UNIVERSITÀ DI BOLOGNA

School of Engineering

Master Degree in Automation Engineering

Distributed Control Systems

COVERAGE CONTROL FOR COOPERATIVE MULTI-ROBOT NETWORKS

Professor: **Giuseppe Notarstefano**

Students:
Donato Brusamento
Mattia Micozzi
Guido Carnevale
Lorenzo Draghetti

Academic year 2018/2019

Abstract

The aim of the project hereby presented was to transpose the algorithms described in the research paper [1] into a proper Matlab environment suitable for simulation and results analysis. That paper proposes a procedure to manage coverage control for mobile sensing networks, focusing on autonomous vehicles playing the role of mobile tunable sensors.
The purpose of those (and our) algorithms is to find a converging method able to drive the robots into the best configuration, taking into account the spatial probability distribution of the occurring event to detect (peculiarity to measure) and the sensors degradation.
The algorithms presented in [1] have been simplified in order to adapt to both an offline, sequential approach (from now on called **offline**,or **centralized**) and to a parallel (single machine), asynchronous one (from now on called **online**,or **distributed**).
In both approaches, the simulation implements a simple proportional closed loop control, but while in the former it is all done inside a centralized *for loop*, in which informations about each agent is readily available to all others, in the latter we tried to be more faithful to the original formulation, in which every agents has to exchange with other agents informations about each other's positions in a timely manner, while being "deaf" (to a certain extent) in-between successive communications.
To achieve all of this we've made use of Matlab's *Parallel Computing Toolbox* to simulate the independent agents as communicating threads.
In the distributed version of the simulation we also implemented a visual comparison between the two approaches so to show how they bring slightly different results.

Contents

List of Figures

Introduction

A mobile wireless sensor network (MWSN) can be defined as a wireless sensor network (WSN) in which the sensor nodes are mobile. For example the nodes can be wheeled robots scattered in a given area.

A problem that can arise in this context is the **optimization** of coverage of a given area knowing the density distribution function, defined on that area, of a given property we would like to measure. The objective of this project was to implement a **distributed and asynchronous algorithm** to solve this problem, following the method proposed in the research paper [1].

The considered framework, as suggested by the aforementioned paper, is the following: n mobile sensor-robots modelled as simple integrators strewn on an area given by a **convex polytope** defined in $\mathbb{R}^2$; on this area a 2-dimensional distribution $\phi : Q \to \mathbb{R}_+$ is defined to represent the density of the feature that nodes have to measure.
Each node has sensing (can locate the other nodes positions) and/or communication capabilities.

Conceptually this algorithm runs iteratively 2 subroutines: one, called "*Adjust-sensing radius algorithm*", to compute the **Voronoi cell** of each sensor, and another one, called "*Monitoring algorithm*" to check if the computation of the Voronoi cell has to be updated because of significant changes in the network (variation of nodes positions, node failures,...).
We implemented this procedure in Matlab in 2 ways:

1. in a **centralized** sequential way
2. in a **distributed** parallel fashion, reading and writing files (stored in a master server) to realize communication among nodes.

These two implementations and the differences between them are well-explained and analysed in this report. In particular, the first chapter regards the problem set-up and description of proposed solutions; the second chapter contains the analysis of obtained results.

Motivations

MWSNs are an emerging field of research in contrast to their well-established predecessor. They are way more flexible than static sensor networks as they can be deployed in any scenario and deal with rapid topology changes due to node failures or new added sensors. In general each node consists of a radio transceiver, a microcontroller powered by a battery and one or more sensors to detect certain properties of the environment. [2]
Typical applications of this kind of network are environment monitoring, surveillance, search and recovery operation, exploration.

In line with this arising topic, our project has fixed the goal of introducing a basic simulation environment that could be useful for elementary tests and above all that is suitable for further developments.

The algorithms and simulations have been implemented in *Matlab* because of its (relatively speaking) simplicity and useful add-ons.
In particular, thanks to the graphical tools readily made available we could also create a real time results **visualization** to accompany the simulations, which is well suited to analyse the behaviour of both algorithm, compare them, and easily fix or add missing features.

Contributions

We believe that, thanks to this project, we created a starting point for building and implementing on real MWSN the proposed control algorithms; while further steps have to be taken in the direction of optimizing the code for embedded platforms (and simulations as well), the main idea of the paper used as a guideline, and its efficiency, have been demonstrated through our framework.

Chapter 1

Problem set-up and solutions

1.1 Setup

1.1.1 Theoretical notions

The algorithm proposed by the main paper is **Lloyd's gradient descent**, which guarantees the convergence to an optimal solution intuitively given by the trade-off between each node's coverage (position) and relative degradation of sensing capabilities.
A main point to be considered is that this solution makes use of the concept of **Voronoi partition**.
Given a subspace S in $\mathbb{R}^N$ (typically a plane) and a set of n points $P = \{p_1, ..., p_n\}$ belonging to this subspace we call Voronoi partition a partition such that there is an associated region for each point and this region consists of all points closer to that point than any other point. We call these regions *Voronoi cells*.
Let's see an example of Voronoi partition of a plane:

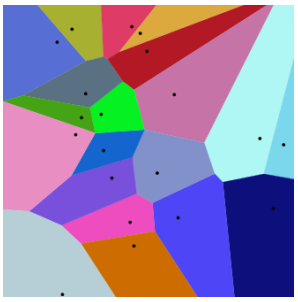

Figure 1.1: Voronoi cells of 20 points on a plane.

Considering a convex polytope Q in $\mathbb{R}^N$ and n sensors, for the sensor positions $P = \{p_1, ..., p_n\}$ the optimal partition of Q is the Voronoi partition $V(P) = \{V_1, ..., V_n\}$, where V_i is the Voronoi cell of the $i - th$ sensor:

$$V_i = \{q \in Q | \, \|q - p_i\| \leq \|q - p_j\|, \forall j \neq i\}$$

Intuitively speaking, each V_i represents the region of space in which the respective sensor i is performing its task.

Because of noise and loss of resolution, we assume also that the sensing quality of a sensor placed at p_i with respect to a point q decreases as much as the distance between p_i and q is large. Then once we have computed the Voronoi partition we have also to find for each Voronoi cell the point in which the sensing degradation is minimized in order to move the sensor in this point. As the paper suggests, we have modeled the sensing degradation as the squared distance between the sensor location and the considered point.
Moreover we have to remember that we want to measure a certain property that is distributed in the space.
Under these assumptions we have that for each Voronoi cell the point in which the sensing degradation is minimized and the "property sensing" is maximized is the so called generalized centroid C_v of the Voronoi cell, namely the center of mass (generalized mass M_v), that we find considering the distribution density function ϕ of the variable that we want to measure as a mass density function ρ.

$$M_v = \int_V \rho(q) \, dq$$

$$C_v = \frac{1}{M_v} \int_V q\rho(q) \, dq$$

Obviously the motion of the agent towards this point modifies the situation and very likely changes the Voronoi partition.

The adopted algorithm **Lloyd's gradient descent** guarantees to reach a global convergence to an optimal solution under the given assumptions.

1.1.2 Adopted assumptions

Although the code was implemented with the objective in mind of staying as much faithful to the paper as possible, some assumptions had to be made in order to work with the simulations running on a single machine, especially

considering that some pieces of code (e.g. the computation of Voronoi cells) tend to slow them down quite a bit, and should then be optimized during further developments.

In particular, with respect to the "ideal" version of the algorithm proposed in the paper [1], the assumptions made are:

1. The *sensing* of other agents is not implemented by a specific **communication protocol**; in the distribution version case, text files were written and read in a similar fashion to *POSIX pipes*, and the only reason preventing an immediate sensing is the lock put to files during mutual r/w.
   This, as a side effects, also "forces" the lack of synchronism between threads.

2. The *timing*, as in the **scheduling** of different subroutines like monitoring and sensing, is not implemented, as it would require real platforms with only few unique threads running so to not introduce large discrepancies in convergence speeds.
   This is not only due to (rather severe) constraints of horsepower available on the used machines, but also to the lack of ready to use POSIX-like scheduling primitives in Matlab's *Parallel Computing Toolbox*.
   On top of that, while this would definitely be an issue in the real world implementation of the algorithm, for the purpose of this project (i.e. demonstration and simulation of the algorithm and creation of essential tools for its correct implementation), the proposed version suffices in its goals, and the following point explains the trade-off that was made in accordance with the Professor's directions.

3. For the same reason, while the correct calculation of **neighbours weights** is implemented and also displayed in the Offline algorithm, in the Distributed version agents do not wait for *events* in order to change control, but rather they first reach the last computed region's centroid and only then they sense others' positions to update their own cell and reference centroid.
   However, should the need arise, the primitives and functions aiding in the event generation are already available in *Ver. A* of the code, as mentioned.

1.2 Inner Workings

1.2.1 Offline algorithm (A)

As said before, in this project two ways to reach the final goal have been followed: the **centralized** algorithm, namely the one which exploits the fact that each position of each agent is known by the others, and the control law is applied in a *sequential* manner, is developed in this section.
The fig. 1.2 allows a more comprehensive approach to the explanation of this algorithm.

The initialization phase lets the user to generate the workspace randomly, or to upload a given benchmark to work with.
If the former choice is made, a set of functions will be called:

- *buildArea*: it creates a **convex area**, namely a polygon with a given number of vertices placed in a given range of the plane;
- *generateAgents*: it places a fixed number of **agents** in the already built area, positioning them randomly;
- *genPolyGrid*: it turns the plane individuated by the area of the polygon into a **discrete grid of points**, in order to build the probability distribution function.
- *gaussianDistr2d*: it generates a **2-dimensional Gaussian distribution function** that represents the distribution of the information to be acquired.

The latter choice allows to define the workspace from an external file; then, except *buildArea*, the same functions are used to set the framework.

Once all the benchmark features are defined, the algorithm makes a first computation of the Voronoi cells, analyzing and classifying the neighbours of each agent, and estimating the **centroid** of each cell, weighted with respect to the density (namely, the **center of mass**). To this purpose, some complex functions have been developed, that will be discussed further:

- *calculateCell*;
- *sliceCell*;
- *genMCgauss*.

Thus, after the first partition of the main area, the control parameters are defined, and the control action can start.
Since the project assumptions state that each agent must be considered as a **single-integrator**, the control consists only in a **proportional action**

aimed to fill the gap between the agent position and the centroid of its Voronoi cell. The sequential fashion is implemented by means of a *for loop*, that lets the agents move one by one, toward its own Voronoi centroid.
In each iteration, actually, the single agent doesn't reach necessarily the goal (namely, the centroid), and at the beginning of the next iteration, it computes another time its Voronoi cell, updating the centroid if some changes happened in the neighbourhood.
In this way, each agent approaches constantly a reference that is moving in time, because actually at each iteration the centroid is updated.
Finally, the algorithm stops when every agents has reached a distance from its cell centroid lower than a given threshold.

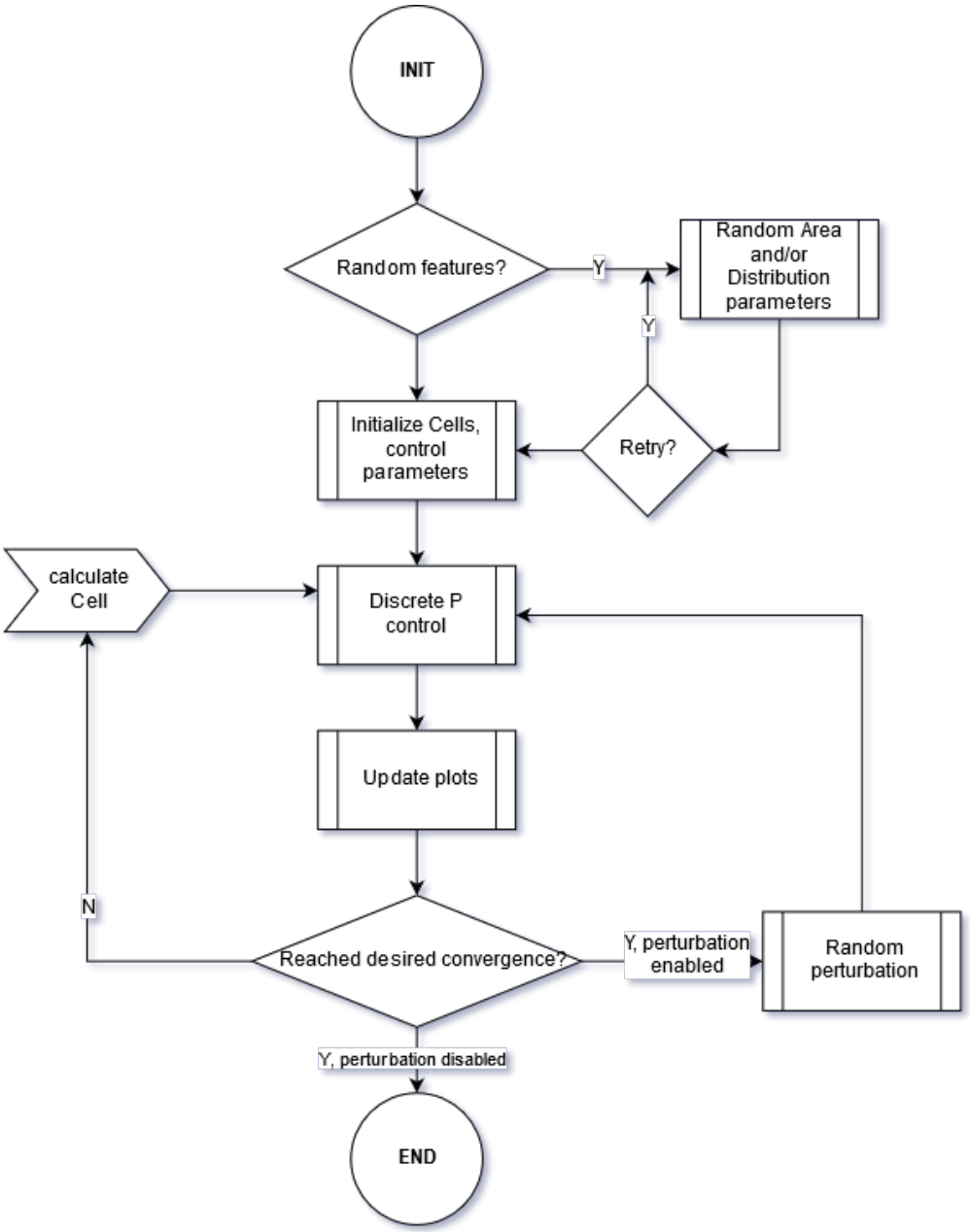

Figure 1.2: Algorithm A flow.

1.2.2 Online algorithm (B)

In this section, the **distributed** or **online** algorithm is explained. As already mentioned,this procedure is based on parallel threads (as many as the number of nodes) exploiting the *Parallel Computing Toolbox* and then it is more similar to the one described in the paper [1] and to a real life possible implementation. Obviously in this case a communication mechanism for the position coordinates of each node (thread) is necessary: this is realized by reading and writing files stored in a master server.
In fig. 1.2 the flow chart of the algorithm is showed to simplify its comprehension.

The initial phase regarding the generation of the workspace and framework is the same as in algorithm (A): it exploits the already presented functions *buildArea, generateAgents, genPolyGrid, gaussianDistr2d.* As before, instead of using the function *buildArea* to create the work area, it is possible to load it from an external file.

Then, after computing the initial Voronoi partition, inizializing the positions files and testing r/w permissions, the parallel section starts:

1. By means of the Matlab's function *spmd* a thread for each agent is created.

2. Each node reads all the others positions, computes its own Voronoi cell and the centroid (using utils *calculateCell, sliceCell* and *genMCgauss*), and updates the control action, which is the same of algorithm (A).

3. This control is mantained until the agent has reached the centroid (or better said, until its distance from the centroid is lower than a proper threshold).

4. While moving, each agent updates its position coordinates on the proper files.

5. When it reaches the centroid, it reads again the others positions, compute its Voronoi cell and centroid, updates the control action and so on (restart from point (2)).

After at most a given number of cycles (for each node), all the threads are synchronized using a barrier to update the graphical representation and to check if the convergence to the optimal solution has been reached. The latter is done calculating the differences between the actual and the previous centroids positions: if they are all below a certain value then the algorithm stops, otherwise we restart from point (1).

Reading the Matlab code, it is easy to see that before the parallel part, the **offline** algorithm is ran and its result is visualized in order to make a comparison between the 2 versions.

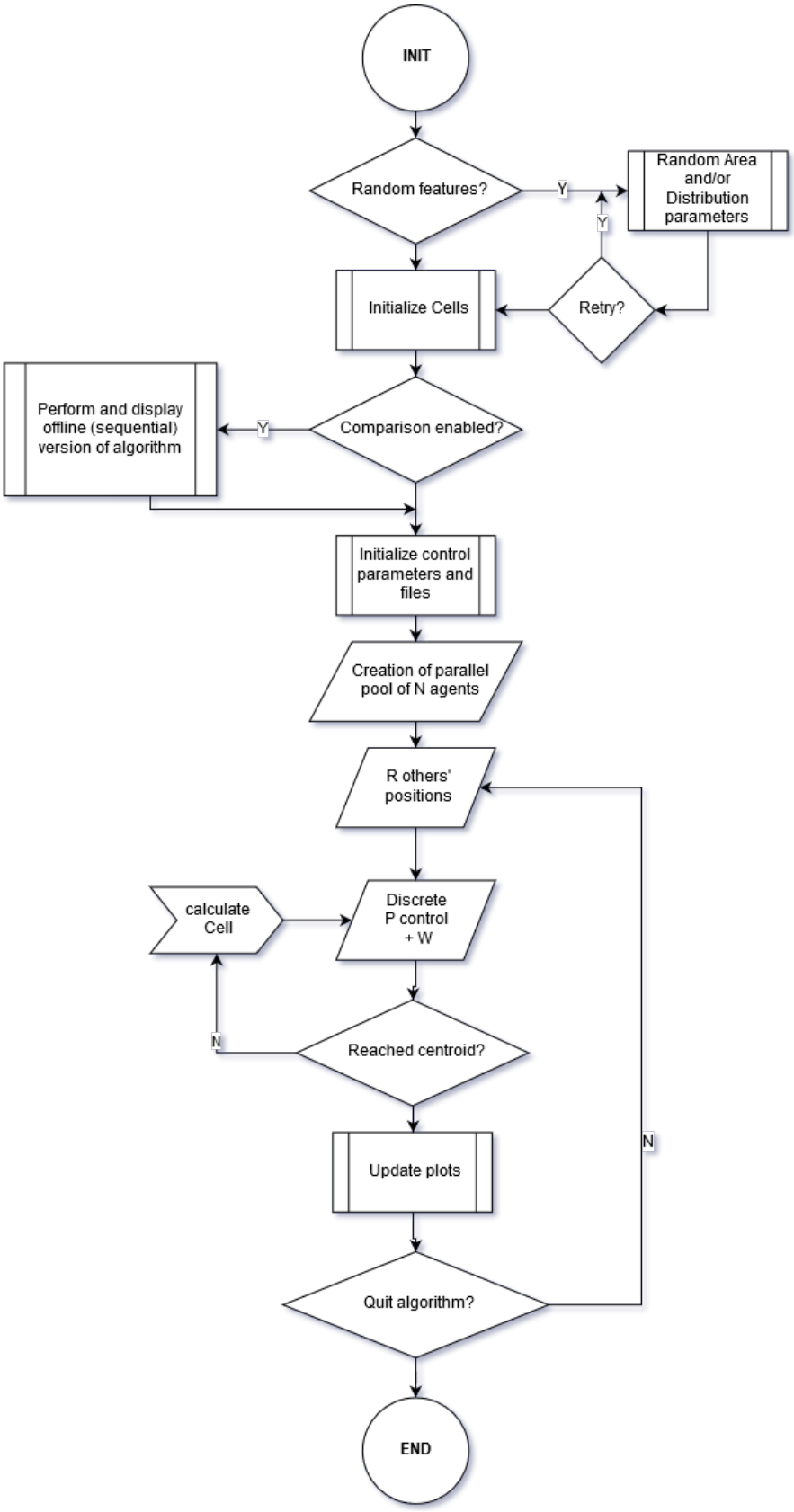

Figure 1.3: Algorithm B flow.

1.2.3 Utils

sliceCell

The need of this function arise from the fact that the *Voronoi* functions implemented in Matlab cannot be applied locally, as in our case. Moreover, the polygons identified by the same functions are quite difficult to manage. Thus, the aim of this function is to provide an alternative way, **geometrically based**, of computing local Voronoi partitions to those given by Matlab.
Geometrically based because it exploits the facts that the common edge between the Voronoi neighbours is obtained from the line passing through the midpoint between the two agents, orthogonal to the segment that links the agents (**perpendicular bisector**).
In simple terms, the function cuts (*slices*) the sensing region of the given agent according to the other agents present in the sensing area, in order to respect the definition of Voronoi partition.
This function takes as **inputs** the position of the two agents to analyze, the previous estimated Voronoi cell and the actual sensing radius.
Since the function deals with lines, slopes and y-intercepts, a *pathological case* should occur: two neighbours should share the same value of the y-coordinate, such that the slope of the seeked line will result infinite. In order to manage this problem, the case of points with the same y-coordinate is analyzed separately.
To make the explanation clearer, the instructions of the algorithm are outlined in the table below. For the sake of simplicity, only the general case with points with slope different from infinite is represented, since the reasoning is almost the same.

Algorithm 1 sliceCell

find the *perpendicular bisector* of the segment linking two agents
intersect the bisector with the allowed sensing area
if no intersections **then**
    return actual boundary area
else
    if the agent is below the bisector **then**
        collect every vertices of the previous cell that are *below* the bisector
    else
        collect every vertices of the previous cell that are *above* the bisector
compute the *convex hull* from the new vertices
return new boundary area
close

It is worth to underline that this function is called inside *calculateCell*, which is explained in the following section.

calculateCell

This function is the one that implements the *Adjust Sensing Radius Algorithm* presented in [1]. Its purpose is to compute the Voronoi cell of each agent, modifying the sensing radius in order to **converge** to a unique and correct solution. The function is slightly different in the two versions of the project: in the offline version (*ver. A*), as already mentioned, the function provides also the calculations of the **neighbours weights**; feature not present in the online version. As **inputs** it needs:

- the actual position of the agent;
- the initial radius (arbitrarily small);
- the total working area;
- (*only for the offline version*) the position and the status of all the agents;

The algorithm of the function can be summarized in the table below:

Algorithm 2 calculateCell

```
intersect the initial radius with the allowable area
loop:
if any other agent found then
    store ID and distance from the agent in exam inside a table
sort the table by increasing distance from the agent
if the table is not empty then
    run sliceCell for each agent in the table
compute the new estimated cell
if offline mode then
    check which agent among those in the table are actually neighbours
    assign weights to the neighbours
if the radius is smaller than twice the distance between the agent and
the furthest vertices of the cell then
    double the radius
    go to loop
else
    return the actual radius
    return the final cell
    if offline mode then
        return the weights of the neighbours
close
```

Further comments are present on the code.

genMCGauss

This function implements the computation of the **generalized centroid** of a given region. In other words, given a region, and the features of the Gaussian distribution function that we have used to model the variable to be sensed (mean coordinates and standard deviation), this function provides the generalized mass and the generalized centroid coordinates (explained in the theoretical notions section) of the considered region. This computation is done by a **quantization** of the region in a certain number of points and associating a generalized mass to each one of these points based on the associated density values. So as input it needs:

- the horizontal coordinate of the mean of the distribution;
- the vertical coordinate of the mean of the distribution;
- the standard deviation of the distribution;
- the so called *pointDensity* that express how much points we use to quantize a unitary area and then is related to the generalized mass associated to each point;

The outputs are:

- the horizontal coordinate of the generalized centroid;
- the vertical coordinate of the mean generalized centroid;
- the generalized mass;

Intuitively speaking this is the function that says to each agent **where it has to move**.

Chapter 2

Results analysis

2.1 Prerequisites

For the correct functioning of the simulations, the following Matlab Toolboxes and FEX files should be installed:

- Parallel Computing Toolbox
- Mapping Toolbox
- labelpoints, by Adam Danz
- Fast and Robust Curve Intersections, by Douglas Schwarz

2.2 Simulation benchmarks

In both approaches, several parameters can be adjusted that affects precision (and conversely, speed of simulations), namely:

- number of agents;
- query points density for distribution computation;
- proportional gain of the control;
- convergence threshold;
- etc.

In addition to that, the generation of the area to cover, of the agents and of the distribution can be set to randomly generate at each start of simulation (see fig. 2.1, fig. 2.2).

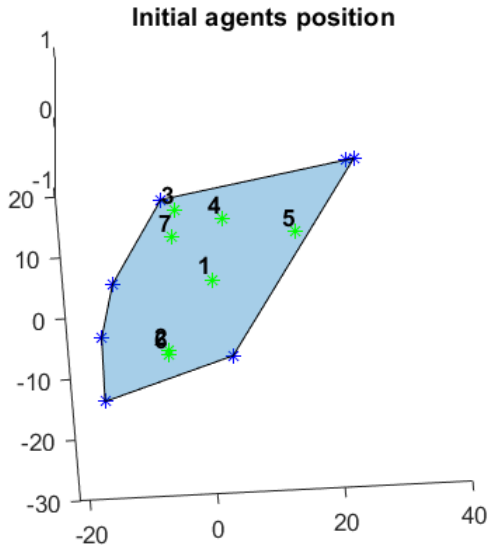

Figure 2.1: Random agents spawning.

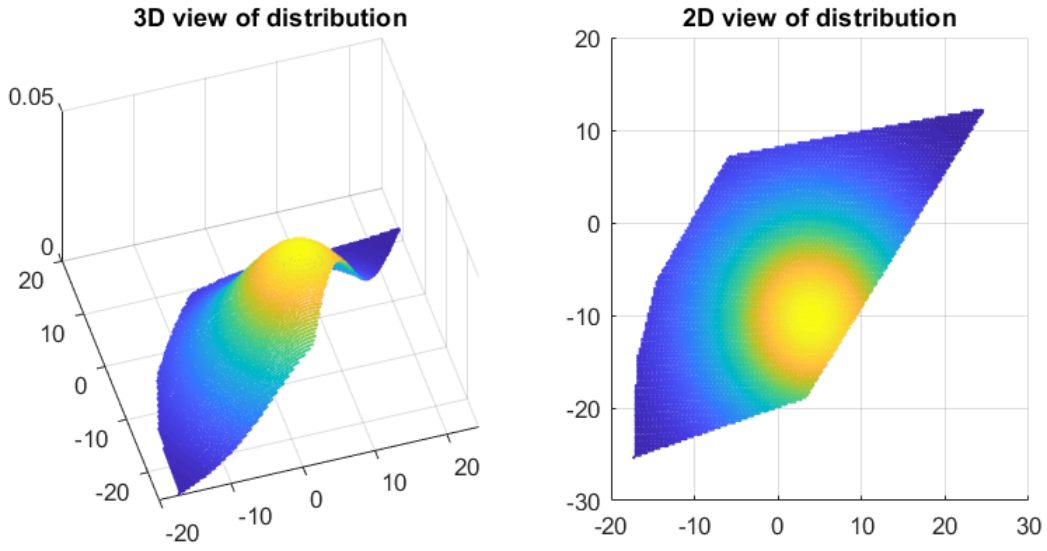

Figure 2.2: Random 2D normal distribution.

2.3 Offline algorithm (A)

The simulation converges with a good speed with few agents and with a reasonable one with more (see fig. 2.3, fig. 2.4); we can speculate that considering the typical speed of mobile robots the speed of every step is well suited.

As for correctness, the only issues encountered up to now is with very few agents (4-5) and a very large area (e.g. 10000 sq. unit), for which the cells tend to not be correct, because of the way "*Fast and Robust Curve Intersections*" works, i.e. by intersecting bounding boxes between the two desired geometrical entities.
Since the agents may be scattered around this large area, the rate of increase of the radius (that is, 2x at each successive step) bring the algorithm to find the bounding box of the radius to completely encompass the area to cover, thus not finding intersections at all; we believe that either a different increase rate of the radius, or better yet, a more efficient way to compute the Voronoi cell may solve the issue.

2.4 Online algorithm (B)

This algorithm is not much slower than the first version, especially for few agents.
It may be faster if implemented with a better communication between agents (instead of r/w files), but not by a large margin.

It is of interest to see how they both arrive to **similar results** but not exactly the same; this is due to the fact that while in the offline algorithm every agent "sense" the others before deciding the next control step, thus continuously updating the centroid, in the distributed version the centroid is updated only when the agent reaches it, making the trajectories of the centroids (and thus of the agents) slightly different.

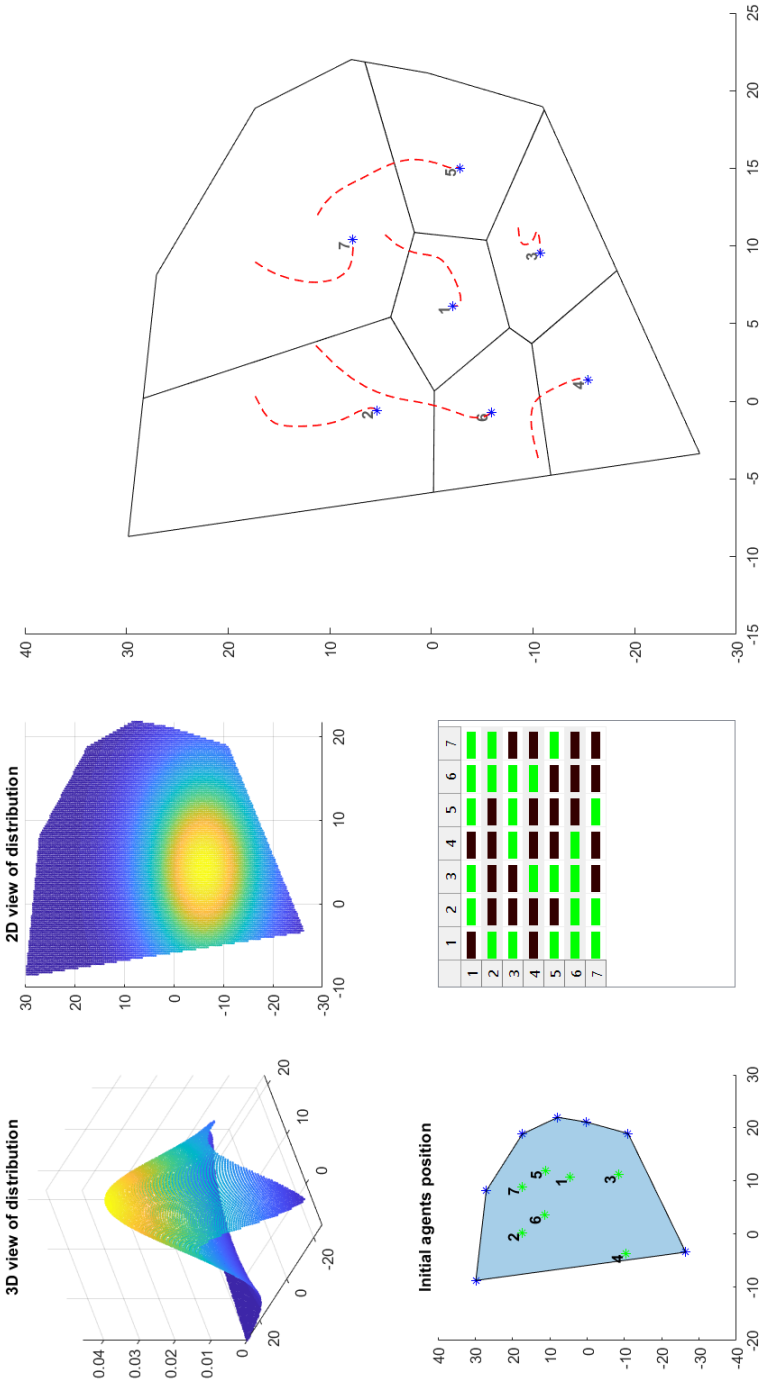

Figure 2.3: Results with 7 agents (ver. A).

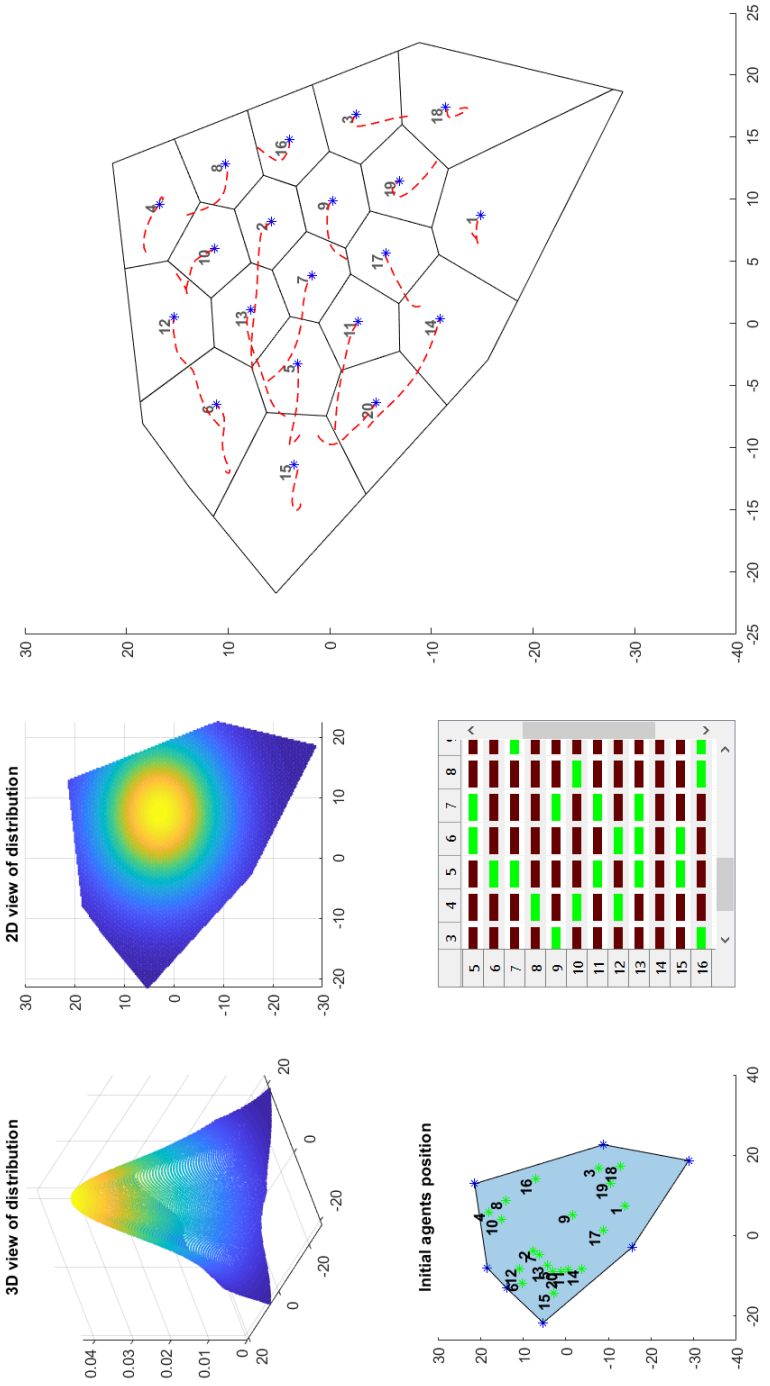

Figure 2.4: Results with 20 agents (ver. A).

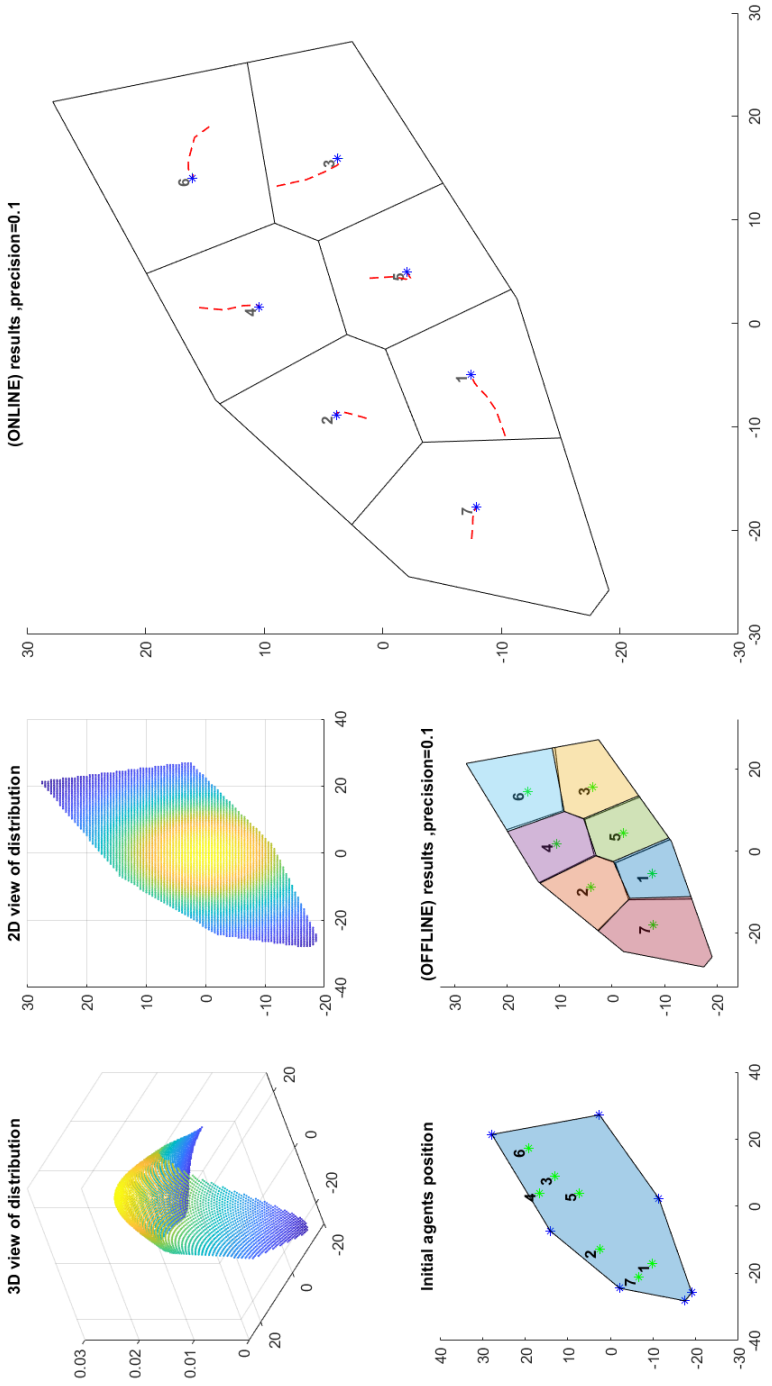

Figure 2.5: Results with 7 agents (ver. B).

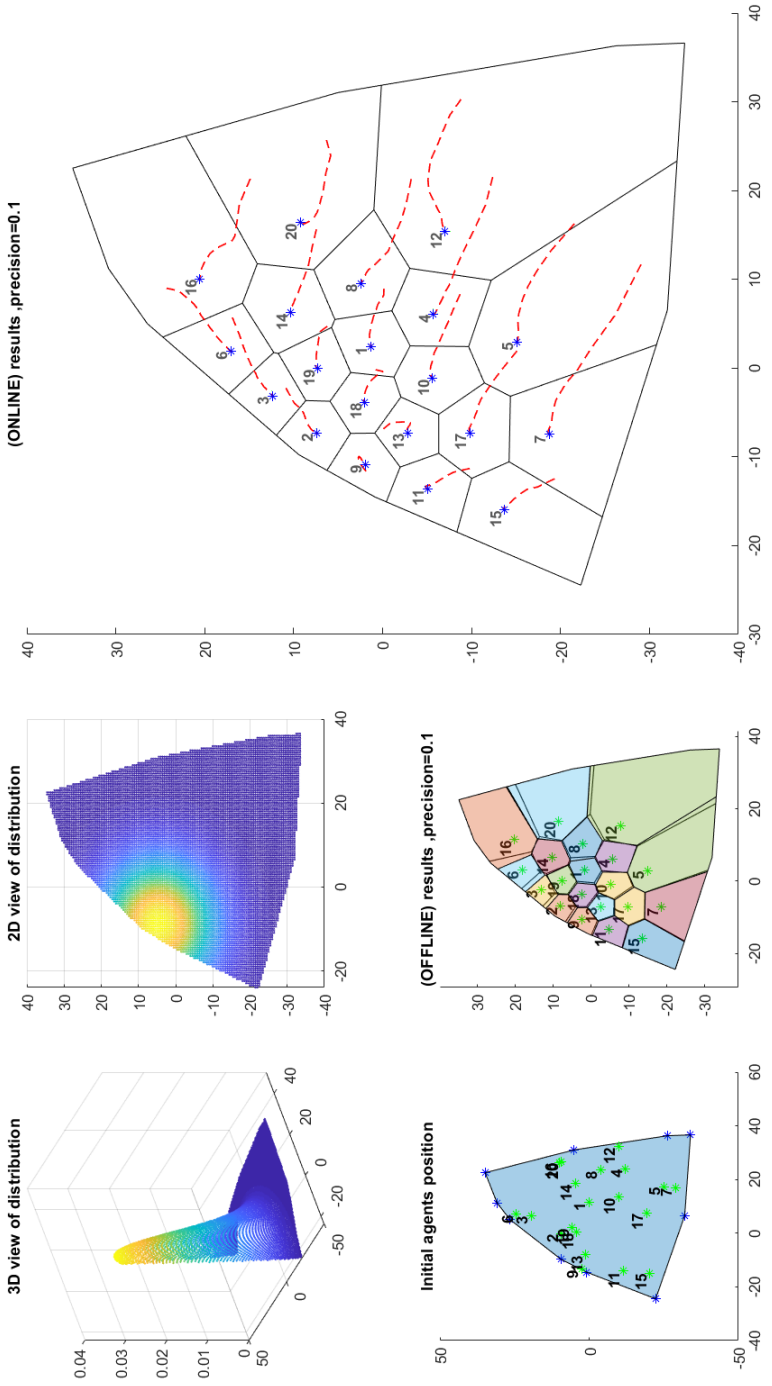

Figure 2.6: Results with 20 agents (ver. B).

Conclusions

The obtained results seem to be **consistent** with those to be expected from the main research paper, made exception for the timing and event control; we think that this work may be a good starting step to realize a real Coverage Control for Cooperative Multi-Robot Networks.

Some code optimization issues have been neglected from the efficiency point of view, nonetheless the execution times are kept **reasonably short**. We believe that the next logical step would be to fix this aforementioned speed issues, by e.g. finding a more efficient Voronoi cell computation, and to implement the timing and scheduling of different control parts on real platforms, which would be way easier and of more significance than only the simulation.

Finally, the challenge that this project bring to us made this work more and more exciting in seeking the best solution we could find. It's safe to say that we considered this work as a defiance in improving our knowledge about this specific field: especially, it was particularly appealing the autonomous research of new tools that could solve our issues.
Moreover, the main outcome of this project is that, taking into account all the constraints we met while we worked on it, the achieved goal has to be satisfactory and motivating for our next challenges.

Bibliography

[1] J. Cortes, S. Martinez, T. Karatas, F. Bullo. "Coverage Control for Mobile Sensing Networks" IEEE Transactions on Robotics and Automation, Vol. 20, No. 2, April 2004

[2] T. Hayes and F.H. Ali. "Mobile Wireless Sensor Networks: Applications and Routing Protocols" Handbook of Research on Next Generation Mobile Communications Systems. 2016.