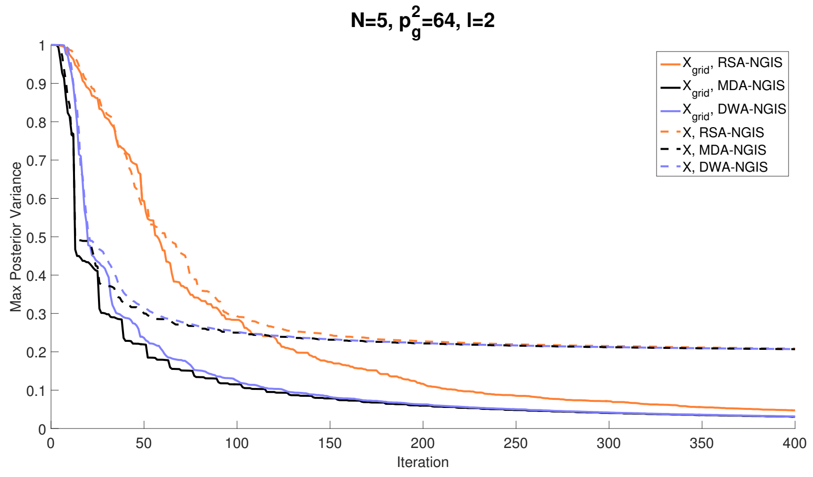


Figure 5.8: Comparsion between RSA-NGIS, MDA-NGIS and DWA-NGIS: evolution of the max of the posterior variance for both the sets $\mathcal{X}$ and $\mathcal{X}_{grid}$ from 0 to 400 iterations averaged over 100 indipendent simulations.

Chapter 6

WHEELPHONE IMPLEMENTATION

To implement our algorithms using several Wheelphone robots we need to have a precise position value of the robot within the map. The odometry we found in Section 4.1.4 is unsuccessful in this sense for many reasons. First of all the Wheelphone does not have a true wheel encoder but it estimates the wheels linear velocity thanks to the voltage applied to the two DC motors. This leads to a raw velocity estimation and thus to a position estimation affected by errors. In addition to this problem wheel slippage, carpet ”springiness” and uneven floors can affect the odometry accuracy. In Fig. 6.1 is depicted a test we did to see the odometry error we were talking about. The robot has been positioned on the origin of the global reference frame and we gave to the agent four points ($x_1 = (0.5, 0)$, $x_2 = (0.5, 0.5)$, $x_3 = (0, 0.5)$ and $x_4 = (0, 0)$) to visit. The blue line represents the ideal trajectory the robot should have done computed thanks to its odomoetry while the red line is the trajectory the robot actually did (recorded thanks to the motion capture system) because of the problem we have just discussed. As we can see after three point to point movements the robot totally failed to reach the final destination.

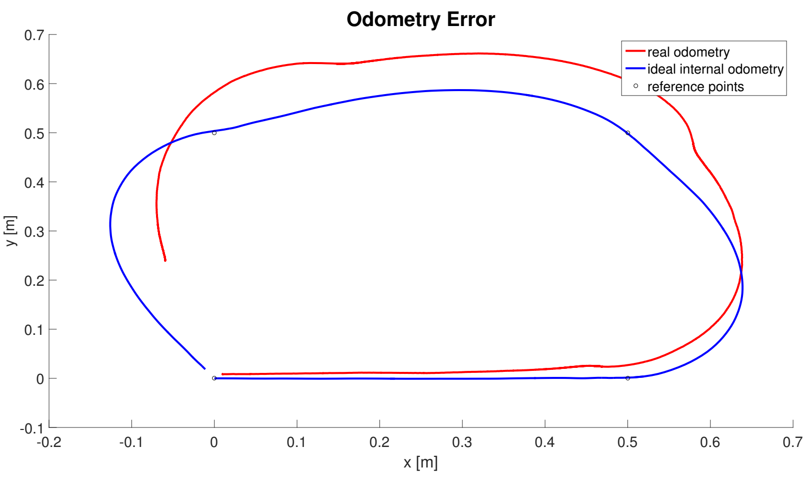


Figure 6.1: Robot trajectories test: the blue line is the ideal path computed thanks to the odomoetry while the red line is the real trajectory the robot did.

For these reasons for our purposes we used an optical motion capture system with passive markers equipped with 12 cameras from *BTS Bioengineering* in the following configuration.

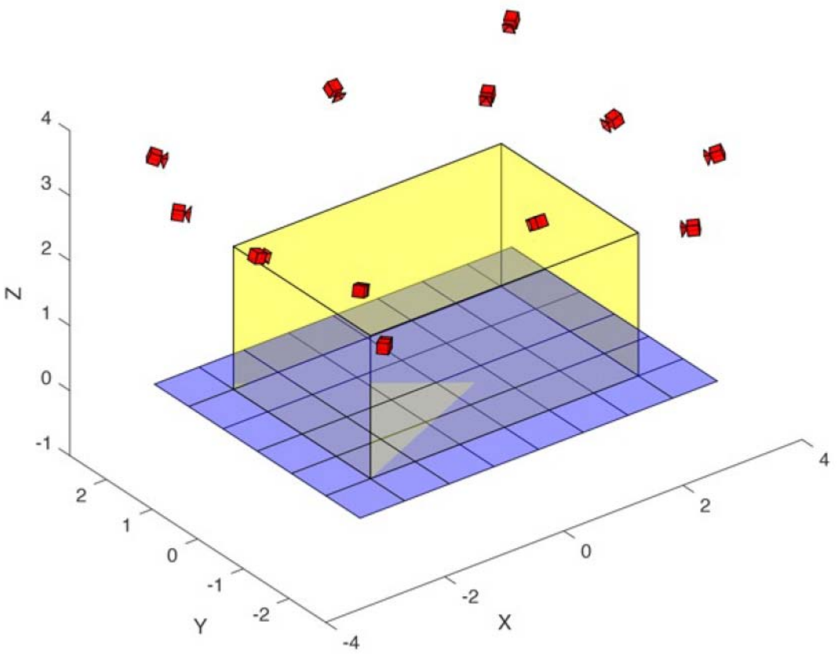
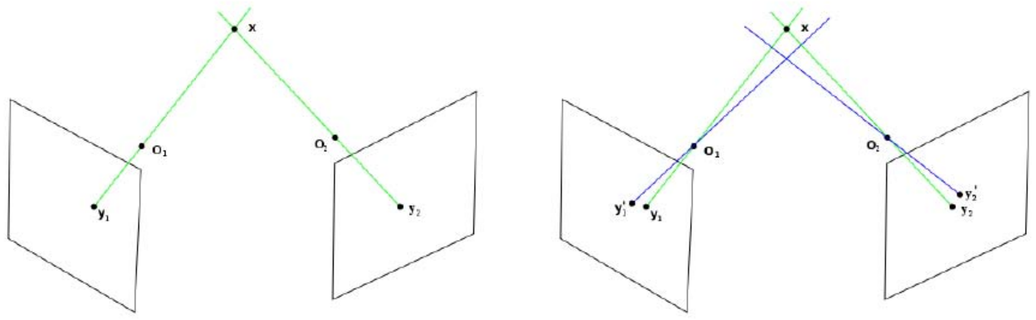


Figure 6.2: MAGIC Lab cameras configuration.

The *BTS* motion capture system is able to return an accurate robots position thanks to 4 markers attached to the Wheelphone chassis in an unique cross configuration. In this chapter we will explain how the motion capture system can estimate the robots pose and how we implemented our algorithms with Wheelphone devices.

6.1 Markers position in space

Assume that we are using two calibrated pinhole cameras with known geometry in space as shown in Fig 6.3 (a).
We denoted the marker position in the 3D space with the point $\mathbf{x}$, $\mathbf{O_1}$ and $\mathbf{O_2}$ are the focal points of the two cameras while $\mathbf{y_1} = [u_1^{'} \quad v_1^{'}]^T$ and $\mathbf{y_2} = [u_2^{''} \quad v_2^{''}]^T$ are the two image points related to $\mathbf{x}$ for the first and the second camera respectively. Under these assumptions it is always possible to determine the two projection lines drawn in green and in the ideal case, those two lines must intersect exactly on $\mathbf{x}$. With basic linear algebra it is easy to determine that intersection point and thus the absolute position of the marker in space.
Fig. 6.3 (b) shows the real case scenario, where $\mathbf{y_1}$ and $\mathbf{y_2}$ can not be measured exactly due to various factors such as lens distortion or noisy image sensors. Let's name the measured values $\mathbf{y_1^{'}}$ and $\mathbf{y_2^{'}}$. In this case the projection lines that we compute, depicted in blue, may not intersect in the 3D space.



(a) Ideal case (b) Real case

Figure 6.3: Triangulation

This means that, given the values $\mathbf{y_1'}$ and $\mathbf{y_2'}$, we have to compute an estimate of the position of the marker $\mathbf{x}_{est}$. In order to achieve this various methods are proposed in literature such as the mid-point method, the direct linear transformation or the optimal triangulation.

6.2 Robot Pose Estimation

In the previous Section we learned how a motion capture system can compute the position of a single marker in space. If we want to estimate the pose of a more complex object (which is the position and orientation of the object in the 3D space), like our robots, we need to work with different patterns of markers. Multiple markers however are non distinguishable from a camera point of view: they have all the same colour and they all look as bright elliptic blobs, whose position is later approximated by their centroid. That's why we need to develop a strong strategy to be able to keep track of our markers individually for every camera frame. The problem of the pose reconstruction can be solved thanks to the **Marker Labelling** and to the **Pose Reconstruction** procedures.

6.2.1 Marker Labelling

Marker Labelling is the procedure of identifying and keeping track of each marker individually in every set of frames within the 3D space. Commonly tracking is achieved by searching for the marker in the next instant of time in the spherical region where it most probably moved. Further improvements of this approach involve Kalman filtering to achieve a better prediction of the motion of the marker and therefore a more accurate search region. Whether this method has proved to work properly it performs poorly when a marker disappear from the scene or when two or more markers appear to be very close to each other which is our case. The approach we use in the MAGIC Lab takes advantage of the rigidity constraints which link every point of the solid object we want to track and in particular every marker attached to it. We placed four markers on each Wheelphone in different patterns in such a way that there were no equally spaced pairs. An example of a markers pattern is shown in Fig.6.4

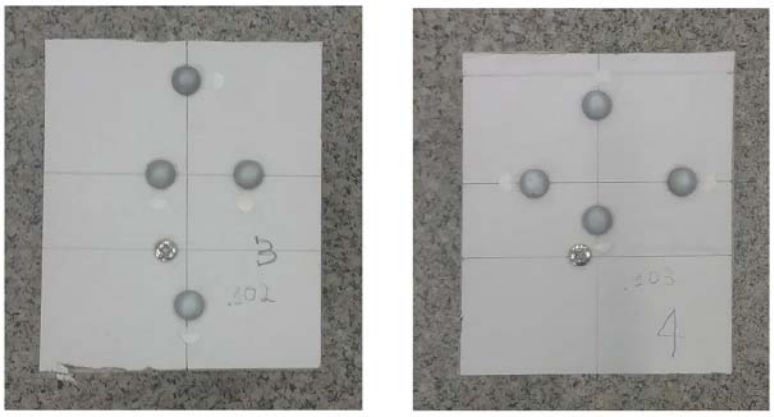


Figure 6.4: Two different cross patterns made of 4 markers

Each pattern is attached to a single robot as shown in Fig. 6.5

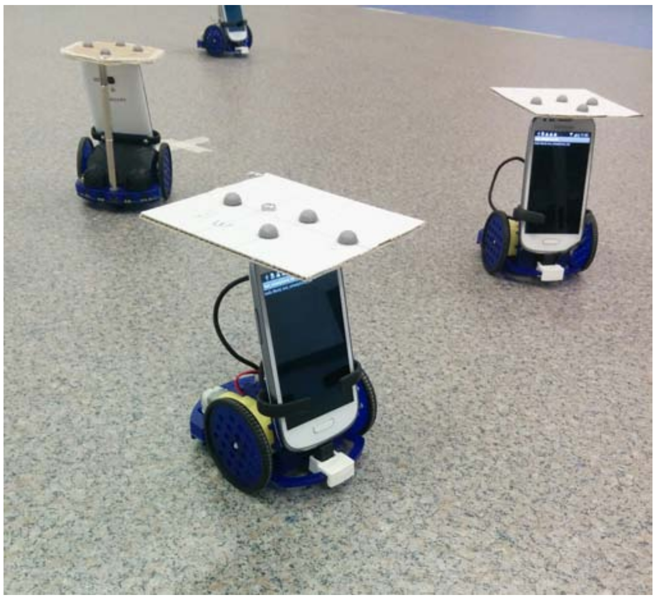


Figure 6.5: Wheelphone robots equipped with their own different cross markers pattern
.

Every cross configuration of markers can be modelled with an undirected connected graph where the nodes represent each different marker and the weights of the edges represent the distance between each pair of nodes. Note that the distances between each pair of markers are invariant for any rotation or translation of the robots 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 model

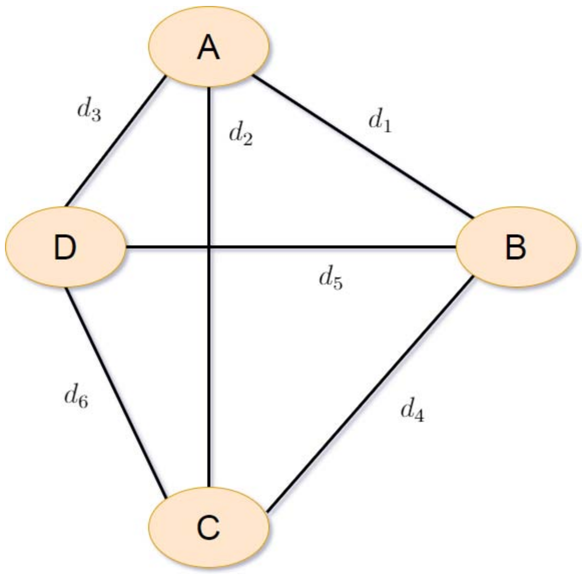


Figure 6.6: Reference Model

for the structure defined by the markers. We do this by means of two matrices, $P_{model} \in \mathbb{R}^{3\times4}$ (which stores the four markers absolute position at initial time) and $D_{model} \in \mathbb{R}^{4\times4}$, where $d_{ij} = \|p_i - p_j\|$ and p_k represents the k-th column of P_{model}.
Referring to Fig. 6.6 we obtain:

$$P_{model} = \begin{bmatrix} x_A & x_B & x_C & x_D \\ y_A & y_B & y_C & y_D \\ z_A & x_B & z_C & z_D \end{bmatrix} \tag{6.1}$$

$$D_{model} = \begin{bmatrix} 0 & d_1 & d_2 & d_3 \\ d_1 & 0 & d_4 & d_5 \\ d_2 & d_4 & 0 & d_6 \\ d_3 & d_5 & d_6 & 0 \end{bmatrix} \tag{6.2}$$

Then the following logic is implemented to successfully label all the markers:

1. for any new marker set obtained at time $t > 0$ we compute with the same logic seen previously the matrices P_{means} and $D_{means} \in \mathbb{R}^{n\times n}$, where $n > 4$ if other markers appear on the scene;

2. we compare each column d_i of D_{model} with the columns of D_{means} and we rearrange the columns of P_{means} as long as we find three correspondences;

3. When the algorithm stops we end up with the matrix P_{meas} ordered and polished of the columns corresponding to the markers on the scene which didnt belong to the model.

We will see shortly that we need just three points to compute the position and orientation of our object in 3D space but using more markers improve the algorithm robustness in case of one or

more misdetections. In fact we can add to our reference model as many markers as we want always being careful to place them in such a way that no equally distanced pairs are involved.
The Marker Labelling pseudocode is here shown:

Algorithm 5 Marker Labelling Algorithm

```
1: Data: $P_{means}$, $D_{means}$, $D_{model}$
2: N = size($D_{model}$);
3: for $j \longleftarrow 1$ to N do
4:     Find the index of the column with three correspondences;
5:     $j \longleftarrow$ whoAmI($D_{model}(:,i), D_{means}$)
6:     $P_{ordered}(:,i) \longleftarrow P_{means}(:,j)$
7: end for
8: Return $P_{ordered}$
```

6.2.2 Pose Reconstruction

Once we know the correspondence between the original model and the set of points at the generic time $t > 0$ it is possible to design an algorithm which reconstructs the pose of the robot equipped with the specific markers configuration represented by the initial model.
Let's recall, by referring to Fig. 6.7, that at every robot has been assigned a cross configuration of markers and for each pattern every distance between a marker and the center of the cross "o" is different in order to doesn't make the motion capture system confuse two pattern models during the marker labelling procedure. It's also important to note that every pattern is made of the same cross lines but the markers are placed on different distances from the origin, therefore every pattern have the same center "o".

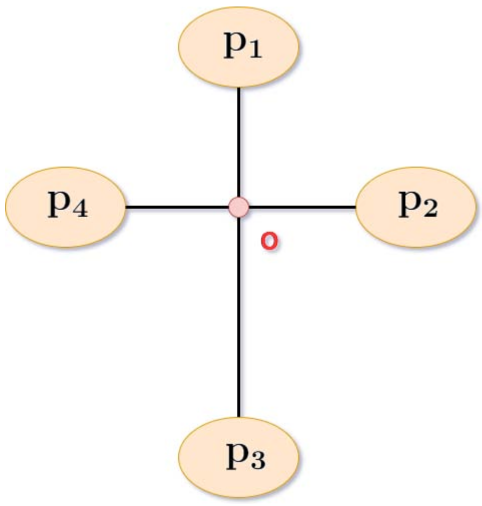


Figure 6.7: Example of a cross pattern of four markers.

.

That being said let $P = \{p_1, \ldots, p_n\}$ and $Q = \{q_1, \ldots, q_n\}$ be two set of corresponding points representing the markers positions in $\mathbb{R}^3$ related by the following equation:

$$q_i = Rp_i + T + N_i \qquad i = 1, \ldots, n \tag{6.3}$$

with N_i noise vector.
If any other robot is in the scene and if all the markers are correctly detected we have that $n = 4$ and thus $P = \{p_1, \ldots, p_4\}$ and $Q = \{q_1, \ldots, q_4\}$.
We wish to find a rigid transformation that optimally aligns the two sets in the least square sense (P represents the model of a markers cross patterns while Q contains the position of the points seen by the cameras at a specific time t), i.e., we seek a rotation R and a translation T such that:

$$(R, T) = \underset{R,T}{\operatorname{argmin}} \sum_{i=1}^{n} w_i \|(R\mathbf{p_i} + T) - \mathbf{q_i}\|^2 \tag{6.4}$$

where $w_i > 0$ are weights for each point pair. The proof of this statement is shown in [22].
Let's suppose that $\hat{R}$ and $\hat{\mathbf{T}}$ are the solutions of the least squares equation (6.4), then Q and $Q^{'} = \hat{R}Q = \hat{T}$ have the same cross center, i.e.

$$o_Q \equiv o_{Q'} \tag{6.5}$$

where o_Q and $o_{Q'}$ are the centers of the cross pattern formed by the markers which are represented in space by the points of Q and $Q^{'}$.
Let Σ^2 be the operator:

$$\Sigma^2 = \|h_i^{'} - Rh_i\|^2, \qquad h_i^{'} = q_i - o_Q \qquad h_i = p_i - o_P \tag{6.6}$$

with o_p the center of the cross pattern formed by the markers which are represented in space by the points of P.
At this point the least square problem (6.4) can be determined by solving the following two problems:

1. find the matrix $\hat{R}$ which minimize the equation (6.6)
2. compute the transitional vector $\hat{T} = o_Q - o_P$.

The problem 1 can be solved thanks to the Singular Value Decomposition (SVD) following these steps:

- **step 1:** given the points sets P and Q let's compute then centers of the cross patterns o_P and o_Q. This can be achieved by, again referring to 6.7, finding first the line passing through the points $p_1 = (x_1, y_1, z_1)$ and $p_3 = (x_3, y_3, z_3)$

$$r_{p_1p_3} = \begin{cases} x = x_1 + (x_3 - x_1)t \\ y = y_1 + (y_3 - y_1)t \\ z = z_1 + (z_3 - z_1)t \end{cases}$$

and then computing the plane perpendicular to the line $r_{p_1p_3}$ and containing the points $p_2 = (x_2, y_2, z_2)$ and p_4:

$$v_1x + v_2y + v_3z + d = 0$$

where $v_1 = (x_3 - x_1)$, $v_2 = (y_3 - x_1)$, $v_3 = (z_3 - x_1)$ and $d = -(v_1x_2 + v_2y_2 + v_3x_3)$. By rearranging the previous equations we have that the center point o_P of the set P is specified by the following system equations:

$$o_P = \begin{cases} v_1x + v_2y + v_3z - (v_1x_2 + v_2y_2 + v_3x_3) = 0 \\ y = y_1 + \dfrac{v_2}{v_1}(x - x_1) \\ z = z_1 + \dfrac{v_3}{v_1}(x - x_1) \end{cases}$$

and thus:

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ -\frac{v_2}{v_1} & 1 & 0 \\ -\frac{v_3}{v_1} & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \begin{bmatrix} v_1x_2 + v_2y_2 + v_3z_2 \\ y_1 - \frac{v_2}{v_1}x_1 \\ z_1 - \frac{v_3}{v_1}x_1 \end{bmatrix} = 0 \tag{6.7}$$

$$AX - b = 0 \tag{6.8}$$

with $A = \begin{bmatrix} v_1 & v_2 & v_3 \\ -\frac{v_2}{v_1} & 1 & 0 \\ -\frac{v_3}{v_1} & 0 & 1 \end{bmatrix}$, $b = \begin{bmatrix} v_1x_2 + v_2y_2 + v_3z_2 \\ y_1 - \frac{v_2}{v_1}x_1 \\ z_1 - \frac{v_3}{v_1}x_1 \end{bmatrix}$ and $X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$.

In the same way we can also obtain o_Q considering the points $q_i \in Q$.

- **step 2:** let's define the matrix $H \in \mathbb{R}^3$ as:

$$H = \sum_{i=1}^{n} h_i h_i'^T;$$

- **step 3:** compute the SVD decomposition of H:

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

  where U and V are orthonormal matrices with the appropriate dimensions;

- **step 4:** compute

$$X = UV^T$$

- **step 5:** lastly if $det(X) = 1$ we have that $\hat{R} = X$. (The case $det(X) = -1$ is shown in [22] and it is not of our interest).

After computing the matrices $\hat{T}$ and $\hat{R}$ we are finally able to reconstruct the robot pose. His position in fact is equal to the vector $\hat{T}$ which represents the position in the 3D space of the cross center o_Q of Q. The orientation, expressed by the angles $\begin{bmatrix}\phi & \theta & \psi\end{bmatrix}^T$, namely roll, pitch and yaw, can be computed from $\hat{R}$. If the rotation matrix $\hat{R}$ is in the form:

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

we can write relations:

$$\phi = Atan2(-r_{32}, \sqrt{r_{31}^2 + r_{33}^2}) \tag{6.9}$$

$$\theta = Atan2(-r_{31}, r_{33}) \tag{6.10}$$

$$\psi = Atan2(-r_{12}, r_{22}) \tag{6.11}$$

where $\phi \in (-\frac{\pi}{2}, +\frac{\pi}{2})$.

6.3 Implementation

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

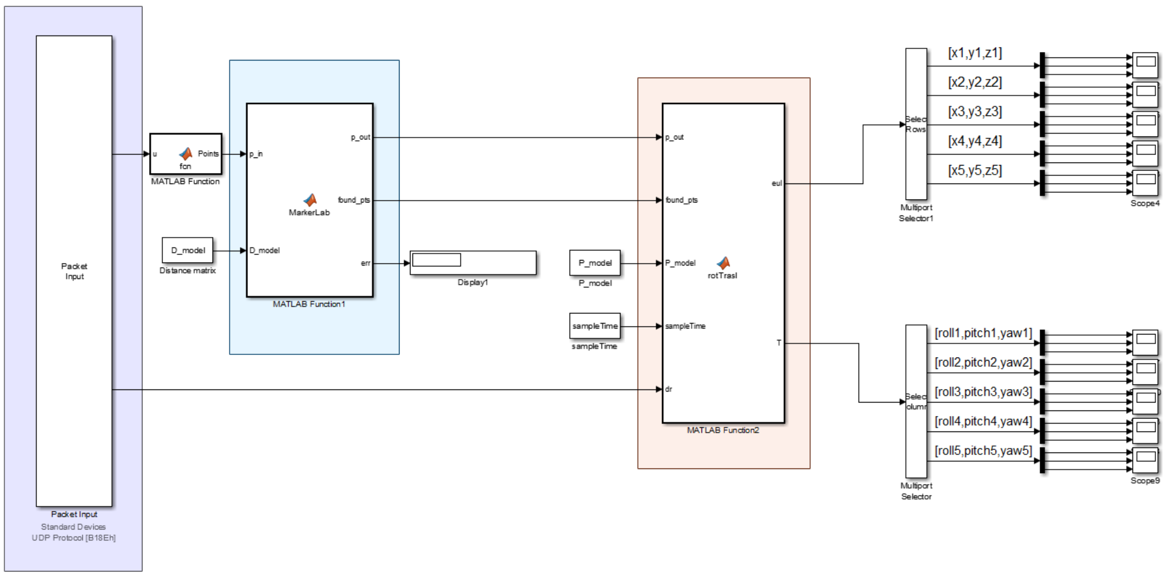


Figure 6.8: Simulink Motion Capture Block scheme

The purple area of Fig. 6.8 implements the communication protocol as seen in Section 2.2, the cyan area implements the marker labelling algorithm which returns the reordered points for each pattern model and finally the red area is able to compute, by means of a MATLAB function, the pose of every cross center for each markers pattern. The x and y coordinates and the yaw angle ψ

of every cross center points are sent to the corresponding robot via UDP communication in order to apply a feedback to our robots control. Control which runs in real time inside the smartphone and which can be seen in Fig. 6.9.

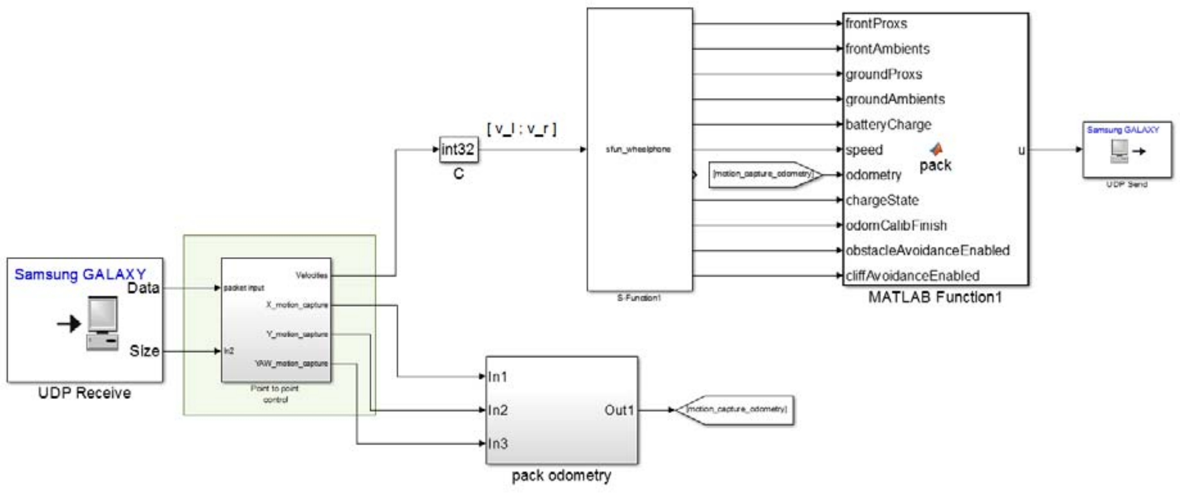


Figure 6.9: Simulink Wheelphone Control Block scheme

The green area implements the point to point control we saw in Section 4.2.1 but this time we will not use the odometry computed by the robot. We'll in fact use the coordinates (x, y, and ψ) sent by the motion capture system. In this case the target point coordinates (x_r, y_r) are given by the mapping algorithm chosen: as we have seen, thanks to the strategy of the algorithm, it will return the points of the map each robot must explore. All the information needed to accomplish the control, such as the target points coordinates and the odometry returned from the motion capture system ,are sent from the server to the robot again via UDP protocol. A more detailed scheme of this point to point control is shown in Fig. 6.10.

6.4 Results

To test our algorithms in our laboratory, for each markers pattern, we decided to position the cross center (which is the point that the motion capture system tracks in the 3D space) on top of the center of the wheels axis of the corresponding robot.
This was done because in our control, when we computed the errors e_d and e_θ and then the velocities (4.14), we chose the midpoint M as reference point.
The arena, which is the squared planar area of $4m^2$ in the MAGIC Lab where our robots can be placed and detected by the motion capture system, has been represented within the MATLAB script by a domain $\mathcal{X} = [-1, 1] \times [-1, 1]$. Thus an unit of the domain corresponds to one meter in the real world.
The first thing we did was to compare the robot internal odometry with the motion capture con-

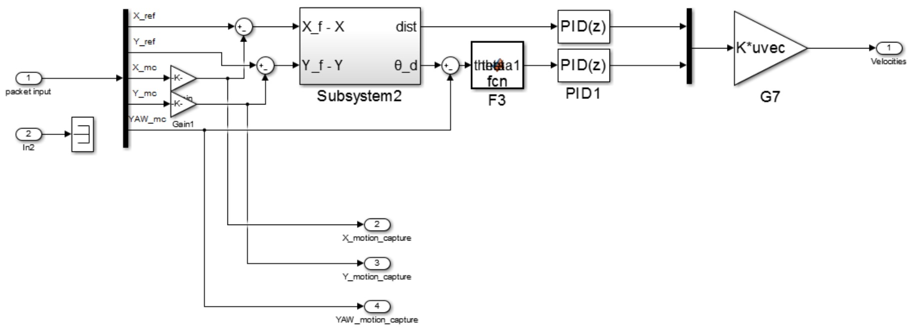


Figure 6.10: Detailed Simulink Wheelphone Point to Point Control Block scheme

trol. We decided in fact to give to a robot four points to visit ($x_1 = (0.5, 0)$, $x_2 = (0.5, 0.5)$, $x_3 = (0, 0.5)$ and $x_4 = (0, 0)$) and to save the trajectories obtained by driving the robot with its odometry and with the motion capture control. The results are shown is Fig. 6.11 where the red line represents the internal odometry trajectory and the blue line represents the motion capture control trajectory.
It's easy to see that the motion capture control is much more accurate and it's capable of returning a reliable and stable position of the cross center of the robot we are tracking. The odometry control, for the same reasons we have seen in the introduction of this chapter, can't even reach the last point. After two movements the internal odometry fails to make the robot move to the

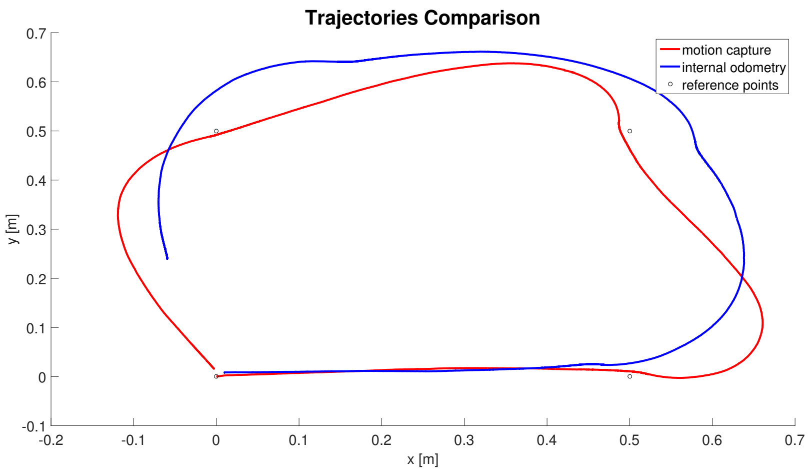


Figure 6.11: Robot trajectories comparison: internal odometry against motion capture control.

second point to visit of about 10 cm and at the fourth movement the robot stopped his run at 25 cm from the last point x_4. This error will continue to increase with the number of movements. Therefore it's clear that for our application, where we want to perform several movements with

various mobile agents, the internal odometry was almost unusable.
The motion capture control is more consistent: it makes the robot visit the points with a precision of few centimeters.
It's also interesting to compare the trajectories done by the robots for every algorithms we designed. We thus decided to compare them in our laboratory, where the walkable planar arena has been again represented by the squared domain $\mathcal{X} = [-1, 1] \times [-1, 1]$, with $N = 4$ agents, for a number of iterations $k = 3$ and with a squared grid points set $\mathcal{X}_{grid}$ with a resolution $\Delta = 0.5$ meters.

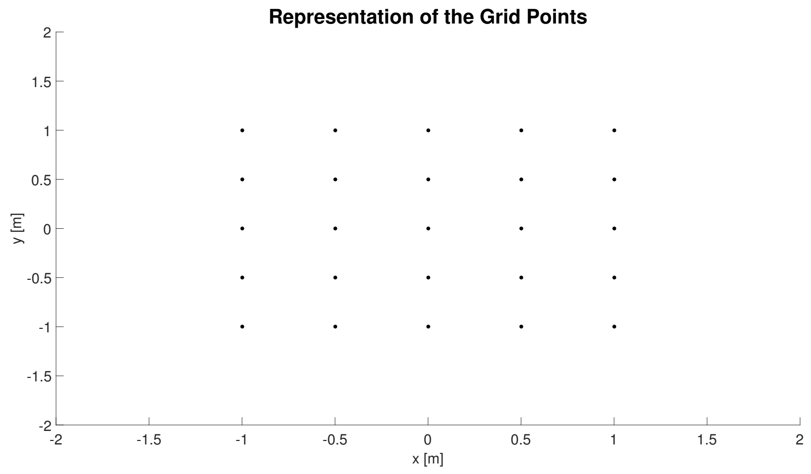
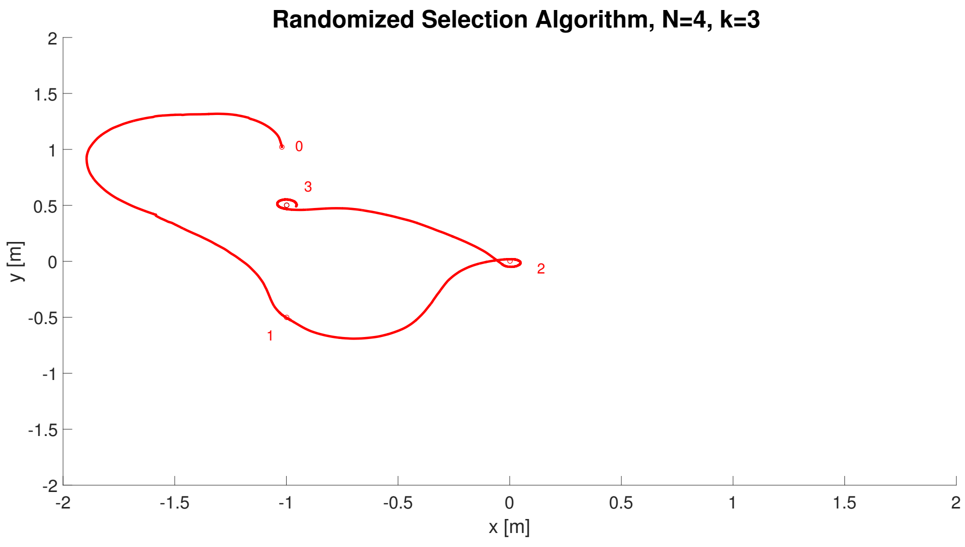


Figure 6.12: Position of the grid points within the map.

In Fig. 6.13 are depicted in different colours the trajectories using the Randomized Selection Algorithm.
As expected the target points for each iteration have been chosen with no particular strategy and the robots trajectories result to be very tangled. It is interesting to note that this mapping algorithm allows a target point to be assigned for two or more consecutive iteration to the same robot as shown in the figure for the black trajectory.
In Fig. 6.14 are shown the trajectories using the Minimum Distance Algorithm.
This time the algorithm, at each iteration, seems to make the mobile vehicles move towards points which are relatively close to each other. This happens because the algorithm, at every iteration and for every device, searches for the points with the highest posterior variance and then assigns them to the robots thanks to a minimum distance criteria. If an area of the map has not been visited yet, the target points result to be all within this unknown region. In the figure, for example, this unexplored area was the center of the map, where at the first iteration the algorithm made the robots move towards an "L" shaped set of target points positioned, in fact, in the middle of our domain $\mathcal{X}$ (see the yellow dashed line).
Lastly in Fig. 6.15 we have the trajectories obtained using the Distance Weighted Algorithm.
In this case it's easy to see that at each robot, for each iteration, is given as target point one of the adjacent points to their position (more specifically the one with the highest ratio (5.9)). This

fact is a consequence of the reason why this algorithm has been designed: in a real case scenario it may be desired to have a good balance between the drain on resources and the quality of the measurements taken in a specific point in space.



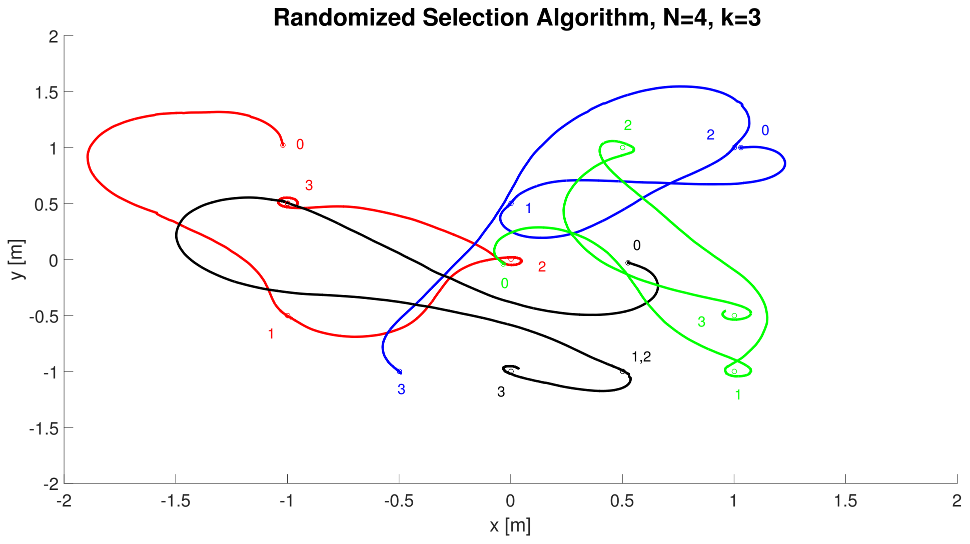
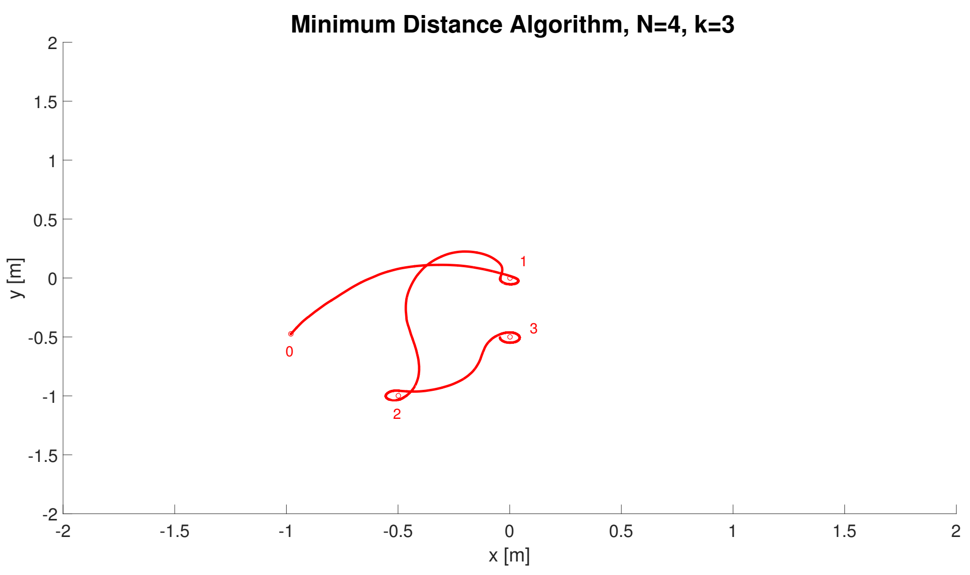


Figure 6.13: Randomized Selection Algorithm trajectories.



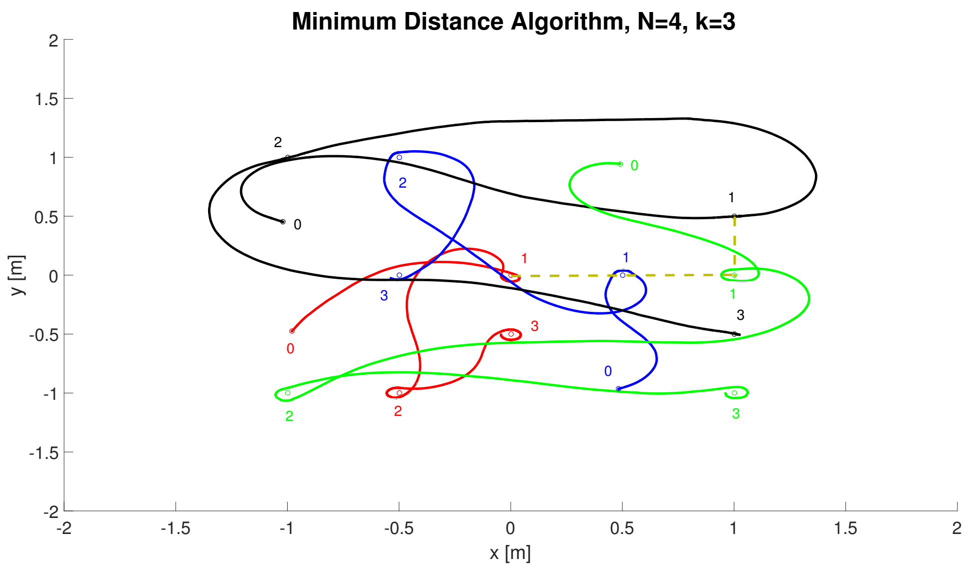
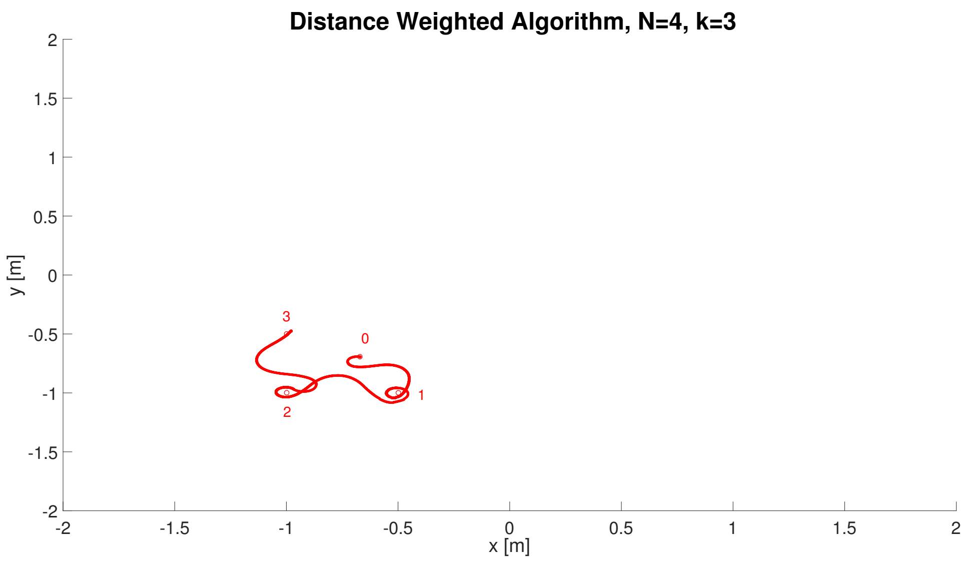


Figure 6.14: Minimum Distance Algorithm trajectories.



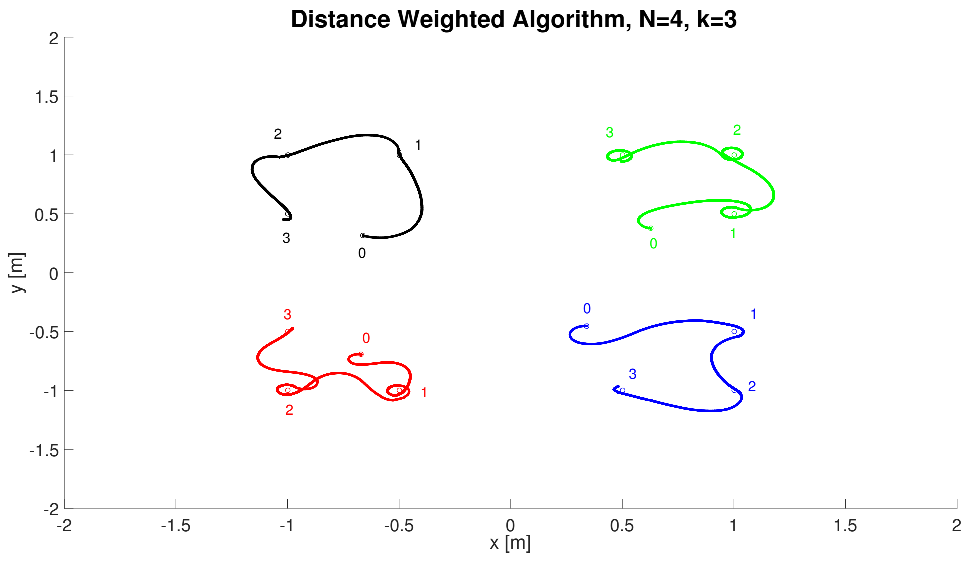


Figure 6.15: Distance Weighted Algorithm trajectories.

Chapter 7

CONCLUSIONS

In this dissertation we proposed and applied with real differential drive agents some mapping algorithms based on the article in [19].

In Chapter 3 we explained how we developed a stable driver interface needed to control, thanks to the MATLAB/Simulink environment, our Wheelphone devices. The drivers file have been written with the intent of don't overwork the computational capabilities of the CPU of the robots.

In Chapter 5 we addressed the problem of mapping with multiple robots assuming the sensory field which approximate the sensory distribution of events is unknown. We proposed three algorithms in the context of a client-server architecture which is guaranteed to asymptotically exactly estimate the unknown sensory function. We also provided an alternative version of the algorithms which, using a non growing information matrix I, have dramatic reduction in terms of memory and CPU requirements. We also tested the performance of the algorithms via extensive numerical simulations and we tried to find the right application for each algorithm.

In Chapter 6 we showed how we tested our algorithms in a real case scenario. The server knows the position of the robots thanks to a motion capture system which, using different cross shaped markers patterns, returns the pose of our agents in the 3D space. This is possible by utilizing the Marker Labelling and the Pose Reconstruction algorithms. The point to point control which the robots use in order to move towards their target points is also proposed. This control, thanks to the work we explained in Chapter 3, runs in real time as an android app on the phone attached to the wheelphone mobile base. We also commented some test we did in our MAGIC Lab where we compared the robots trajectories of each algorithm.

7.1 Future Developments

Several developments should be studied in the future. Here some examples are written.

- Developing the same three algorithms we saw in this thesis but in a Distributed Gossip case where, without the help of a server computer, the target points are computed by the robots which are capable of exchanging information with each other.

- Implementing, with true wheeled devices, some algorithms capable of resolving the problem of simultaneously mapping and coverage.
- Making the mapping algorithms work in the case of a complex environment with insurmountable obstacles. This problem needs to be solved with a collision avoidance and wall following algorithms. This are algorithms which need a reliable hardware such as some good proximity sensors.

Appendices

Appendix A

Target Language Compiler

Target Language Compiler (TLC) is an is an advanced feature within Simulink Coder. It allows you to customize generated code in order to produce platform-specific code or incorporate your own algorithmic changes for performance, code size or compatibility with existing methods that you prefer to maintain. It provides:

- A set of TLC files corresponding to a subset of the provided Simulink blocks.
- TLC files for model-wide information that specify header and parameter information.

The TLC files are ASCII files that explicitly control the way code is generated. By editing a TLC file, we can alter the way code is generated (we can produce ANSI C or C++ code). The Simulink Coder invokes The Target Language Compiler after the Simulink model is compiled into an intermediate form (`model.rtw`) that is suitable for generating code. The TLC then transforms this representation file of the diagram into C or C++ code. The `model.rtw` file contains a partial representation of the model describing the execution of the block diagram in a high-level language. After reading the `model.rtw` file the target Language Compiler generates its code based on target files, which specify particular code for each block, and model wide files which specify the overall code style. To create a target-specific application, the code generator requires a template makefile that specifies a C or C++ compiler options for the build process. The code generator transform the template makefile into a target makefile (`model.mk`). The overall procedure is shown in Fig.A.1.
The TLC is really helpful when we must:

- Change the way code is generated for a particular Simulink block
- Inline S-functions in our model
- Modify the way code is generated in a global sense
- Perform a large scale customization of the generated code, for example, if you need to output the code in a language other than C

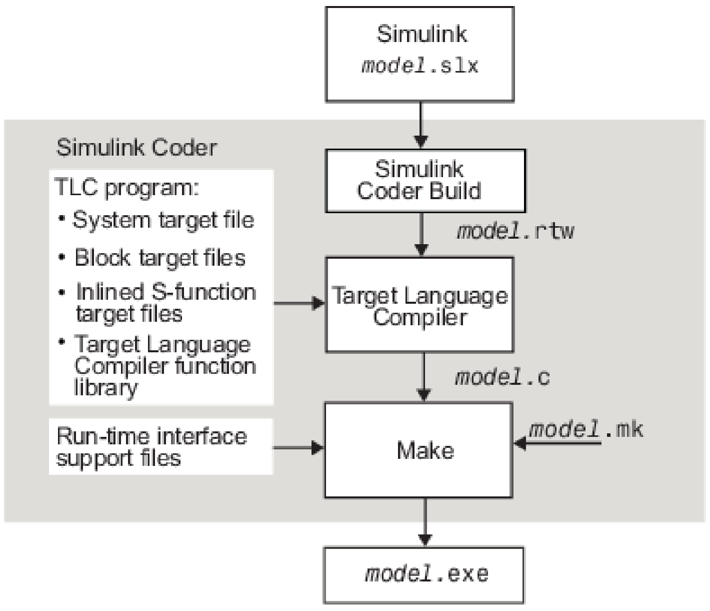


Figure A.1: This diagram shows how the Target Language Compiler fits in with the code generation process.

If an inling TLC file is not provided most targets support the block by recompiling the C MEX S-function for the block (**Noninlined S-Functions**). The drawback is a bad memory usage and low speed performance when using a C/C++ coded S-function. A limited subset of API calls is also supported within the code generator. If the most efficient generated code is wanted we must inline S-functions by writing a TLC file for them.
To inline an S-function means to provide a TLC file for an S-function block (**Inlined S-Function**) that will replace the C, C++, Fortran or MATLAB lenguage version of the block that was used during the simulation. Usually an inlined S-function performs better than a noninlined S-functions. The main idea is to improve code from blocks by understanding what part of a block's operations are active and used in the generate code and what parts can be predetermined or left out. This basically means that the TLC code in the block target file will select an algorithm that is a subset of the algorithms contained in the S-function itself and then it will also selectively hard-code numerical parameters that are not to be changed at run time. This reduces code memory size and results in code that is often much faster than its S-function counterpart. In a TLC file must be specified how the S-function block is initialized and how to compute the block output.
There are two types of inlining:

1. fully inlined S-functions

2. wrapper inlined S-functions

In the first case a full implementation of the block is contained in the TLC file for the block. In the second case instead of generating the algorithmic code in place, the TLC file calls a C function that contains the body of the code.

Appendix B

Tikhonov Regularization

Let $f\colon \mathcal{X} \longrightarrow \mathbb{R}$ denote an unknown deterministic function defined on the compact $\mathcal{X} \in \mathbb{R}^d$. Assume to have a measurement model as the following:

$$y = f(x) + v,$$

with v white noise and x a generic input location ($x \in \mathcal{X}$).
Given the data set $\{x_i, y_i\}_{i=1}^N$, where N is the number of measurements collected, one of the most used approaches to estimate f is the so called Tikhonov Regularization, which relies on the Tikhonov regularization theory. The hypothesis space is typically given by a reproducing kernel Hilbert space (RKHS) defined by a Mercer Kernel $K\colon \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$.
In the following is reported the Representer theorem, which states that the optimal estimate of a function can be represented as a linear combination of basis functions.

Theorem B.1: Representer theorem

Consider: $\Phi\colon H \longrightarrow \mathbb{R}$ where H is a generic Hilbert space, defined as

$$\Phi(f) = F(L_1 f, \cdots, L_m f, \|f\|_H^2)$$

where $F\colon \mathbb{R}^{m+1} \longrightarrow \mathbb{R}$ and where the L_is are linear and bounded functionals. The map is thus the composition of three different maps: F, the norm in H, and the linear and bounded functionals L_i. The last assumption is that Φ is strictly monotonically increasing w.r.t. the last argument, i.e. it is strictly monotonically increasing w.r.t. $\|f\|_H^2$. Define

$$\hat{f} = \operatorname*{argmin}_{f \in H} \Phi(f)$$

Assume that there exists at least one solution of the previous problem (i.e. that the solution exists but may be not unique). Then it has the form

$$\hat{f} = \sum_{i=1}^{m} c_i g_i$$

where the g_i's are the representers of the various L_i's, i.e. $L_i f = \langle f, g_i \rangle_H$ for all $f \in H$

The Tikhonov regularization problem is comprised in this formulation, in fact the typical cost function is of the form:

$$Q(f) = \sum_{i=1}^{N} (y_i - f(x_i))^2 + \gamma \|f\|_H^2 \tag{B.1}$$

where $\|\cdot\|_H^2$ is the norm defined in the RKHS and γ is the regularization parameter that trades off empirical evidence and smoothness information on f. The estimate of the unknown function is

$$\hat{f} = \underset{f \in \mathcal{H}_K}{\operatorname{argmin}}\, Q(f)$$

where $\mathcal{H}_K$ is the associated RKHS. It is known form the literature that $\hat{f}$ admits the structure of a Regularization Network see [21], being the sum of N basis functions with expansion coefficients obtainable by inverting a system of linear equations. More precisely, one has:

$$\hat{f}(x) = \sum_{i=1}^{N} c_i K(x_i, x) \tag{B.2}$$

where the expansion coefficients c_i are obtained as

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = (\bar{K} + \sigma^2 \mathbb{I})^{-1} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{B.3}$$

while the matrix $\bar{K}$ is obtained evaluating the kernel at the input-locations, i.e.

$$\begin{bmatrix} K(a_1, a_1) & \cdots & K(a_1, a_N) \\ \vdots & \ddots & \vdots \\ K(a_N, a_1) & \cdots & K(a_N, a_N) \end{bmatrix}.$$

The estimate of f admits also a Bayesian interpretation in fact, if f is modeled as the realization of a zero-mean Gaussian random field with covariance K, the noise v is Gaussian, independent of the unknown function and with variance σ^2, setting $\gamma = \sigma^2$ one has:

$$\hat{f}(x) = \mathbb{E}[f(x) | \{x_i, y_i\}_{i=1}^{N}]$$

and the associated posterior variance of the estimate at a generic input location $x \in \mathcal{X}$ is

$$V(x) = Var[f(x)|\{x_i, y_i\}_{i=1}^N] =$$

$$= K(x,x) - \begin{bmatrix} K(x_1,x) & \cdots & K(x_N,x) \end{bmatrix} (\bar{K} + \sigma^2 \mathbb{I})^{-1} \begin{bmatrix} K(x_1,x) \\ \vdots \\ K(x_N,x) \end{bmatrix} \tag{B.4}$$

Notice that the computation of $\hat{f}$ requires $O(N)$ operations and the processing unit has to store all the x_is.

Appendix C

sfun_wheelphone.c

```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)
{   /* Number of expected parameters */
    ssSetNumSFcnParams(S, NUM_PARAMS);
    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);
    /*direct input signal access*/
    ssSetInputPortRequiredContiguous(S, 0, 1);

    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);
}

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

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

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

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

/* Change to #undef to remove function */
#undef MDL_DERIVATIVES
#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
```

Appendix D

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
```

Appendix E

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)
{
}
```

Appendix F

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;         // receiving
       packet length (this the maximum packet length, actually only 23
       bytes are used)
    private final static int packetLengthSend = 63;         // sending
       packet legnth (this the maximum packet length, actually only 3
       bytes are used)
    private final static int USBAccessoryWhat = 0;          // type of
       message received
    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;                    // flag
       indicating if the robot is connected (and exchanging packets) with
        the phone
    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;             // scale
    the speed given in mm/s to a byte sent to the microcontroller
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

//  Others
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() {

    Override
    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 2:
                            case 3:
                                //  send connection req to Wheelphone
                                    PIC24F
                                commandPacket1[0] = (byte)
                                   APP_CONNECT;
                                commandPacket1[1] = 0;
                                accessoryManager.write(commandPacket1
                                   );
                                deviceAttached = true;

                                //  enable control to get new packet
                                   with 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
```

Bibliography

[1] Brian Yamauchi. *A Frontier-Based Approach for Autonomous Exploration*. Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation, Monterey, CA, July 1997, pp. 146-151

[2] G. Dudek, M. Jenkin, E. Milios and D. Wilkes. *Robotic exploration as graph construction.*. PIEEE Transactions on Robotics and Automation, 7:6, pp. 859-865, 1991.

[3] Brian Yamauchi. *Frontier-Based Exploration Using Multiple Robots*. Proceedings of the Second International Conference on Autonomous Agents (Agents '98), Minneapolis, MN, May 1998, pp. 47-53

[4] B. Yamauchi, *"Decentralized coordination for multirobot exploration"* Robotics and Autonomous Systems, vol. 29, no. 2-3, pp. 111-118, 1999.

[5] R. Simmons, D. Apfelbaum, W. Burgard, D. Fox, M. Moors, S. Thrun, and H. Younes, *"Coordination for multi-robot exploration and mapping"* in Proceedings of the National Conference on Artificial Intelligence, 2000, pp. 852-858.

[6] W. Burgard, M. Moors, C. Stachniss, and F. Schneider, *"Coordinated multi- robot exploration"* IEEE Transactions on Robotics, vol. 21, no. 3, pp. 376- 386, 2005.

[7] Ling Wu, Miguel Angel Garcia, Domenec Puig and Albert Sole, *"Voronoi-Based Space Partitioning for Coordinated Multi-Robot Exploration"* Journal Of Physical Agents, vol. 1, no. 1, 2007

[8] Keith Kotay, Ron Peterson, and Daniela Rus, *"Experiments with Robots and Sensor Networks for Mapping and Navigation* Field and Service Robotics Results of the 5th International Conference. 2006

[9] L. Ljung, editor. *"System Identification (2Nd Ed.): Theory for the User"*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[10] T. Soderstrom and P. Storica, editors. *System Identification"*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[11] G. Pillonetto, A. Chiuso, and G. De Nicolao. *"Prediction error identification of linear systems: A nonparametric Gaussian regression approach"*. Automatica, 47(2):291 305, 2011.

[12] T. Poggio and F. Girosi. *"Networks for approximation and learning"*. In Proceedings of the IEEE, volume 78, pages 14811497, 1990.

[13] S. Smale and D.X. Zhou. *"Online learning with Markov sampling"*. Analysis and Applications, 07(01):87113, 2009.

[14] Y. Xu, J. Choi, S. Dass, and T. Maiti. *"Sequential Bayesian prediction and adaptive sampling algorithms for mobile sensor networks"*. IEEE Transactions on Automatic Control, 57(8):20782084, 2012.

[15] Y. Xu, J. Choi, and S. Oh. *"Mobile sensor network navigation using Gaussian processes with truncated observations"*. IEEE Transactions on Robotics, 27(6):11181131, 2011.

[16] 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):35203530, 2013.

[17] Nikolay A Atanasov, Jerome Le Ny, and George J Pappas. *"Distributed algorithms for stochastic source seeking with mobile robot networks"*. Journal of Dynamic Systems, Measurement, and Control, 137(3):031004, 2015.

[18] J. Choi, J. Lee, and S. Oh. *"Swarm intelligence for achieving the global maximum using spatio-temporal Gaussian processes"*. In American Control Conference, 2008, pages 135140. IEEE, 2008.

[19] M. Todescato, A. Carron, R. Carli, G. Pillonetto, L. Schenato. *"Multi-Robots Gaussian Estimation and Coverage Control: from Server-based to Distributed Architecture"*. Automatica, 20XX

[20] A. N. Tikhonov and V. Y. Arsenin, *"Solution of Ill-posed Problems"*. Wiston, 1977.

[21] F. Girosi, M. Jones, and T. Poggio, *Regularization theory and neural networks architecture, Neural Computation*, vol. 7, pp. 219269, 1995.

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