POLITECNICO DI TORINO

Master's Degree
in MECHATRONIC ENGINEERING

Master's Degree Thesis

Navigation Algorithms for Unmanned Ground Vehicles in Precision Agriculture Applications

Supervisor
Prof. Marcello Chiaberge

Candidate
Jurgen Zoto

October 2018

Abstract

Robotics for agriculture can be considered a very recent application of one of the most ancient and important sectors, where the latest and most advanced innovations have been brought.
Over the years, thanks to continous improvement in mechanization and automation, crop output has extremely increased, enabling a large growth in population and enhancing the quality of life around the world. Both these factors, as a consequence, are leading to a higher demand for agriculture and forestry output.
Precision agriculture defined as the correct management of crops for increasing its productivity and maximizing harvest, is considered the answer to this issue. As a matter of fact, thanks to the development of portable sensors, the availability of satellite images and the use of drones, the collection of data is allowing a vast development in this field.
This thesis adresses in general robotics for agriculture in the form of a solution to be applied in order to improve robot mobility, in particular automated path planning in vineyards, by proposing a method to classify different parcels which make up the vineyard and to assign a precise task to the terrestrial unmanned robot.
The first part discusses how to generate a canopy segmentation from the mask obtained by processing images taken from unmanned aerial vehicles (UAVs). The developed algorithm is based on multiple steps: a first clustering of the mask is performed to identify each vine row, then a Least Squares regression is applied in order to be used in the following clustering step to detect each parcel which composes the map. Finally a recombination of the vine rows is carried out for the purpose of avoiding the problem of missing plants and defective rows.
The second part focuses its attention on the development of a path planning algorithm that can be integrated in every environment: it combines the A* search algorithm and path smoothing by exploiting the Gradient Descent algorithm.

The last part addresses the issue of applying the path generation in order to cover the desired parcel with the cooperation of both path planning and clustering.

Contents

Introduction

Overview and Motivations

Since the end of the second industrial revolution, robotics and automation have lead to significant improvements on agriculture with particular attention to:

- Precision agriculture, which could be defined according to [18] as "the site specific management of crops heterogeneity both at time and spatial scale in order to enhance the efficiency of agricultural inputs to increase yield, quality and sustainability of production"
- Auto-guidance on field crop machinery, which today can drive down a field with an accuracy unattainable by human drivers
- Machines that harvest fruits and vegetables for processing (e.g., tomato paste and orange juice).

In order to ensure an increase in productivity, scientific researches have now drawn their attention to the development of the next generation of sensing, mobility and manipulation technologies.
Sensing refers to the acquisition of different information such as crop temperature, humidity, pH, wetness, image, range, which precedes the combination and analysis of the data for specific aims. A possible application could be the vineyard detection from grey-scale unmanned aerial systems images, which can be used to extract features from the map in question [5].
Mobility describes several stages of vehicle automation that allows to optimize field coverage and minimizing fuel consumption without any human assistance (e.g. driverless tractors). More recently, auto-guidance has started to migrate to orchard vehicles as well, although due to poor satellite reception under thick canopies, automated vehicles need to be equipped with suitable devices that enable (semi-) autonomous seeding, spraying, mowing, weed removal, harvesting, and animal feeding, among other operations. Key element

in mobility is the sensor-based perception, provided by GPS/GNSS, inertial units, cameras, lidars, radars, etc. These sensors are not to be confused with those described in the previous paragraph, although there certainly are situations where a sensor can perform double duty of sensing for decision making and for navigation.
Manipulation is related to diverse actions carried out directly on the crop, including pruning, thinning, harvesting, fruit-gathering, etc.. In this case more advanced sensor-based perception technologies are necessary rather then mobility.
For long time the interests of agricultural research have been to follow well-defined traffic lanes with the purpose of minimizing damages on soil and plant growth. The introduction of automation and control technology has facilitated agricultural machine systems to follow paths spatially and temporally, especially by exploiting automated path planning to further optimize field work.

When robots have to achieve tasks that are too difficult to indicate the proper actions for all possible cases, it is necessary that they can perform themselves the most suitable solution to accomplish the task. In order to perform their action line adequately, robots require to think about the actions they are planning on executing, their future consequences and the side effects, whether they can be performed taking into account the different circumstances that may occur, and other situations. This requires that robots have an explicit representation of aspects of their environment to reason about. As a consequence it is necessary to know where the representation comes from, namely the generation and maintenance in real time of the environment, or at least some part of it based on past information collected by sensors, is an important aspect to take into account.

With Shakey the robot [21], robots started to be provided with reasoning capacities in order to make decisions about their own actions. This is why planning was one of the first research topics in Artificial Intelligience.
To design a planning system it is necessary to reach some targets by finding a solution to three main questions:

- World representation
- Actions representation
- Plan search process guidance

To answer these questions, the planning system should face the constraints imposed by the real world considering at the same time all the issues abovementioned.
Therefore, the work developed here aims to find a possible solution able to link the recognition of the environment with the path plan which the UGV will use to explore it and eventually to interact with it.

Objectives

In order to correctly design a navigation algorithm, the following objectives are fixed:

- Analyze the concepts and the already developed techniques for both path planning and agricultural field assessment.
- Define the principal characteristics of path planning algorithms
- Evaluate the used algorithm paying particular attention to the trade-off between safe trajectories and optimal solution in terms of journey and computational time.
- Develop an algorithm for identification of parcels starting from a previously processed digital image of the agricultural environment
- Implement path planning in the identified parcels
- Test the realized algorithm for different environment configurations, to assess the reliability of the method to give conclusions and suggestions for future development.

Chapter 1

Path Planning - Theory

The representation of the environment through a model is of fundamental importance for the development of various applications of mobile robot systems. Thanks to this, robot can adjust its decisions according to the world it is facing every second. To create environment models from sensor data there are various issues to deal with. Firstly the models must be such that they can be used effectively by other actors of the system, such as path planners. Second, the models should be developed as consistent as possible to the field of application, in this case the type of environment that is studied. This means that a general representation for mobile robots does not exist and each case requires a personalised approach.
Lastly, the adaption of uncertainty coming from both sensor data and the robot's state estimation system must be considered. Relevant meaning in the model representation and construction has the latter point, considering that sensor readings of distances are accumulated with respect to the same reference frame. Hence it is very likely that position estimates are affected by errors due to drift.
Historically, researchers' efforts were put on robots operating in indoor environments with the advantage that the world can be depicted as vertical structures on reference ground planes. As a result the world can be described as a two-dimensional (2-D) grid and uncertainty in the measurements and in the robot's pose can be modeled by means of probabilities of occupancy in the grid rather than binary occupied/empty flags [20].
With continued progress on different technological aspects of mobile robot systems such as sensing (e.g., threedimensional laser range scanners and stereo vision), and mechanical and controls, it became possible to develop robots for operations in unstructured, natural terrain. In these situations,

data cannot be properly applied in a 2-D grid, and environments need a significant magnitude of geometric elements. In most case where there is a reference ground plane, it is still correct a 2.5-D grid representation, in which each cell contains the elevation of the terrain at that location [16]. Despite their large usage, the main problems related to this approach are that it is not a compact representation and that it is difficult to integrate uncertainty in the representation. The most significant limitation of elevation maps has become more evident in recent years as applications of aerial data, which involves overhanging structures such as tree canopies. This has led to the development of three-dimensional (3-D) representations, such as point clouds, 3-D grids, and meshes, which increase the problem complexity with the introduction of the third dimension.
Agricultural researchers are nowadays working over 3-D structures and probabilistic representations of 3-D data. Because of the wide amount of data involved, it is necessary to process the data depending on the application. At a first stage, processing involves classifying the points into classes that are relevant to navigation tasks, such as disciminating between vegetation and ground, extraxting walls, tree surfaces, etc.. Then a second level of representation can be identified with the work of extracting part of the environments that are considered landmarks of interest (e.g. roads). Finally, the extraction and representation of the objects in the environments is carried out (e.g. natural obstacles).

In order to talk about motion planning, it is necessary to underline the hypothesis that the knowledge of a global and accurate map of the environment is crucial to develop consistent algorithms. Furthermore the considered system is a set of equations that does not explain exhaustively the entire physical system: the presence of uncertainties in the world or system modeling is not considered.
Such hypotheses are basically very strong. For this reason research in obstacle avoidance has been done in parallel in a realistic manner. The problem here is to consider simpler systems with respect to their geometric shape and integrate sensor-based motions to face the physical issues of a real system navigating in a real world.
Obstacle avoidance directs its attention to resolve the problem about navigating toward a goal in an unknown environment when the obstacles to evade have just been detected in real time.

The goal of this chapter is to deepen the aspects related to pathfinding,

once the environment model has been properly studied and chosen, and then to propose an algorithm to extract a continous path.

1.1 World and Terrain Models for Natural Environments

Occupancy grid maps, are a probabilistic approach to represent environment developed in the *1980s* by *Movarec* and *Elfes* [20]. They are an approximative technique where each cell of a discrete grid corresponds to the posterior probability that it is occupied by an obstacle. The advantage of occupancy grids is related to the fact that they do not depend on any predefined features and they provide the capability to represent unknown areas. The latter is of great importance in exploration tasks. Their disadvantage lies in possible discretization errors and the high memory requirements.
This solution cannot completely satisfy environment model requirements, such as information about terrain and elevation. From [28], where *Thrun* makes a full environment modeling analysis with emphasis on probabilistic techniques, it is possible to obtain a taxonomy in different directions. As a matter of fact, in order to be able to obtain a good environment model, it is necessary to introduce purely geometric models, such as elevation grid, 3-D grid, etc., and then add low level attributes in a cost map.
Assuming to represent the terrain as a funcion $h = f(x, y)$, where x and y are the coordinates of a reference place and h is the respective elevation. A natural representation is a digital *elevation map* which stores the value of h at discrete locations (x_i, y_i).
The most direct use of elevation maps is to compute *traversability costs* at each cell of the grid. A possible cost can be calculated, for example, by considering the local slope and 3-D texture of the terrain [6]. It is not easy task to express the exact relation between the costs and the elevation value stored in the grid. For this reason, recent researches has focused their attention on obtaining cost maps directly from observations.
An example of such a combination of grid representation and dynamic planner is the A* algorithm, which will be treated in depth in continuing this chapter.

1.2 Pathfinding

Many problems in robotics deal with finding a path through a graph. A simple example of such problems can be navigation through a maze. They have usually been set about in two possible ways, called in [10] *mathematical* and *heuristic* approach. The mathematical typically deals with the properties of abstract graphs and with algorithms that assign an regular analysis of nodes of a graph in order to compute a minimum cost path.
The latter typically uses special knowledge about the domain of the problem to be able to improve the computation efficiency of solutions of graph-searching problems. The idea developed by *Hart*, *Nilsson* and *Raphael* in their work is to use "together the above two approaches by describing how information from a problem domain can be incorporated in a formal mathematical approach to a graph analysis problem. It also presents a general algorithm which prescribes how to use such information to find a minimum cost path through a graph. Finally, it proves, under mild assumptions, that this algorithm is optimal in the sense that it examines the smallest number of nodes necessary to guarantee a minimum cost solution".

1.3 Algorithm for Finding Minimum Cost Paths

The aim is to find an optimal path from the *start* to a *goal* node. From the starting node, the algorithm will generate some part of the subgraph, applying repeatedly the successor operator Γ. If Γ is applied to a node, then it has been *expanded*. To collect the minimum cost path from *start* to each node encountered, each time the node is expanded, each successor node n is stored with both the cost of getting to it by the lowest cost path encountered until then and a pointer to the predecessor of n along that path. At some stage the algorithm ends at some goal node t, and no more nodes are expanded. It is now possible to reconstruct a minimum cost path from *start* to t simply by retracing the steps through the pointers.
The algorithm is defined *admissible* if an optimal path is ensured for any δ graph. In the next section, the A* algorithm will be proposed and demonstrated that, under mild assumption, it uses the information contained in the graph in an optimal way, expanding the smallest number of nodes needed to guarantee finding an optimal path.

1.3.1 A* - Algorithm Description

In order to expand the fewest possible nodes, it is necessary that the search algorithm chooses constantly and properly the next node to expand, in such a way that as less time as possible is devoted to wasting efforts.
At the same time, if it continues to ignore nodes that might be on an optimal path, it will sometimes not be able to find such a path becoming not admissible. To help in the decision of the next node to be expanded an *evaluation function* $\hat{f}(n)$ is computed for any node n, so that the available node having the smallest value $\hat{f}$ is the node to be expanded next. The search algorithm can be described as follows.

Search Algorithm A*

1. Mark *start* "open" and calculate $\hat{f}(start)$
2. Select the open node n whose value of $\hat{f}$ is smallest. Resolve ties arbitrarily, but always in favour of any node $n \in T$
3. If $n \in T$, mark n "closed" and terminate the algorithm
4. Otherwise, mark n closed and apply the successor operator Γ to n. Calculate $\hat{f}$ for each successor of n and mark as open each successor not already marked closed. Remark as open any closed node n and for each $\hat{f}(n_i)$ is smaller now than it was when n, was marked closed. Go to step 2.

The Evaluation Function

For any subgraph G_s and any goal set T, let $f(n)$ be the actual cost of an optimal path *constrained to go through* n, from *start* to a preferred goal node of *start*. Note that $f(s) = h(s)$ is the cost of an unconstrained optimal path from *start* to a preferred goal node of *start*. In fact, $f(n) - f(start)$ for every node n on an optimal path, and $f(n) > f(s)$ for every node n not on an optimal path. Therefore, although $f(n)$ is not known a priori, it seems reasonable to use an estimate of $f(n)$ as the evaluation function $\hat{f}(n)$. It is possible to write $f(n)$ as follows:

$$f(n) = g(n) + h(n) \tag{1.1}$$

where $g(n)$ is the actual cost of an optimal path from *start* to n, and $h(n)$ is actual cost of an optimal path from n to a preferred gole node of n. If g and h are known, they can be added to form an estimate of f. Let $\hat{g}(n)$ be an estimate of $g(n)$. An obvious choice for $g(n)$ is the cost of the path from *start* to n having the smallest cost so far found by the algorithm.
Consider the subgraph shown in figure 1.1. It consists of a start node *start* and three nodes, n_1, n_2 and n_3. The arcs are shwon with arrowheads and costs. Let us trace how algorithm A* proceeded in generating this subgraph. Starting with *start*, n_1 and n_2 successor are obtained. The estimates $\hat{g}(n_1)$ and $\hat{g}(n_2)$ are then 3 and 7, respectively.
Suppose A* expands n_1 next with successors n_2 and n_3. At this stage $\hat{g}(n_3) = 3 + 2 = 5$, and $\hat{g}(n_3)$ is lowered to $3 + 3 = 6$ because a less costly path to it has been found. The value of $\hat{g}(n_1)$ remains equal to 3. Next an extimate $\hat{h}(n)$ of $h(n)$ must be computed. It depends on information coming from the problem domain, which usually consists of finding a minimum cost path through a graph with "physical" information.

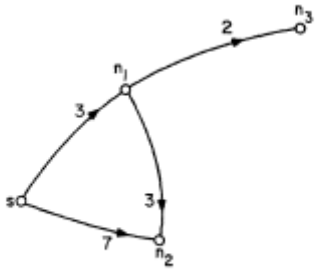


Figure 1.1

Limitation of Subgraphs by Information from the Problem

Often the information about the constraints of the set of possible subgraphs to each node is given.
When a real problem is modeled by a graph, each node of the graph corresponds to some state in the problem domain. The algorithm A * is actually a family of algorithms; the choice of a particular function $\hat{h}$ selects a particular algorithm from the family. The function $\hat{h}$ can be used to tailor A* for particular applications.
The choice $\hat{h} = 0$ corresponds to the case of knowing, or at least of using,

absolutely no information from the problem domain. Assuming to have a graph which models the connection between cities with roads, this would correspond to assuming a priori that any city may be an arbitrarily small road distance from any other city regardless of their geographic coordinates. Knowing the nature of Euclidean space, the information about $\hat{h}(n)$ is increased from 0 to $\sqrt{x^2 + y^2}$ (where x and y are the magnitudes of the differences in the x, y coordinates of the node n and its closest goal). The algorithm would then still find the shortest path, but would do so by expanding, typically, considerably fewer nodes. In fact, A * expands no more nodes than any admissible algorithm that uses no more information from the problem domain.
In general the h function must be an admissible heuristic, therefore it must not overestimate the distance to the goal. For example, for application of pathfinding, it might represent the straight-line distance to the goal, that is physically the smallest possible distance between any two points or nodes [22].
The time complexity of the algorithm depends on the heuristic function h. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$\left|h(n) - \hat{h}(n)\right| = \mathcal{O}(\log \hat{h}(n)) \tag{1.2}$$

This means that the error of h will not grow faster than the logarithm of $\hat{h}$ that returns the true distance from the node to the goal.

1.3.2 A* - Properties

To summarize A* has the following properties:

- It is complete: it will always find a solution if it exists
- It can use a heuristic to significantly speed up the process
- It can have variable node to node movement costs. This enables things like certain nodes or paths being more difficult to traverse.
- It can search in many different directions if desired

1.3.3 A* - Pseudocode

In 1.1 is described the pseudocode of the A* algorithm presented in 1.3.1, while the entire code can be found in Appendix A.
The goal node is denoted by node_goal and the source node is denoted by node_start. There are two lists: OPEN and CLOSED.
OPEN consists on nodes that have been visited but not expanded, meaning that successors have not been explored yet. This is the list of pending tasks.
CLOSED consists on nodes that have been visited and expanded (successors have been explored already and included in the open list).

1.4 Trajectory Modification

In the previous section there has been talk about finding paths. The objective now is to transform these paths into actual motion commands, in particular it will be proposed an algorithm to generate smooth paths.
Considering that the planning takes place in a discrete world, having a path like in figure 1.2 has lots of disadvantages. In fact, due to their mechanical constraints it is not recommended in robots to take 90° or in general a sharp turn. For this reason the actual motion requires that the robot stops, makes

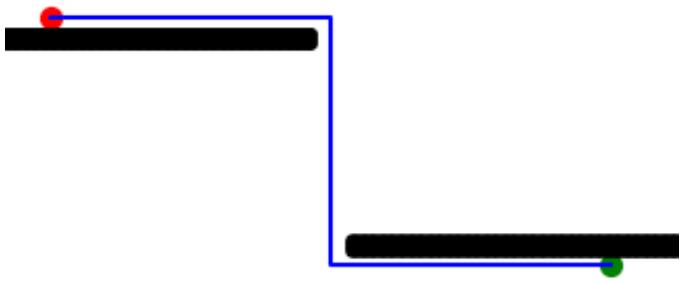


Figure 1.2

a turn and then goes again. The question is whether is it possible to generate a smooth path starting for example from the blue line in figure 1.2.
The path generated in blue is specified as a sequence of points, each of which is defined as X_i with 2 dimensional coordinates.

Algorithm 1.1 A* search

1: Put **node_start** in the **OPEN** list with $f(node\_start) \leftarrow h(node\_start)$
2: **while OPEN** list not empty **do**
3: Take from the **OPEN** list the node **node_current** with the lowest $f(node\_current) \leftarrow g(node\_current) + h(node\_current)$
4: **if node_current** = **node_goal then**
5: Solution found
6: **break**
7: **end if**
8: Generate each state **node_successor** that comes after **node_current**
9: **for node_successor** of **node_current do**
10: $successor\_current\_cost \leftarrow g(node\_current) + w(node\_current, node\_successor)$
11: **if node_successor** is in **OPEN** list **then**
12: **if** $g(node\_successor) \leq successor\_current\_cost$ **then**
13: **continue** to line 26
14: **end if**
15: **else if node_successor** is in **CLOSED** list **then**
16: **if** $g(node\_successor) \leq successor\_current\_cost$ **then**
17: **continue** to line 26
18: **end if**
19: Move **node_successor** from the **CLOSED** list to the **OPEN** list
20: **else**
21: Add **node_successor** to the **OPEN** list
22: $h(node\_successor) \leftarrow$ heuristic distance to **node_goal**
23: **end if**
24: $g(node\_successor) \leftarrow successor\_current\_cost$
25: Set the parent of **node_successor** to **node_current**
26: **end for**
27: Add **node_current** to the **CLOSED** list
28: **end while**
29: **if node_current** $\neq$ **node_goal then**
30: exit with error the **OPEN** list is empty
31: **end if**

Smoothing Algorithm - Description

Initially the variable Y_i are created with the same value of X_i 1.3, which represent the non-smooth locations that the planner has found.

$$Y_i = X_i \tag{1.3}$$

Then two criteria are assumed to be minimized: in the first the error of the *i-th* original point with the *i-th* smooth point,

$$\min[(X_i - Y_i)^2] \tag{1.4}$$

while in the second the distance between consecutive smooth points.

$$\min[(Y_i - Y_{i+1})^2] \tag{1.5}$$

The minimization has no effect if only the first criterion is applied, in fact in such a way we obtain again the origal path. On the other hand no path is obtained if only the second criterion is considered: it asks that all the Y_i are as similar as possible, which means that a single point is obtained if all the Y_i are the same. It is easy to note that these two criteria are in conflict to each other. By minimizing both at the same time a sort of weigth α is introduced in 1.5, which smooths the path in accordance to its value: the stronger it is, the smoother the path is. On the contrary, the smaller α, the more the original path is retained.

To optimize the two criteria the idea is to apply the *Gradient Descent*, which is a *first-order iterative optimization algorithm* for finding a minimum of a function.
It is based on the observation that if the multivariable function $F(x)$ is *defined* and *differentiable* in a neighborhood of a point a, then $F(x)$ decreases *fastest* if one goes from a in the direction of the negative gradient of F at a, $-\nabla F(a)$ as shown in figure 1.3. It follows that, if

$$a_{n+1} = a_n - \gamma \nabla F(a_n) \tag{1.6}$$

For γ small enough, the $F(a_n) \geq F(a_{n+1})$. With certain assumptions of the function F and particular choices of γ convergence to a local minimum

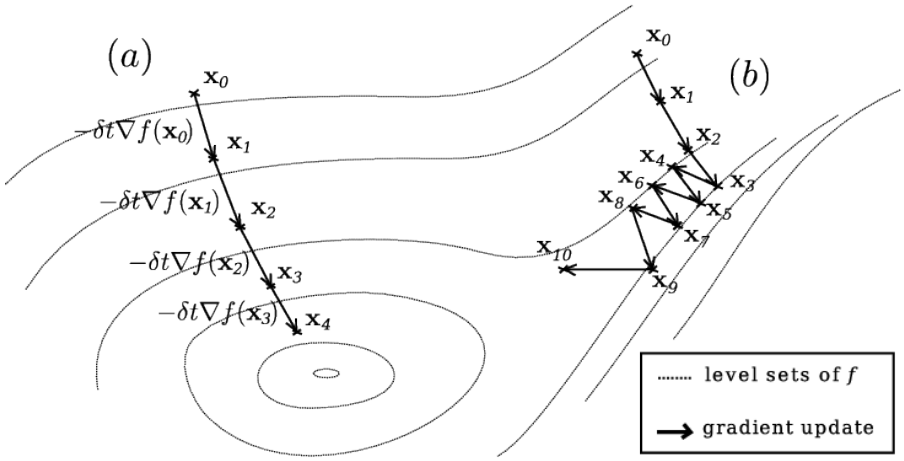


Figure 1.3

can be guaranteed. In the case treated in this section, the function to be minimized with 1.6

$$Y_i = \alpha(X_i - Y_i)^2 + \beta(Y_i - Y_{i+1})^2 + \beta(Y_i - Y_{i-1})^2 \tag{1.7}$$

In this function the previous Y_{i-1} and the following Y_{i+1} position are taken into account to determine the next actual position of Y_i. By applying the *Gradient Descent* the following function is obtained

$$Y_i = Y_i + \alpha(X_i - Y_i) + \beta(Y_{i+1} + Y_{i-1} - 2Y_i) \tag{1.8}$$

Where $\alpha(X_i - Y_i)$ means that a small step is always taken in the direction of minimizing the error in 1.4.
In the last term of the equation $\beta(Y_{i+1} + Y_{i-1} - 2Y_i)$, the old Y_i variable is retained moving slightly in the direction of Y_{i+1} and Y_{i-1} away from Y_i by combining the step on the left and on the right and realizing that this occurs twice. As a matter of fact, through the optimization Y_i should be as close to Y_{i-1} and simultaneously be as close to Y_{i+1}.
In Appendix A can be found the code relative to the path smoothing which

is explained in the pseudocode 1.2.

In figure 1.4 are represented some significant examples of trajectory modi-

Algorithm 1.2 Path smoothing

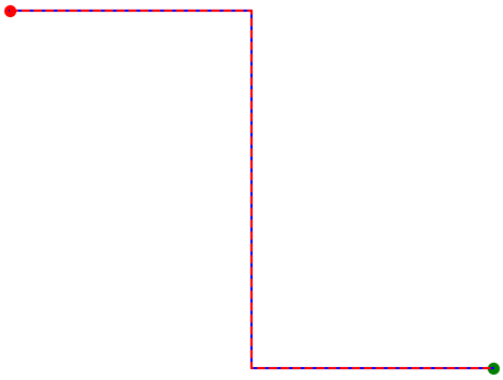
newpath ← **path**
change ← **tolerance**
while change ≥ **tolerance do**
  change ← 0
  for Y_i of **newpath** and X_i of **path do**
    $v \leftarrow Y_i$
    $Y_i \leftarrow Y_i + \alpha(X_i - Y_i) + \beta(Y_{i+1} + Y_{i-1} - 2Y_i)$
    change ← $|v - Y_i|$
  end for
end while

fication obtained tuning the weight values α and β.

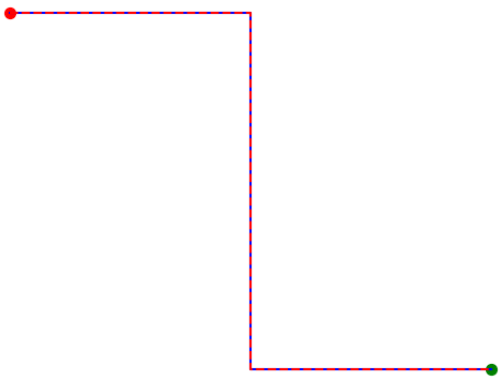
It can be noted that in figure 1.5 the algorithm behaves strangely, as proof of the fact that a tuning did before is necessary in order to avoid such performance. This problem is linked to the optimization of both the criteria 1.4 and 1.5.

For instance, in figure 1.6 it is possible to see the algorithm applied to the figure 1.2 with parameter α and β set both to 0.5.

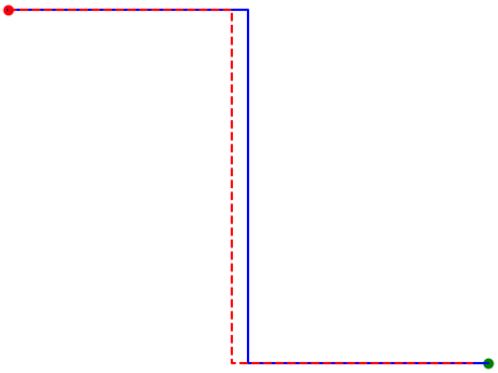
In the next chapter the path planning algorithm will be exploited and customized in order to obtain an adequate result for our applications.



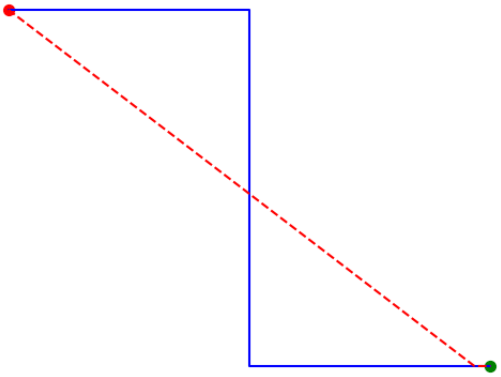
(a) $\alpha = 0,\ \beta = 0$



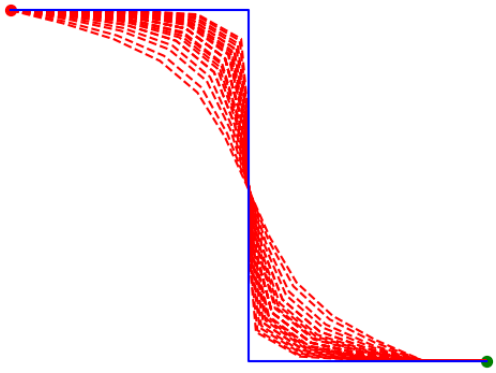
(b) $\alpha = 1,\ \beta = 0$



(c) $\alpha = 1,\ \beta = 1$



(d) $\alpha = 0,\ \beta = 0.5$



(e) $0.1 \leq \alpha \leq 0.5,\ 0.1 \leq \beta \leq 0.5$

Figure 1.4

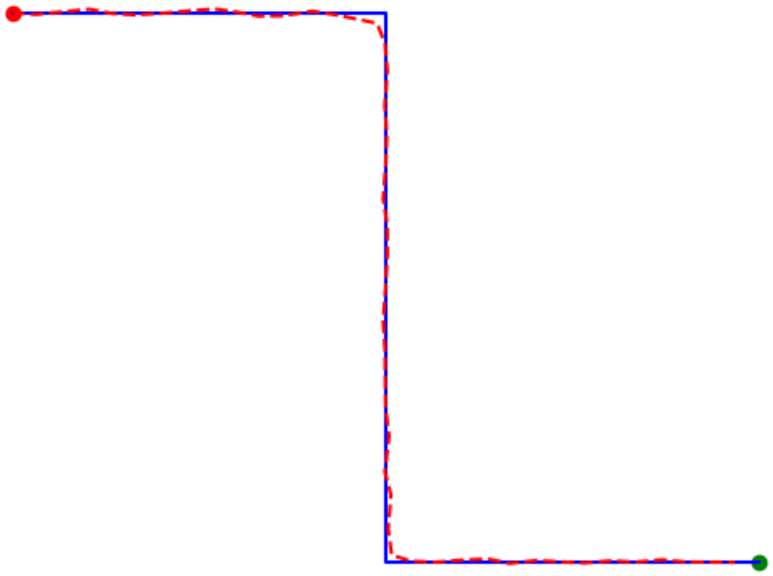


Figure 1.5: $\alpha = 0.8,\ \beta = 0.6$

Figure 1.6

Chapter 2

Path Planning - Application

In this chapter some changes will be applied to the *Path Planning* algorithms, in order to deal with real world situations, both for indoor and for outdoor environments.

All the algorithms were built using Python 2.7.15 and performed on an Intel(R) Core(TM) i7-4700U CPU at 2.40 GHz with 8 GB of RAM memory.

2.1 Indoor Navigation

The idea is to generate a path to be followed in such a way that the robot does not collide with an obstacle of the environment.
Starting from the assumption that the provided map is of good quality, which also means that the map has a good resolution, it is possible also to define a more restrictive and safer path, although it cannot be always optimal in terms of covered distance.
The figure 1.2 shows that no constraints were applied to the robot in terms of distance from obstacles, and as a consequence, when the path is smoothed it is necessary to take into account a possible collision with obstacles as in figure 2.1. Some tactics can be adopted in order to improve the robot behaviour:

1. Add more *movements* in the search algorithm
2. Set a safety distance from the obstacles
3. Set appropriate values of the parameters α and β.

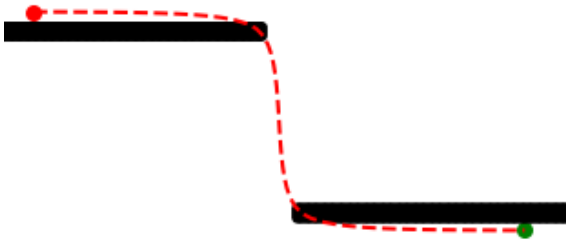


Figure 2.1

To add more *movements* means to have a more dense network of connections for each node, which can be translated in our domain as more freedom of movement. At the same time extending the range of proximity the algorithm will check if all the connections of the node are *expandable*, imposing consequently a higher safety distance.

```
1  a = self.robotx
2  b = self.roboty
3  for i in range(len(delta)):
4    x2 = x + self.step * delta[i][0]
5    y2 = y + self.step * delta[i][1]
6    if x2 >= 0 and x2 < self.dimx and y2 >= 0 and y2 < self.dimy:
7      if x2 - a >= 0 and x2 + a < self.dimx and y2 - b >= 0 and y2 + b < self.dimy:
8        truth_table = numpy.array(self.grid[x2 - a: x2 + a, y2 - b: y2 + b] == 0)
9        obstacle = False
10       if not truth_table.all():
11         closed[x2][y2] = 1
12   else:
13     obstacle = True
14     closed[x2][y2] = 1
```

These adjustments have been implemented in the code above. The variable *delta* represents all the possible directions that the robot can take, while in lines 4 and 5 *self.step* is used to increase robot movement in all the directions. From line 7 to 14 the algorithm checks whether the robot can find obstacles in its path considering its dimensions defined in lines 1 and 2.

It is important to underline that, in order to improve time performances of the algorithm, the *Numpy* library has played a crucial role. As a matter of fact, lines 8 and 10 have substituted many lines of code.

2.1.1 The Work Environment

To develop and test the algorithm, firstly a simulated indoor environment has been created in the rooms of Politecnico di Torino; to represent a generic maze some polystyrene panels of dimension 1m x 0.5m x 10cm have been exploited as shown in fig. 3.2, then the map was generated using the TurtleBot3 Waffle (fig.2.3), a small, programmable, ROS-based mobile robot able to map the environment using a 360 Laser Distance Sensor LDS-01 (fig.2.4), a 2D laser scanner capable of sensing 360 degrees that collects a set of data around the robot.
In Table 2.2 and Table 2.2 are reported respectively the specifications of the TurtleBot3 Waffle and the LDS sensor.

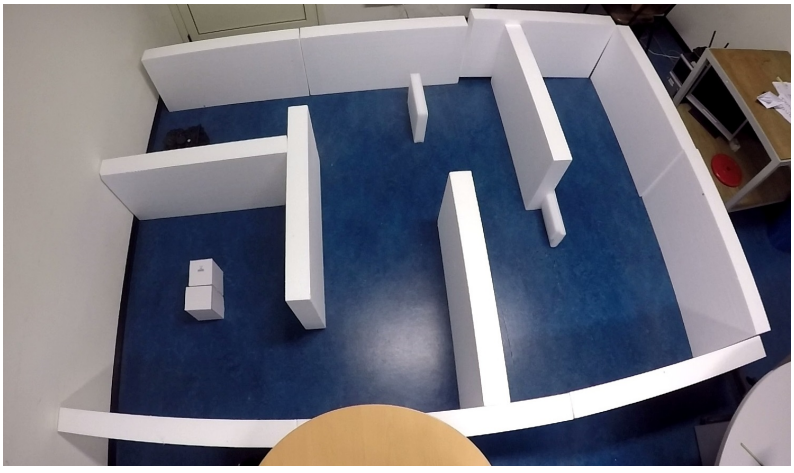


Figure 2.2: Environment test for *Path Planning*

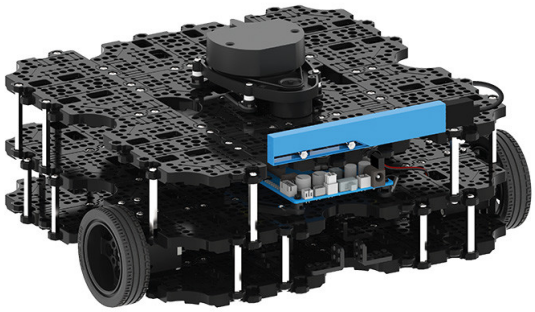


Figure 2.3: The TurtleBot3 Waffle

Figure 2.4: 360 Laser Distance Sensor LDS-01

Items	**Specifications**
Operating supply voltage	5V DC ±5%
Light source	Semiconductor Laser Diode(λ=785nm)
LASER safety	IEC60825-1 Class 1
Current consumption	400mA or less (Rush current 1A)
Detection distance	120mm 3,500mm
Interface	3.3V USART (230,400 bps) 42bytes per 6 degrees
Ambient Light Resistance	10,000 lux or less
Sampling Rate	1.8 kHz
Dimensions	69.5(W) X 95.5(D) X 39.5(H)mm
Mass	Under 125g

Table 2.1: Laser Distance Sensor specifications

Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1.8kg
Threshold of climbing	10 mm or lower
Expected operating time	2h
Expected charging time	2h 30m
SBC (Single Board Computers)	Intel® Joule™ 570x
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Actuator	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01
Camera	Intel® Realsense™ R200
IMU	3 Axis Accelerometer 3 Axis Gyroscope 3 Axis Magnetometer
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A
Expansion pins	GPIO 18 pins Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3
Programmable LEDs	User LED x 4
Status LEDs	Board status LED Arduino LED Power LED
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh/19.98Wh 5C
PC connection	USB
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Table 2.2: TurtleBot3 Hardware specifications

2.1.2 Path Plan

From figure 2.2 it has been extracted the digital map in figure 2.5.

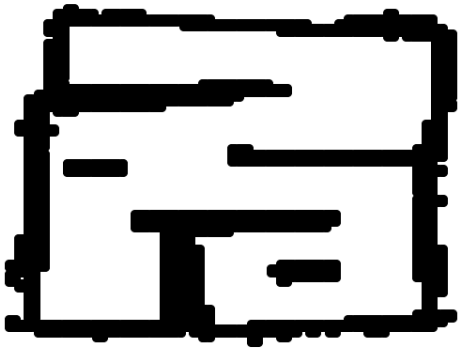
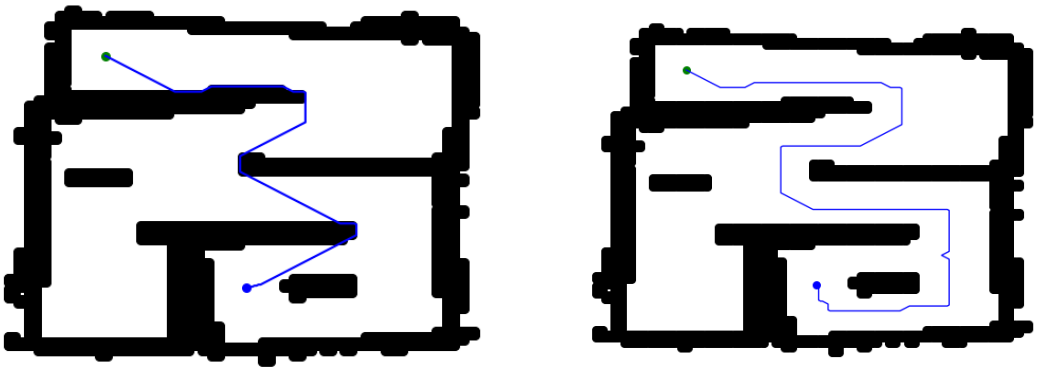


Figure 2.5

The next step is to test if it is possible to find a path given arbitrary initial and ending points. Figure 2.6(a) represents the *raw* path of the robot, however it can be noticed that it cannot be a possible solution due to its proximity to the obstacles. For this reason, as explained previously, a safety distance has been integrated in the code. The result can be seen in the 2.6(b), where the blue line always keeps a distance greater than robots dimensions from the obstacles.



(a) Unsafe path estimation (b) Safe path estimation

Figure 2.6

Once the path has been estimated, it is necessary to smooth it in order to

allow the robot to cover an easier trajectory. The solution proposed in figure 2.7 has parameters value $\alpha = 0.2$ and $\beta = 0.8$, which by trial and error best satisfies the requirements, imposing a safety distance to the robot.

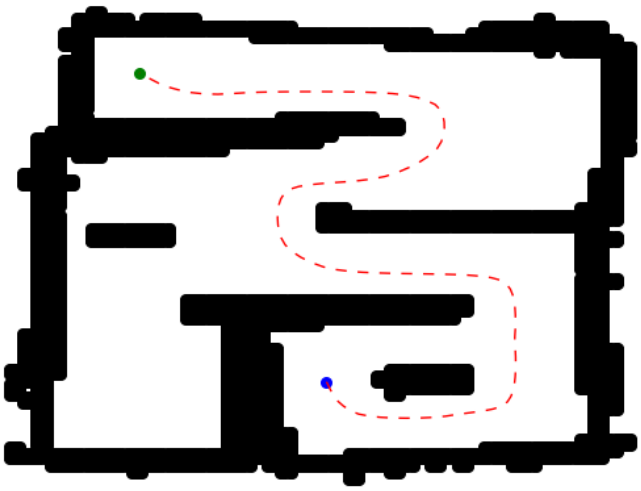


Figure 2.7: Final path plan

Chapter 3

Rows Clustering

Unmanned aerial vehicles (UAVs) have recently begun being applied to precision agriculture. This is a new development and results are still preliminary, albeit very promising and capable to supplement or even substitute satellites and aircrafts in agriculture remoting sense [12], namely collecting visual and sensors data about vigour of the plantation, canopy, water and plant stress in order to assess the field condition through photogrammetry.
The purpose of this chapter is not to deal with raw images taken from UAVs, but with previously elaborated ones [5] [24], in order to detect and select different parcels belonging to the same field.
All the algorithms were built using Python 2.7.15 and performed on an Intel(R) Core(TM) i7-4700U CPU at 2.40 GHz with 8 GB of RAM memory.

3.1 Introduction

The reason behind this algorithm is to create a way of automatically identify from a piece of land specific indipendent parcels, in which the Unmanned Ground Vehicle (UGV) can navigate in accordance with the user's wishes.

Since the vegetation type explored by the drone can be any, a generic approach has been followed to develop the algorithm. However it is necessary to have a pre-processed map where all the vegetation is defined with a digital value, i.e. everything that represents vegetation is coloured by a black pixel, whereas free space corresponds to a white pixel.
There are no constraints related to map resolution, except the fact that the vegetation must be distinguishable with respect to the free space where the

robot can move.

3.2 The Workflow

In order to better elucidate the steps performed by the algorithm, the following workflow has been created.

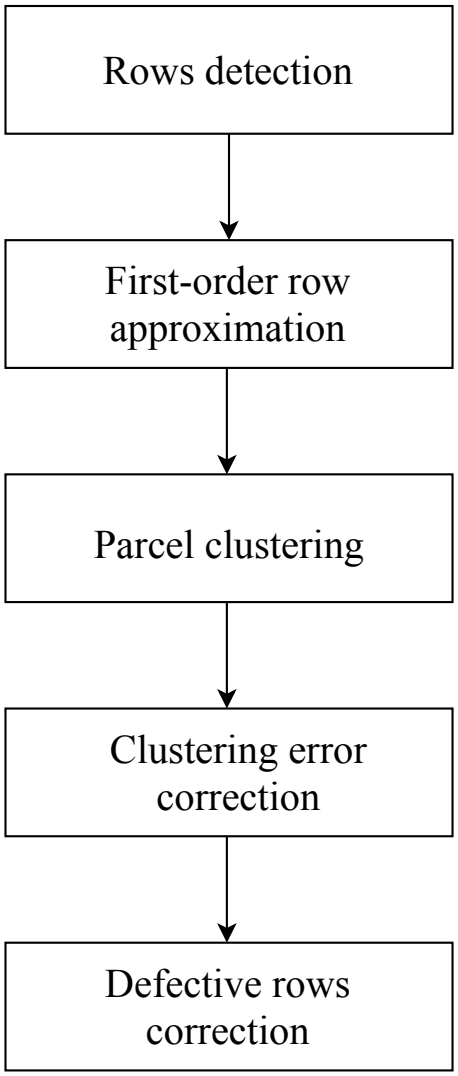


Figure 3.1: Algorithm Workflow

3.3 The Work Environment

To develop and test the algorithm, firstly various simulated environments have been created in the rooms of Politecnico di Torino; to represent a generic canopy some polystyrene panels of dimension 1m x 0.5m x 10cm have been exploited as shown in fig. 3.2, then the map was generated using the TurtleBot3 Waffle (fig.2.3), a small, programmable, ROS-based mobile robot able to map the environment using a 360 Laser Distance Sensor LDS-01 (fig.2.4), a 2D laser scanner capable of sensing 360 degrees that collects a set of data around the robot.
In Table 2.2 and Table 2.2 are reported respectively the specifications of the TurtleBot3 Waffle and the LDS sensor.

Figure 3.2: Example of simulation environment test

3.4 Rows Detection

The first part of the algorithm deals with the detection of the rows that compose the vegetation represented by binary image, in which to the potential row pixels is assigned 1 and the background pixels is assigned 0.

In this way, along the image entensive groups of "1s" representing the canopy rows, while the "0s" the background can be identified.
Each group of interconnected "1s" is called *cluster*. The task of recognizing and labelling the interconnected pixels is performed through the first developed algorithm. The clusters are labelled with an increasing number, which when the algorithm ends identifies also the number of clusters.
If all the clusters are correctly separated each of them should represent one canopy row.

3.4.1 *Findrows* Algorithm

The algorithm is based on the idea of visiting all the matrix map using a *sliding window*: starting from the first pixel in the upper left to the last pixel on the bottom right. The *sliding window* is defined with a proper width called ρ, which depends on the type of vegetation we are interested to deal with.
In a first moment the window will translate only along an horizontal line until a black pixel is found. Then the search in all the directions of contigous black pixels starts, and when a new black pixel is found, it becomes a *visited* pixel. It will fill a list called OPEN, which includes all that pixels *visited* but not yet *expanded*.
A matrix called CLOSED is exploited, to represent the map as a *False Positive*, where all background points are identified as foreground, and vice versa. As the map is explored this matrix will be filled with "1s", helping the algorithm in two main purposes:

1. The same pixel cannot belong to two different clusters

2. The algorithm, as a consequence, will not visit the same pixel again terminating its operation in less time

The output of this first step will be a list of labeled clusters, which will be widely used in the next computations.
As can be noticed, this algorithm requires a threshold value for ρ which is set in order to take into account the possibility of having different distances in accordance to the plantation, as can be easily guessed for example talking about cornfields, where corn rows are closer to each other with respect to vineyards, where vine rows are much more distant. Due to this reason it is very important, before clustering, to choose the right value of the *sliding window*'s width.

In 3.1 is proposed the pseudocode of the algorithm in order to better understand the code steps in Appendix B.

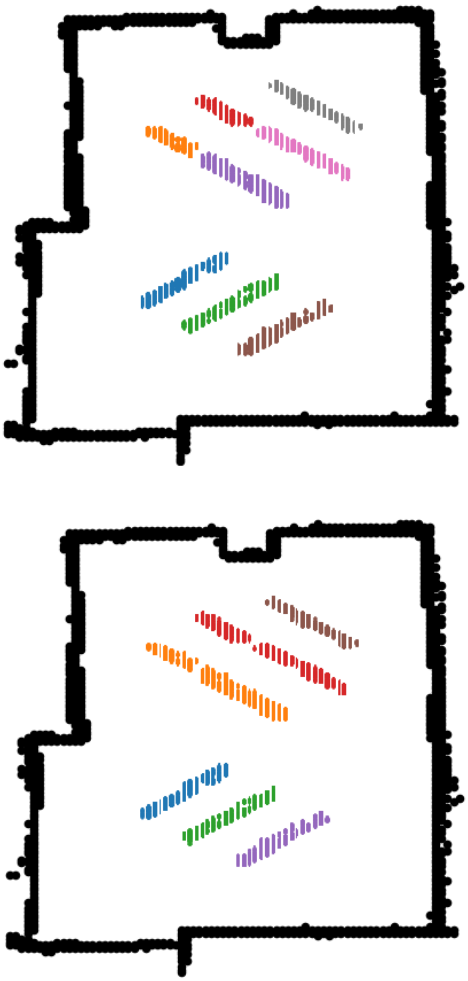


Figure 3.3: The images show the row detection applying two different slinding window sizes: the picture at the top represents the correct points clustering.

Algorithm 3.1 Findrows

```
1: Input: Binary map matrix
2: Set a value to ρ according to vegetation characteristics
3: Create an empty list OPEN
4: Create the CLOSED list as False Positive of the initial matrix
5: for line of map matrix do
6:   while column ≤ total columns in map matrix do
7:     while black_pixel not found do
8:       if pixel = 1 and not yet visited then
9:         black_pixel found
10:        start_pixel ← black_pixel
11:        Set start_pixel as visited in CLOSED list
12:        Set start_pixel not yet expanded in OPEN list
13:      else
14:        Increment column
15:      end if
16:    end while
17:    while OPEN list not empty do
18:      Remove the first black_pixel from OPEN list and
         set it as next to be explored
19:      for any pixel distant less than ρ from next do
20:        if pixel = 1 and not yet visited then
21:          Set pixel as not yet expanded in OPEN list
22:          Set pixel as visited in CLOSED list
23:        end if
24:      end for
25:    end while
26:    Increment column
27:    Store all black_pixels in CLUSTER list
28:    Label the CLUSTER with a number
29:    Store the CLUSTER in CLUSTERS
30:  end while
31: end for
```

3.5 First-order Row Approximation

At this stage, analysing the content of the previous section, it can be noticed that a recurring problem occurs between consecutive clusters: the elements stored in a cluster are not a complete row but part of the original one. This fact can be attributed to missing plants in the rows or even a defective row, as can be seen for example in figure 3.4.

On the other hand, dealing with more than one parcel per map, it may

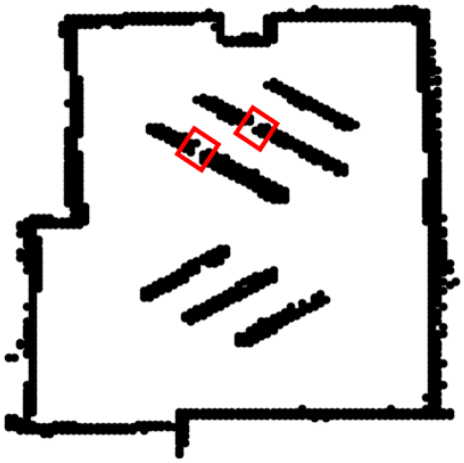


Figure 3.4: Gaps between consecutive rows are highlighted with squared boxes

occur that two of them are separated by a narrow path where the farmer or the vehicle have the possibility to pass. For this reason the problem is twofold: how to recombine only the segments that effectively belong to the same row?

In order to answer this question further steps of the algorithm need to be investigated. For the moment, it is possible to state that the linear regression, which will be treated in this section, is at the base of future discussions.

The idea is to perform the linear regression of every row, finding the line that best fits each cluster, that is to say it is necessary to find for every detected cluster, a line that best fits the distributed points within it.

For this case the *Ordinary Least Squares* method has been chosen to parametrize each of the rows identified in the previous section. *Ordinary Least Squares* is one of the most well known methods of linear regression. The OLS determines the parameters of a linear function given a set of variables: it minimizes the

sum of the squares of the differences between the observed dependent variable in the given dataset and those predicted by the linear function.

Consider an *overdetermined* system

$$\sum_{j=1}^{n} X_{ij}\theta_j = y_i \qquad i = 1,2,\ldots,m \tag{3.1}$$

of m linear equations in n unknown coeffcients, $\beta_1, \beta_2, \ldots, \beta_n$, with $m \geq n$, which can be rewritten as

$$\mathbf{X}\theta = \mathbf{y} \tag{3.2}$$

where

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \ldots & X_{1n} \\ X_{21} & X_{22} & \ldots & X_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m1} & X_{m2} & \ldots & X_{mn} \end{bmatrix} \tag{3.3}$$

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \tag{3.4}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{3.5}$$

The goal is to find if possible the values of n parameters $\theta_1, \theta_2, \ldots, \theta_n$ which *best* fit the equations, in the sense of solving the *quadratic minimization* problem

$$\hat{\theta} = \underset{\theta \in R^n}{\arg\min}\, J(\theta) \tag{3.6}$$

where the objective function $J(\theta)$ is given by

$$J(\theta) = \sum_{i=1}^{m} \left| y_i - \sum_{j=1}^{n} X_{ij}\theta_j \right|^2 = \|\mathbf{y} - \mathbf{X}\theta\|^2 \tag{3.7}$$

which corresponds to the non-zero error. Provided that the n columns of the matrix X are linearly independent, this minimization problem has a unique solution given by the result of the *normal equations*

$$(\mathbf{X^T X})\hat{\theta} = \mathbf{X^T y} \tag{3.8}$$

The estimate of the parameters θ is obtained by the *Least Squares* estimate:

$$\hat{\theta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y} \tag{3.9}$$

As the canopy rows are, generally, disposed as straight lines, the most recurrent form for y is a first order polynomial:

$$y(x) = mx + q \tag{3.10}$$

where m stands for the *gain* and q for the offset.
The error is computed only in the vertical axis and then can be passive of error if the points are spread around the vertical axis. Due to the limitation of the model, if the points to be approximated are distributed along a vertical line and the variance occurs mainly in the horizontal direction, the *OLS* will not perform a good parametrization of the model. The problem occurs mainly because the *OLS* is designed to fit points with varancies in the vertical direction, i.e. the y-axis. Given a dataset where the data points are spread along the vertical direction, the variance is in general found in the x variable and not in y, therefore the equations on which the function relies are the opposite to what was hyphotetized before. Processing the dataset directly using the OLS method returns lines for each row that do not correspond to a correct parametrization. In order to solve this problem, before projecting the data points into the *OLS* space, a rotation of 90 degrees was applied to the clusters which presented such *Verticality*, defined as follows:

$$Verticality = \frac{\sigma_Y}{\sigma_X} \tag{3.11}$$

where σ_Y and σ_X represent respectively the standard deviation of Y and X data points. When the data points variance is more spread along the x-axis with respect to the y-axis, the *Verticality* is much higher than the unity. This parameter could be very useful when dealing with long rows like in the case study. The procedure ensures that all the rows are correctly linearized. It is important to notice that if the rows are horizontally oriented, the rotation does not need to be applied.

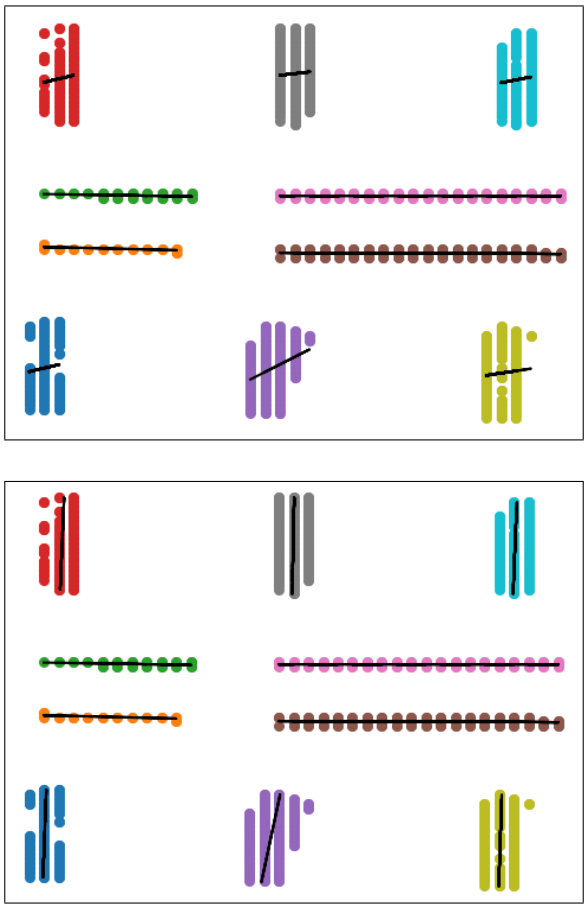


Figure 3.5: Rows parametrized using the *OLS* method. The wrong approximation in upper image is corrected in the lower image.

The figure 3.5 illustrates the differences between the parametrization performed using *OLS* of the same row before and after the rotation is applied. Notice that when the parametrization is performed on vertically oriented rows the results are far from the expected. On the contrary, when rows are horizontally oriented the *OLS* performs an optimal approximation.
To improve performance in terms of computational time, during the implementation in Python environment, the *Scikit-Learn* library has been exploited, in particular the *LinearRegression* method. Below it is possible to have a sight of the code.

```
1 from sklearn.linear_model import LinearRegression
2 regressor = LinearRegression()
```

```
coeffs = []
for cluster in clusters:
  x = []
  y = []
  xv = []
  yv = []
  for i in range(len(cluster)):
    x.append(cluster[i][0])
    y.append(cluster[i][1])
    xv.append(-cluster[i][1])
    yv.append(cluster[i][0])
  coef = np.polyfit(x, y, 1)
  x = np.reshape(x, (-1, 1))
  y = np.reshape(y, (-1, 1))
  stDevX = np.std(x)
  stDevY = np.std(y)
  verticality = stDevY / stDevX
  isVertical = verticality > 1.5
  regressor.fit(x, y)
if (isVertical):
  coef = np.polyfit(xv, yv, 1)
  regressor.fit(-y, x)
  x_hat = regressor.predict(-y)
else:
  regressor.fit(x, y)
  y_hat = regressor.predict(x)
```

As output, the coefficients for a polynomial of degree n that best fits the y data are returned. Those coefficients define the line that best fits the n data points passed as input for the *OLS* function. At the end, the identifications of each parameter are stored as well as the extreme points and the mean point of the parametrization line. These data will be exploited later in the next steps of the algorithm presented in the next chapter.

Figure 3.6

Figure 3.7

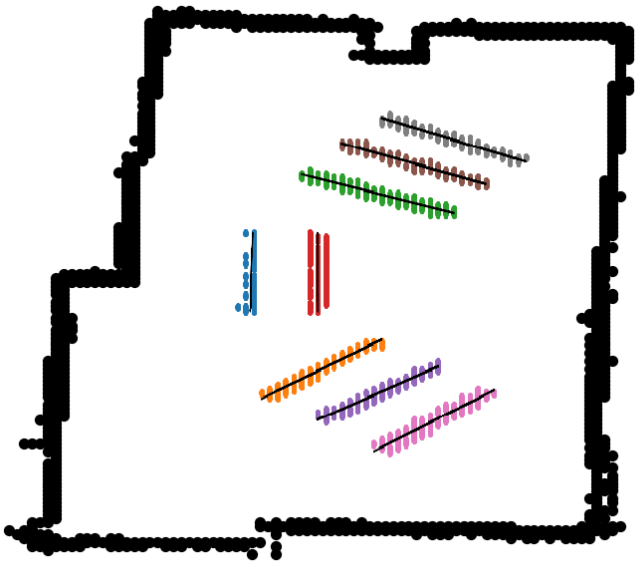


Figure 3.8

Figure 3.9

Figure 3.10

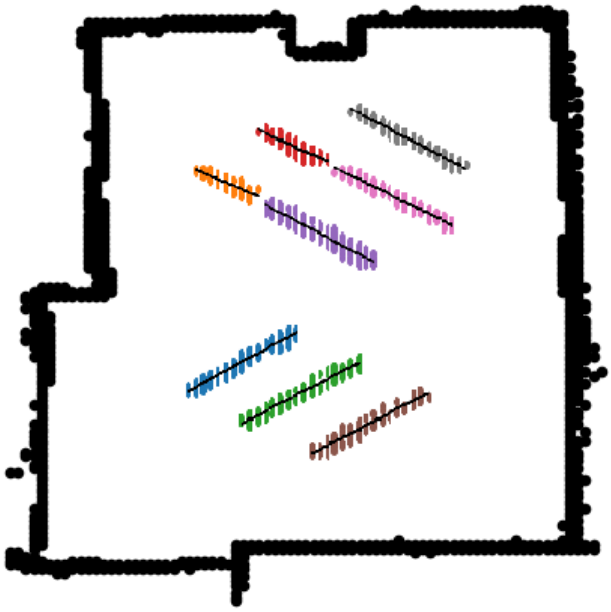


Figure 3.11

Figure 3.12

Figure 3.13

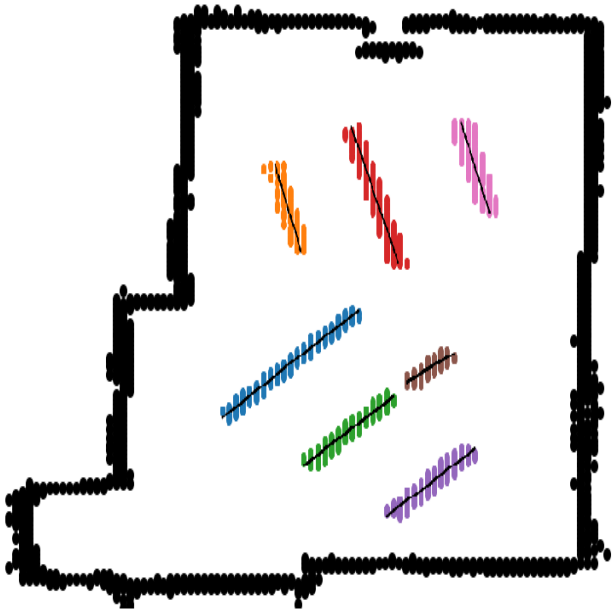


Figure 3.14

Figure 3.15

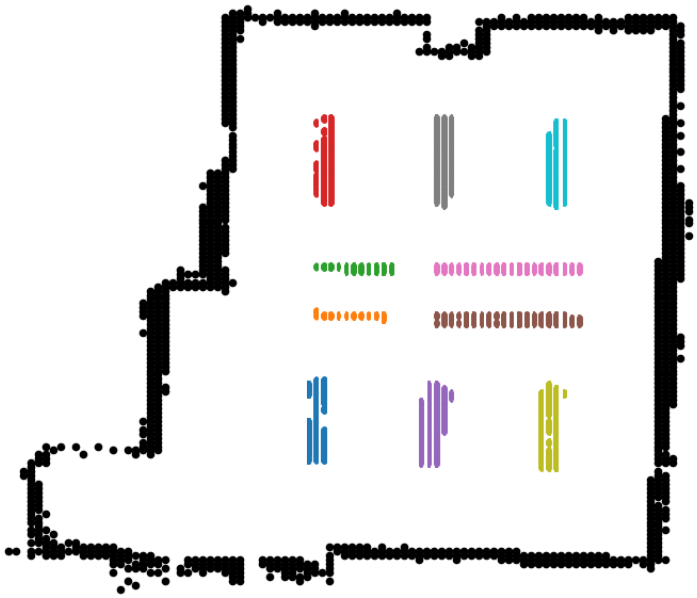


Figure 3.16

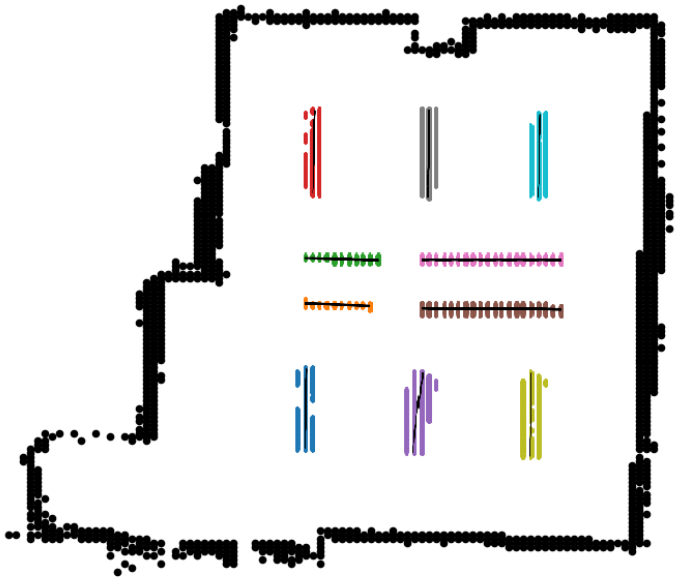


Figure 3.17

Chapter 4

Parcel Clustering

As anticipated in the previous chapter, once all the rows have been identified, they need to be separated and grouped in parcels. For this reason it is necessary to find a criterion which can be used to properly correlate only the rows of the same parcel, avoiding as much as possible errors.

An in-depth analysis of the various clustering algorithms present in literature has been carried out. In parallel, particular attention has been given to the implementation, which has further narrowed down potential solution. The biggest problem encountered is related to the fact that the number of parcels within a field is not known a priori, unless the final user provides this information to the algorithm. However, the idea behind this work is to produce a platform as autonomous as possible and at the same time general purpose, meaning that the human intervention should be limited to a small number of actions related mostly to initial settings.
For this reason the simpler algorithms or very hardware-demanding, such as *K-means* or *DBSCAN*, have been discarded [30] [11] [29].
The author has decided to focus his efforts on studying and implementing clustering through the *Dirichlet Process* model, which will be explained to the reader in this chapter.

4.1 Dirichlet Mixture model

Starting from the point made at the end of the previous chapter, it is now the moment to think about the use of the lines approximation with the *Ordinary Least Squares* method. Considering two parcels separated by a free space of higher dimension with respect to the inter-rows distance belonging to the

same parcel. Furthermore, assuming they are oriented differently and the rows have similar orientation between them, it is possible to the human eye to distinguish them without any kind of problem. The idea is to use their *slope* to identify them uniquely. The result will be a collection of slopes close to each other if they belong to the same parcel (fig. 4.1).

For this reason the slope of the rows can be considered a good feature,

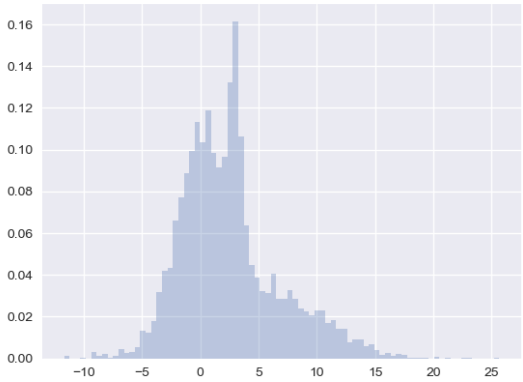


Figure 4.1: Example of data collection distribution

which enables the clustering process. Due to the presence of different slopes, the resulting statistical distribution of angular coefficient will be in general multimodal (fig. 4.2). The assumption that all the data are generated from a

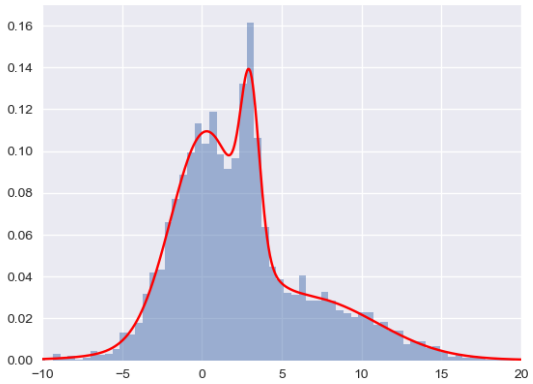


Figure 4.2: Example of a multimodal distribution

mixture of finite number of distributions with unknown parameters(fig. 4.3)

allows to introduce the core of the clustering, the *Dirichlet Process*.

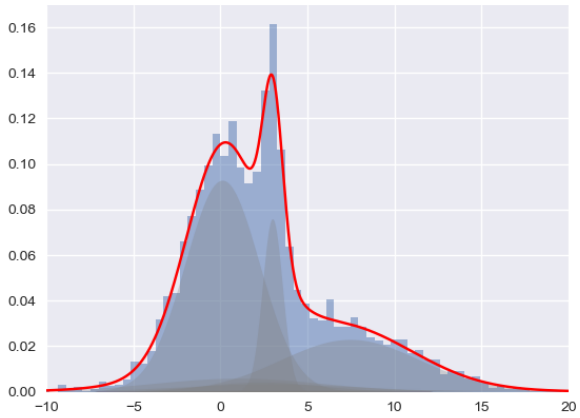


Figure 4.3: Multimodal distribution composition

Introduction

In *parametric* modeling, it is assumed that data can be represented by models using a fixed, finite number of parameters. Examples of parametric models include clusters of K Gaussians and polynomial regression models. In many problems, determining the number of parameters a priori is difficult; for example, selecting the number of clusters in a cluster model, the number of segments in an image segmentation problem, the number of chains in a hidden Markov model, or the number of topics in a topic modelling problem before the data is seen can be problematic.
In *nonparametric* modeling, the number of parameters is not fixed, and often grows with the sample size. Kernel density estimation is an example of a nonparametric model. In Bayesian nonparametrics, the number of parameters is itself considered to be a random variable. One example is to do clustering with *k-means* (or mixture of Gaussians) while the number of clusters k is unknown. Bayesian inference addresses this problem by treating k itself as a random variable. A prior is defined over an infinite dimensional model space, and inference is done to select the number of parameters. Such models have infinite capacity, in that they include an infinite number of parameters a priori; however, given finite data, only a finite set of these parameters will be used. Unused parameters will be integrated out.

Mixture Model Example

An example of a parametric model is the mixture model of K Gaussians with finite number of parameters:

$$p(x_1, \ldots, x_N \mid \pi, \{\mu_k\}, \{\Sigma_k\}) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n \mid \mu_k, \Sigma_k) \tag{4.1}$$

Adopting a Bayesian approach, each parameter would be given a prior distribution and integrated out:

$$p(x_1, \ldots, x_N) = \tag{4.2}$$
$$\int \cdots \int \Big(\prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n \mid \mu_k, \Sigma_k) \Big) p(\pi) p(\mu_{1:K}) p(\Sigma_{1:K}) \; d\pi d\mu_{1:K} d\Sigma_{1:K}$$

It is best to choose conjugate prior distributions to simplify posterior inference. For the Gaussian parameters, the Gaussian and inverse Wishart distributions are the conjugate distributions for the mean and co-variance respectively. For the mixture weights, the conjugate is the Dirichlet distribution.

The Dirichlet Distribution

The Dirichlet distribution is a distribution over the (K-1)-dimensional simplex; that is, it is a distribution over the relative values of K components, whose sum is restricted to be 1. It is parameterized by a K-dimensional vector (α_1, …, α_K), where $\alpha_k \geq 0 \; \forall k$ and $\Sigma_k \alpha_k > 0$. Its distribution is given by:

$$\pi = (\pi_1, \ldots, \pi_K) \sim \text{Dirichlet}(\alpha_1, \ldots, \alpha_K) = \frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma\left(\sum_{k=1}^{K} \alpha_k\right)} \prod_{k=1}^{K} \pi_k^{\alpha_k - 1} \tag{4.3}$$

If $\pi \sim \text{Dirichlet}(\alpha_1, \ldots, \alpha_K)$ then $\pi_k \geq 0 \; \forall k$ and $\Sigma_k \alpha_k = 1$. The expectation of the distribution is:

$$\mathbb{E}[(\pi_1, \ldots, \pi_K)] = \frac{(\alpha_1, \ldots, \alpha_k)}{\Sigma_k \alpha_k} \tag{4.4}$$

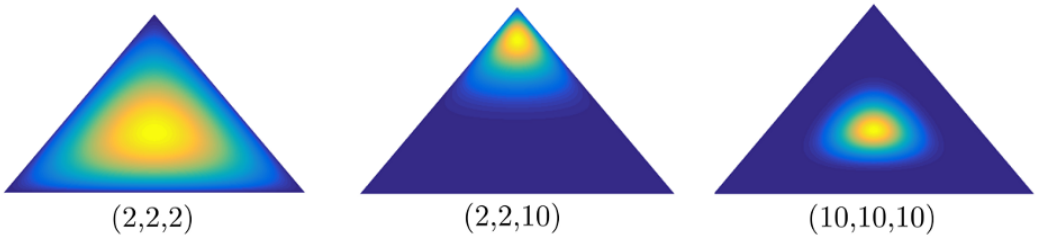


Figure 4.4: Density of the 3-component Dirichlet distribution for different parameter settings.

Figure 4.4 shows how the density of a Dirichlet distribution over 3 components varies for different settings of its scaling parameters α. Note that as the parameter values become larger, the distribution becomes more concentrated at the extremes (i.e. it is more likely that one component take on value 1 and the rest value 0). Furthermore, different values for the parameters can skew the distribution.

Conjugacy with the Multinomial Distribution

It can be shown that the Dirichlet distribution is the conjugate of the Multinomial distribution. If $\pi \sim \text{Dirichlet}(\alpha_1, \ldots, \alpha_K)$ and $x_n \sim \text{Multinomial}(\pi)$ are *indipendent* and *identically distributed* samples, then:

$$
\begin{aligned}
p(\pi \mid x_1, \ldots, x_n) &\propto p(x_1, \ldots, x_n \mid \pi) p(\pi) \qquad (4.5) \\
&= \Big(\frac{\prod_{k=1}^{K} \Gamma(\alpha_k)}{\Gamma\big(\sum_{k=1}^{K} \alpha_k\big)} \prod_{k=1}^{K} \pi_k^{\alpha_k - 1} \Big) \Big(\frac{n!}{m_1! \ldots m_K!} \pi_1^{m_1} \ldots \pi_K^{m_K} \Big) \\
&\propto \frac{\prod_{k=1}^{K} \Gamma(\alpha_k + m_k)}{\Gamma\big(\sum_{k=1}^{K} \alpha_k + m_k\big)} \prod_{k=1}^{K} \pi_k^{\alpha_k + m_k - 1} \\
&= \text{Dirichlet}(\alpha_1 + m_1, \ldots, \alpha_K + m_K) \qquad (4.6)
\end{aligned}
$$

Where m_k represent the counts of instances of $x_n = k$ in the data set.
The Dirichlet distribution can be viewed as a distribution over finite-dimensional

distributions; that is, it is a distribution over parameters for the Multinomial distribution, where each sample from the Dirichlet distribution can be regarded as a Multinomial distribution. Furthermore, it is possible to associate each component with a set of parameters. In the case of a Gaussian mixture model, these parameters would be the mean and covariance of each cluster. It is necessary therefore to define a prior distribution over Gaussians. In the Bayesian setting, these parameters are themselves random variables. In a Bayesian finite mixture model, a Gaussian prior is combined over the centroid locations of clusters with a Dirichlet prior over the cluster weights.

Other Properties of the Dirichlet Distribution

The Dirichlet distribution satisfies the coalesce rule:

$$(\pi_1 + \pi_2, \pi_3, \ldots, \pi_K) \sim \text{Dirichlet}(\alpha_1 + \alpha_2, \alpha_3, \ldots, \alpha_K) \tag{4.7}$$

The Dirichlet distribution also satisfies the expansion or combination rule, which allows us to increase the dimensionality of a Dirichlet distribution. Note that the Dirichlet distribution over the 1-dimensional simplex is simply the Beta distribution. Let $(\pi_1, \ldots, \pi_K) \sim \text{Dirichlet}(\alpha_1, \ldots, \alpha_K)$ and $\theta \sim \text{Beta}((\alpha_1 b, \alpha_1(1-b))$ for $0 < b < 1$. Then one dimension of the Dirichlet distribution can be split into two dimensions as follows:

$$(\pi_1\theta, \pi_1(1-\theta), \pi_2, \ldots, \pi_K) \sim \text{Dirichlet}(\alpha_1 b, \alpha_1(1-b), \alpha_2, \alpha_K) \tag{4.8}$$

More generally, if $\theta \sim \text{Dirichlet}(\alpha_1 b_1, \alpha_1 b_2, \ldots, \alpha_1 b_N)$ and $\Sigma_i b_i = 1$, then:

$$(\pi_1\theta_1, \ldots, \pi_1\theta_N, \pi_2, \ldots, \pi_K) \sim \text{Dirichlet}(\alpha_1 b_1, \ldots, \alpha_1 b_N, \alpha_2, \alpha_K) \tag{4.9}$$

Finally, The Dirichlet distribution also satisfies the renormalization rule. If $(\pi_1, \ldots, \pi_K) \sim \text{Dirichlet}(\alpha_1, \ldots, \alpha_K)$ then:

$$\frac{(\pi_2, \ldots, \pi_K)}{\sum_{k=1}^{k} \pi_k} \sim \text{Dirichlet}(\alpha_2, \ldots, \alpha_K) \tag{4.10}$$

Constructing an Infinite-Dimensional Prior

In problems such as clustering, the number of clusters is not known a priori. When defining a prior for the mixture weights,a distribution that allows an infinite number of clusters is needed, so that it will be always possible to

have more clusters than needed in any given problem. An infinite mixture model has the form:

$$p(x_1 \mid \pi, \{\mu_k\}, \{\Sigma_k\}) = \sum_{k=1}^{\infty} \pi_k \mathcal{N}(x_n \mid \mu_k, \Sigma_k) \tag{4.11}$$

Therefore it is preferred to use a prior that has properties like that of the Dirichlet distribution (such as conjugacy with the Multinomial), but is infinite-dimensional. To define such a distribution, let consider the following scheme. Begin with a two-component Dirichlet distribution, with scaling parameter α divided equally between both components, for the sake of symmetry:

$$\pi^{(2)} = (\pi_1^{(2)}, \pi_2^{(2)}) \sim \text{Dirichlet}(\frac{\alpha}{2}, \frac{\alpha}{2}) \tag{4.12}$$

Then, split off components according to the expansion rule:

$$\theta_1^{(2)}, \theta_2^{(2)} \sim \text{Beta}(\frac{\alpha}{2} \cdot \frac{1}{2}, \frac{\alpha}{2} \cdot \frac{1}{2}) \tag{4.13}$$

$$\pi^{(4)} = (\theta_1^{(2)}\pi_1^{(2)}, (1-\theta_1^{(2)})\pi_1^{(2)}, \theta_2^{(2)}\pi_2^{(2)}, (1-\theta_2^{(2)})\pi_2^{(2)}) \sim \text{Dirichlet}(\frac{\alpha}{4}, \frac{\alpha}{4}, \frac{\alpha}{4}, \frac{\alpha}{4})$$

By repeating this process, the final distribution is such that:

$$\pi^{(K)} \sim \text{Dirichlet}\big(\frac{\alpha}{K}, \ldots, \frac{\alpha}{K}\big) \tag{4.14}$$

In the limit as K goes to infinity, the prior over an infinite-dimensional space is obtained. In practice all these components will never be used, but only the components which reflect the data.

4.2 Dirichlet Process

Let the base measure H be a distribution over some space Ω(for example, a Gaussian distribution). Let:

$$\pi \sim \lim_{K \to \infty} \text{Dirichlet}\big(\frac{\alpha}{K}, \frac{\alpha}{K}\big) \tag{4.15}$$

For each point in this Dirichlet distribution, it is possible to associate a draw from the base measure:

$$\theta_k \sim H \qquad \text{for} k = 1, \ldots, \infty \tag{4.16}$$

Then:

$$G = \sum_{k=1}^{\infty} \pi_k \delta_{\theta k} \tag{4.17}$$

is an infinite discrete distribution over the continous space Ω. It can be written as a Dirichlet Process:

$$G \sim \mathrm{DP}(\alpha, H) \tag{4.18}$$

Samples from the Dirichlet Process are discrete. The point masses in the resulting distribution are called atoms; their positions in Ω are drawn from the base measure H, while their weights are drawn from an infinite-dimensional Dirichlet distribution. The concentration parameter α determines the distribution over atom weights; smaller values lead to sparser distributions, with larger weights on each atom.

A Dirichlet Process is a unique distribution over probability distributions on some space such that for any finite partition $A_1, \ldots, A_K$ of Ω, the total mass assigned to each partition is distributed according to:

$$(P(A_1), \ldots, P(A_K)) \sim \mathrm{Dirichlet}(\alpha H(A_1), \ldots, \alpha H(A_K)) \tag{4.19}$$

Note that H may be un-normalized. Furthermore, a cumulative distribution function G, on possible worlds of random partition follows a Dirichlet Process if for any measurable finit partition $(\phi_1, \phi_2, \ldots, \phi_m)$:

$$(G(\phi_1), \ldots, G(\phi_m)) \sim \mathrm{Dirichlet}(\alpha G_0(\phi_1), \ldots, \alpha G_0(\phi_m)) \tag{4.20}$$

where G_0 is the base measure and α is the scale parameter.

Conjugacy

Let $A_1, \ldots, A_K$ be a partition of Ω, and let H be a measure on Ω. Let $P(A_k)$ be the mass assigned by $G \sim \mathrm{DP}(\alpha, H)$ to partition A_k, then:

$$(P(A_1), \ldots, P(A_K)) \sim \mathrm{Dirichlet}(\alpha H(A_1), \ldots, \alpha H(A_K) \tag{4.21}$$

Focusing on an observation in the J^{th} segment(of fraction), then:

$$\begin{aligned}&(P(A_1), \ldots, P(A_j), \ldots, P(A_K) \mid X_1 \in A_j) \sim \\ &\mathrm{Dirichlet}(\alpha H(A_1), \ldots, \alpha H(A_j) + 1, \ldots, \alpha H(A_K))\end{aligned} \tag{4.22}$$

Since this must be true for all possible partitions of Ω, this is only possible if the posterior gor G is given by:

$$G \mid X_1 = x \sim \mathrm{DP}\Big(\alpha + 1, \frac{\alpha H + \delta_x}{\alpha + 1}\Big) \tag{4.23}$$

Predictive Distribution

The Dirichlet distribution can be a prior for mixture models, thus the Dirichlet Process could be further used to cluster observations. A new data point can either join an existing cluster or start a new cluster. Assume H is a continuous distribution on Ω, which are the parameters of modeling the observed data points. Once a first data point is available, it is possible to start a new cluster with a sampled parameter θ_1. Now, the parameter space can be split in two: the singleton θ_1, and everything else. Let π_1 be the atom at θ_1. The combined mass of all the other atoms is $\pi_* = 1 - \pi_1$, and

$$prior : (\pi_1, \pi_*) \sim \text{Dirichlet}(0, \alpha) \tag{4.24}$$

$$posterior : (\pi_1, \pi_*) \mid X_1 = \theta_1 \sim \text{Dirichlet}(1, \alpha) \tag{4.25}$$

Integrating out π_1

$$\begin{aligned} P(X_2 = \theta_k \mid X_1 = \theta_1) &= \int P(X_2 = \theta_k \mid (\pi_1, \pi_*)) P((\pi_1, \pi_*) \mid X_1 = \theta_1)\ d\pi_1 \\ &= \int \pi_k \text{Dirichlet}(1, \alpha)\ d\pi_1 \\ &= \mathbb{E}_{\text{Dirichlet}(1-\alpha)}[\pi_k] \\ &= \begin{cases} \frac{1}{1+\alpha} & \text{if } k = 1 \\ \frac{\alpha}{1+\alpha} & \text{for new } k. \end{cases} \end{aligned} \tag{4.26}$$

This basically tells that with probability $\frac{1}{1+\alpha}$, the parameter θ stays in the old cluster and with probability $\frac{\alpha}{1+\alpha}$ it starts a new cluster. If a new cluster is chosen to be started a new parameter $\theta_2 \sim H$ is sampled. Let π_2 be the size of the atom at θ_2, and

$$posterior : (\pi_1, \pi_2, \pi_*) \mid X_1 = \theta_1, X_2 = \theta_2 \sim \text{Dirichlet}(1,1, \alpha) \tag{4.27}$$

By integrating out $\pi = (\pi_1, \pi_2, \pi_*)$

$$\begin{aligned} &P(X_3 = \theta_k \mid X_1 = \theta_1, X_2 = \theta_2) = \\ &= \int P(X_3 = \theta_k \mid \pi) P(\pi \mid X_1 = \theta_1, X_2 = \theta_2)\ d\pi \\ &= \mathbb{E}_{\text{Dirichlet}(1,1,\alpha)}[\pi_k] \\ &= \begin{cases} \frac{1}{2+\alpha} & \text{if } k = 1 \\ \frac{1}{2+\alpha} & \text{if } k = 2 \\ \frac{\alpha}{2+\alpha} & \text{for new } k. \end{cases} \end{aligned} \tag{4.28}$$

In general, if m_k is the number of times $X_i = k$ occur, and K is the total number of observed values:

$$P(X_{n+1} = \theta_k \mid X_1 = \theta_1, \ldots, X_n) = \int P(X_{n+1} = \theta_k \mid \pi) P(\pi \mid X_1, \ldots, X_n)\, d\pi \tag{4.29}$$

$$= \mathbb{E}_{\text{Dirichlet}(m_1,\ldots,m_k,\alpha)}[\pi_k]$$

$$= \begin{cases} \frac{m_k}{n+\alpha} & \text{if } k \leq K \\ \frac{\alpha}{n+\alpha} & \text{for new cluster.} \end{cases}$$

which gives a simple closed-form predictive distribution for the next observation. Such a predictive distribution is especially useful for sampling and inference in Dirichlet Processes. Note that this distribution has a "rich-get-richer" property; clusters with more observations are more likely to have new observations.
However, there is always the possibility of seeing a novel observation, with controlling the tendency to initiate a new cluster.

Useful Metaphors

Several useful metaphors exist for helping to understand the Dirichlet Process.

Pòlya Urn Process: Consider an urn with a black ball of mass α. Iteratively sample balls from the urn with probability proportional to their mass. If the ball is black, return it to the urn, choose a previously unseen color, and add a unit mass ball of that color to the urn. If the ball is colored, return it an another unit mass ball of the same color to the urn.

Chinese Restaurant Process: Consider a Chinese restaurant with infinitely many tables. As customers enter, they may sit at an occupied table with probability proportionate to how many customers are already seated there, or they may sit at an unoccupied table with probability proportionate to α. Also, at each table, a dish is selected and shared by the customers seated there; this is analogous to a draw θ from the base measure H. From this example it is possible to see that the distribution does not depend on the ordering in which the customers arrived; this is the property of exchangeability. This way, each customer can be treated indepdendently, as if they were the last to arrive; this is a useful property for Gibbs sampling in Dirichlet

Processes.

Stick-breaking Process: Consider a stick of unit length. Iteratively sample a random variable b_k from Beta$(1, \alpha)$ and break off a fraction of b_k of the stick. This is the weight of the k^{th} atom. Sample a location for this atom from the base measure H and repeat. This will give an infinite number of atoms, but as the process continues, the weights of the atoms will decrease to be negligible. This motivates truncation approximations, in which the full Dirichlet Process is approximated using a finite number of clusters.

4.3 Clustering Implementation

Once defined and understood the meaning of the *Dirichlet Process*, it is possible to apply this kind of clustering to the map which represents the environment. Since it is a prior probability distribution on clusterings with an infinite, unbounded, number of partitions, it suits the purposes of this work. To do the partitioning the already mentioned Python library *Scikit-learn* [23] has been used, in particular the *sklearn.mixture.BayesianGaussianMixture* class, which allows to infer an approximate posterior distribution over the parameters of a Gaussian mixture distribution as reported in the documentation. The effective number of components can be inferred from the data.
This class implements two types of prior for the weights distribution: a finite mixture model with Dirichlet distribution and an infinite mixture model with the Dirichlet Process. In practice Dirichlet Process inference algorithm is approximated and uses a truncated distribution with a fixed maximum number of components (called the Stick-breaking representation). The number of components actually used almost always depends on the data.
Its most important parameters are the following:

- The number of mixture components: depending on the data and the value of the weigth concentration prior, it is likely that the model decides not to use all components. As a consequence the number of effectively used components will be always smaller than that set
- The covariance type, which takes into account the covariance matrix of the components
- The maximum number of iterations to be performed
- The type of the weight concentration prior: it can be either a *Dirichlet Process* or a *Dirichlet Distribution*

- The prior on the covariance distribution (Wishart)

- The convergence threshold: the iterations will stop when the lower bound average gain on the likelihood (of the training data with respect to the model) is below this threshold.

Considering the fact that no information is given in advance, and being an *unsupervised* clustering, the most generic parameteres have been set.
As first attempt, the algorithm tried to do the clustering using as feature only the information about the slope of the rows.
Nevertheless, at first hand it seems that this feature is not enough restrictive to bind together the right rows, as figure 4.5 illustrates.
This can be confirmed by the fact that when the absolute value of the slopes

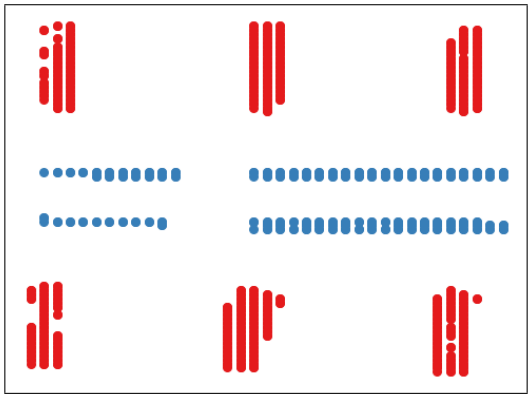


Figure 4.5: Clustering with only the information about the slope

of distinct parcels are visibly different the algorithm works better. To solve this problem a further feature must be included, which can correlate the rows according to their mutual position. A possible solution could be to pick from each row the coordinates of its mean point, in such a way that distant mean points are unlikely to be recognized as part of the same cluster.
By introducing this new feature, it is possible to appreciate the improvements in the results: in figure 4.6 the clustering has been performed significantly better. Appendix C shows the entire code implementation.
However, some imperfections are still present, basically due to the rows dimension. In fact, very small rows, due to their round shape, have the inconvenience to be linearized with lines completely different from the others. As a consequence, their clustering results even more difficult.

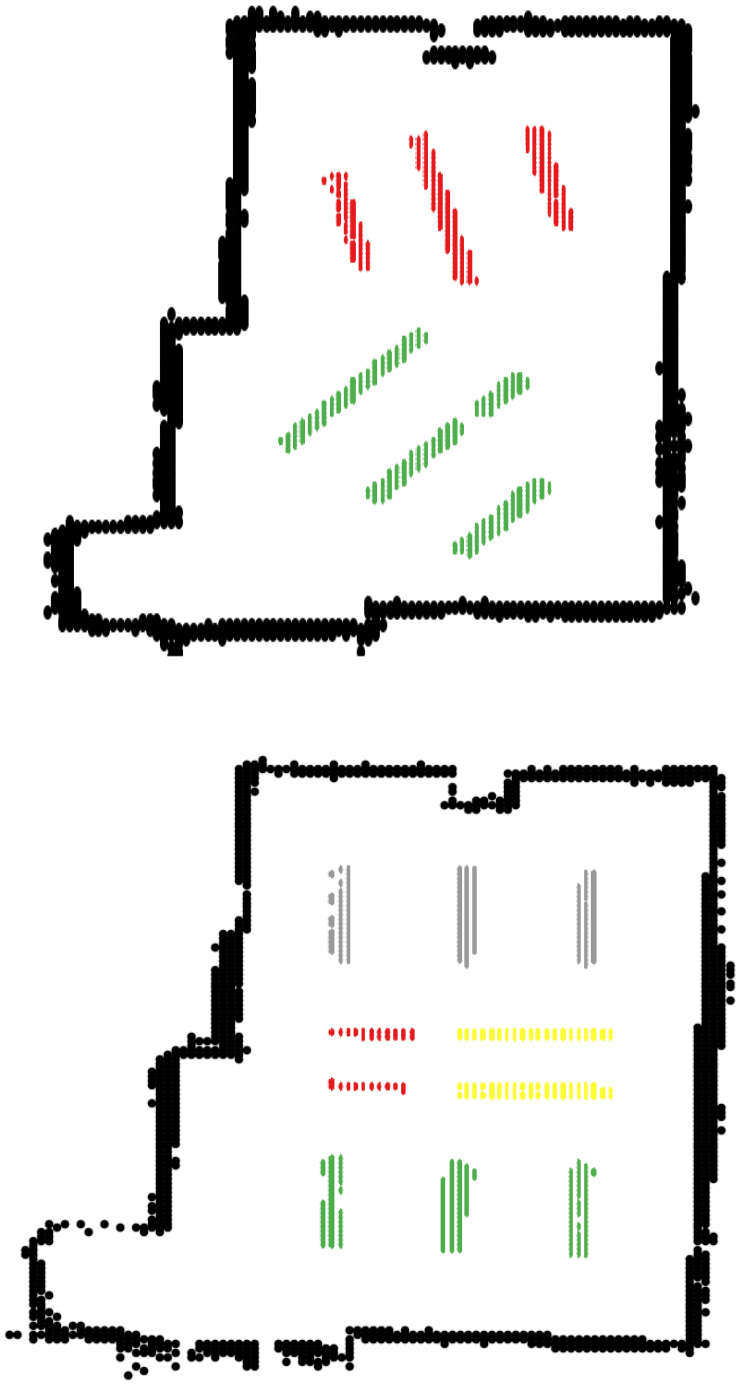


Figure 4.6: Rows clustering of different environment configurations

4.4 Clustering Error Correction

This section will address the problem of clustering imperfections in parcels. In order do identify the rows to whom it was assigned from the previous algorithm an incorrect *label*, corresponding to the parcel that specific row should belong, it is necessary to find a criterion which can operate locally.
In other words, each row must be treated as a single unit with its characterising label and the only entities able to confirm its belonging to the right group are its neighbours.
To apply this principle the *Cluster Correction* algorithm has been developed, which is explained in the following by means of the flowchart in figure 4.7. Due to its complexity and length, it was decided not to report all the code in the Appendix D, but only some significant parts. The basic principle of the algorithm is the following. It continues recursively to do the following operations until no more changes in the map are defined:

1. It looks, starting from the mean of the cluster, in the four directions until it finds a neighbour. In the case that no neighbour is reached in a precise direction, this will not be considered and a label *None* is assigned.
   The search process is done by extending the approximating line from the row's extremes until the neighbour is reached, concerning upper and lower neighbours, while to look to the right and the left the perpendicular to the approximating line is expanded until the neighbour is met.

2. Once the neighbours are detected, it is performed a count of how many times each label related to the neighbours occurs and then they are sorted in decreasing order.

3. In case the row's label is different from that of the neighbours and at least two of them out of four have a different label, then the change is applied.

Once all the parcels are identified, the only thing left is the analysis of consecutive rows in order to be merged in one cluster before the final clustering.

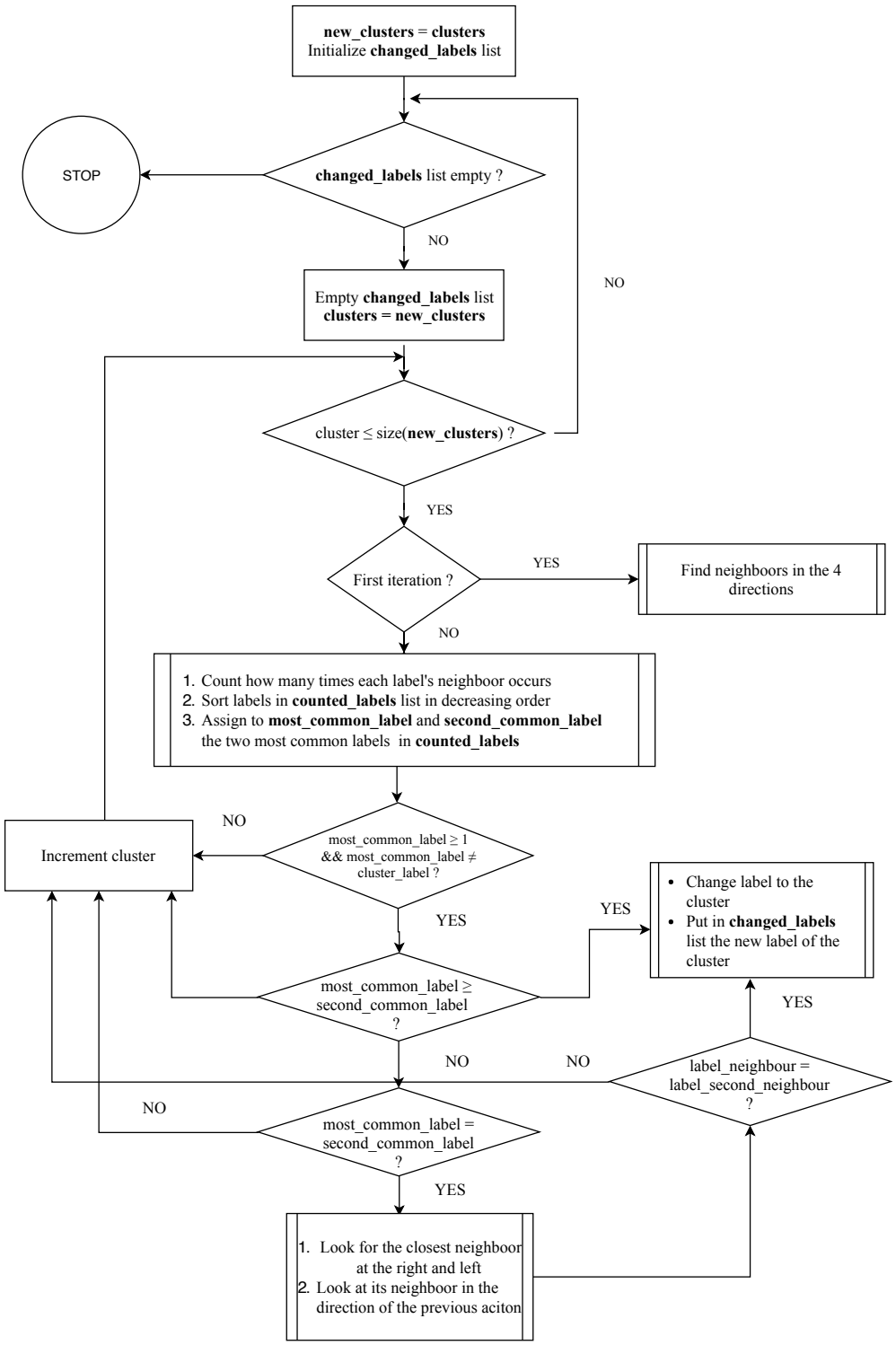


Figure 4.7: Cluster Correction flowchart

4.5 Defective Rows Correction

To complete the row detection, a recombination of the clusters is needed, in some cases, to deal with the problem of missing plants and defective rows. Indeed, a good number of short clusters of interconnected pixels were identified as being a segment, even if they should be part of a more extended row. The *Defective Rows Correction* in Appendix E is then called to solve those small misclassifications. The operations performed are described in the flowchart in figure 4.8.

The connection of two contiguous rows is done in the following steps:

1. The approximating line is extended from the row's extremes until the neighbour is reached.
2. The neighbour and the connecting points are integrated in the main row.
3. The neighbour row is removed from the list of rows of the cluster.

The figure 4.9 illustrates the impact of the function in the final clustering and labelling of the rows.

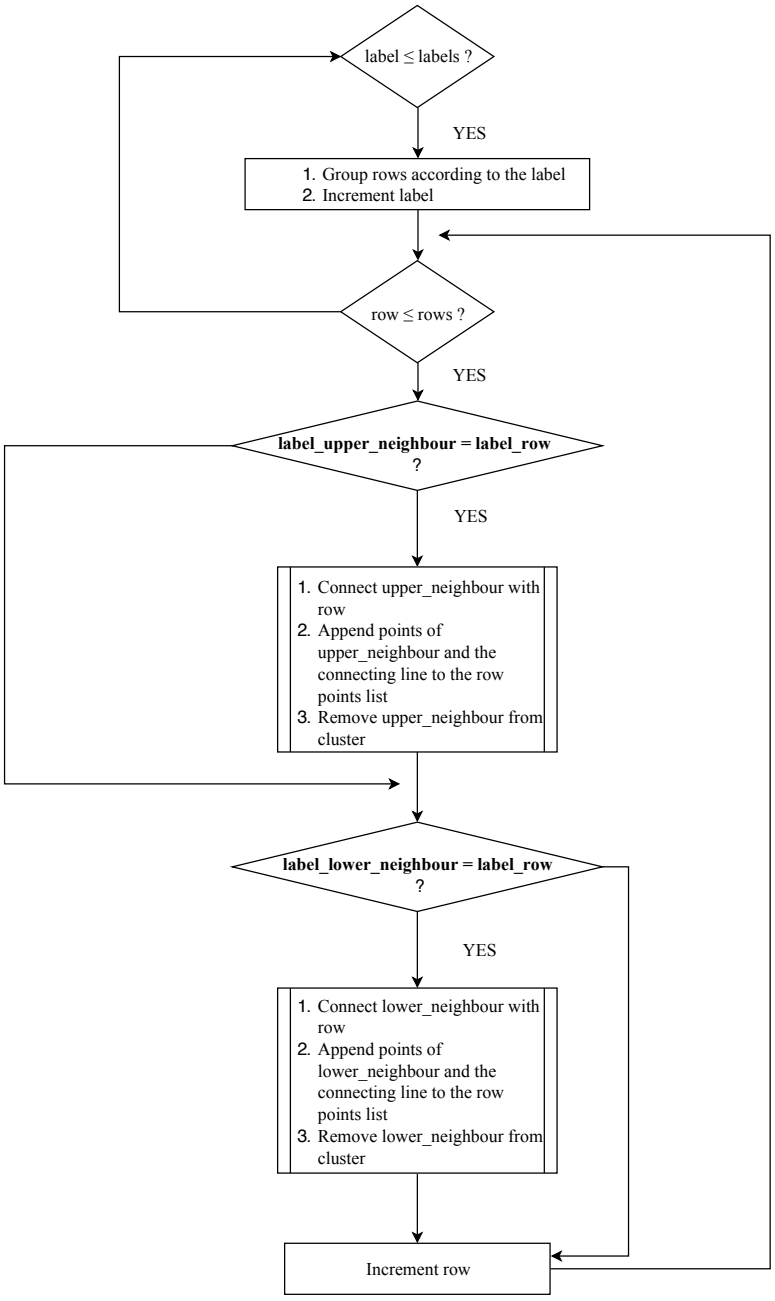


Figure 4.8: Defective Rows Correction flowchart

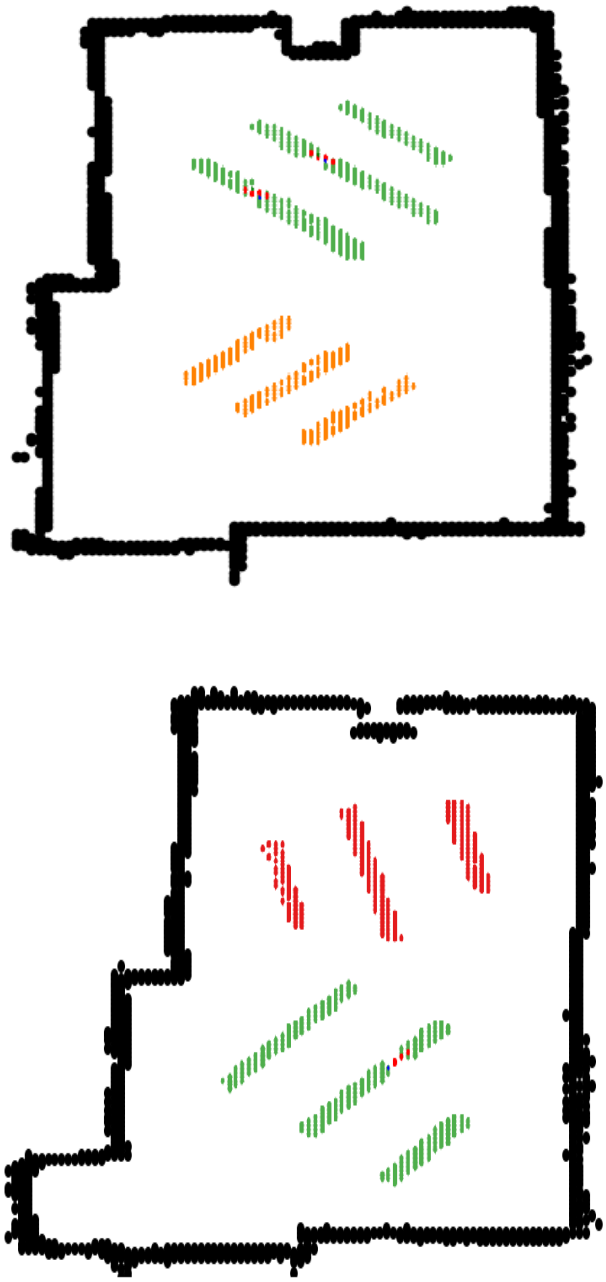


Figure 4.9: Recombination of consecutive rows with inner holes

Chapter 5

Navigation in Agricultural Environment

The objective of this chapter is to connect the work presented previously. Once the information about the environment have been corretly acquired and post-processed, it is necessary to exploit them in order to create a path plan for the robot able to cover all the rows within the same parcel.
Considering the proposed path planning algorithm, it can be noted that the desired route is not actually the one which goes from the starting point to the arrival in the suboptimal way generated from the $A*$ represented in figure 5.1, but rather the one which allows to cover all the rows.

For this reason, the robot needs to receive some "checkpoints" along the

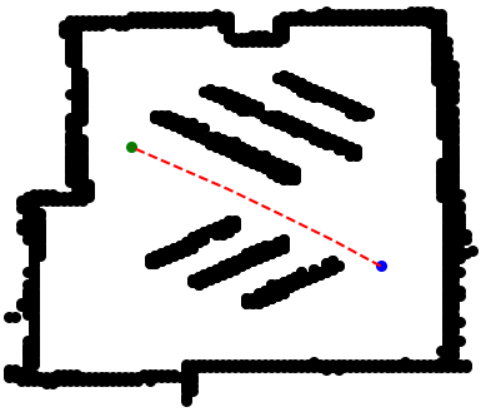


Figure 5.1

path between the starting position and the goal, that it must reach to be able to proceed. In this way a path to be covered is guaranteed.

5.1 Steps Generation

A possible solution to this problem is to exploit once again the structure of the environment, namely the line which best approximates the row and the distance between two close rows. With these information it is possible to create a grid that contains the parcel, whose dimensions must be such that the robot can navigate free of incident. Hence a safety distance must be included when it is built.
The desired steps will be the intersection of the lines which compose the grid in figure 5.2, which is created with the following criteria:

1. The rows are extended in both directions until the blue dots at least the dimensions of robot in such a way the $A*$ can perform the curve without too much trouble

2. From the points determined in the previous step, the average point is computed(red dots), meaning that when the robot will end to perform the curve, it will lie exactly in the middle of the two rows.

3. Once the inner lines have been found, to enclose the parcel the external points are determined by extending the line which best fits the last red point with the blue one.

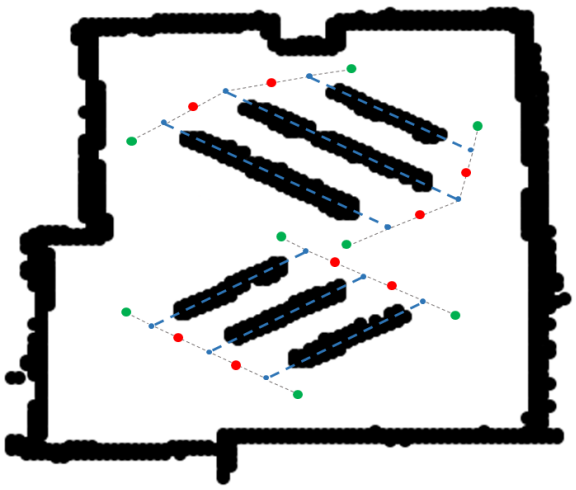


Figure 5.2: Red(inner steps) and green(external steps) define the grid

The flowchart in figure 5.3 will better elucidate the what previously explained.
All the procedure will also facilitate the task of the robot in the research of the path. In particular its main concern will be to find an available path between the rows. The parameter ρ defined in the flowchart describes the desired distance which the robot must take from the row. In Appendix F it is possible to view the code.

5.2 Steps Coverage

Once the steps are defined, it is still necessary to define the order in which the robot must cover them. For this reason a simple sorting function has been developed. According to the needs it can be used either to cover all the parcels which compose the map or only the single parcel.
It is based on the following principle. The data structure of the steps to be covered is such that for each row there is a vector which contains upper point and lower point, which are originally ordered according to the rows' allocation. Then the function is made up of these items:

1. From the starting point of the robot look at the external step points, referred to the first and last row that compose the parcel and find which one is closer in *Euclidian distance* to it

2. Sort the just found upper and lower external points belonging to the closest row to the start according to their *Euclidian distance* from it and assign as next starting point the farthest one

3. Repeat the process iteratively for each row in the parcel.

If there is the necessity to make the coverage of the whole map the same algorithm needs a further iteration. The developed code is proposed below.

```
1 def dist(x, y):
2   d = (x[0] — y[0]) ** 2 + (x[1] — y[1]) ** 2
3   return d
4
5 path = [start]
6 next = start
7 for label, steps in clusters_goals.iteritems():
```

```
8  distance_from_last = dist(start, steps[-1][0])
9  distance_from_first = dist(start, steps[0][0])
10 if distance_from_first > distance_from_last:
11   steps.reverse()
12 for step in steps:
13   step.sort(key=lambda x: (x[0] - next[0]) ** 2 + (x[1] - next[1]) ** 2)
14   path.append([int(step[0][0]), int(step[0][1])])
15   next = [step[1][0], step[1][1]]
16   path.append(next)
17   start = next
```

At this point it is possible to deploy the *Path Planner* to the obtained steps. Figure 5.4 represents the path generation on a single parcel while 5.5 on the entire map when it is desired.

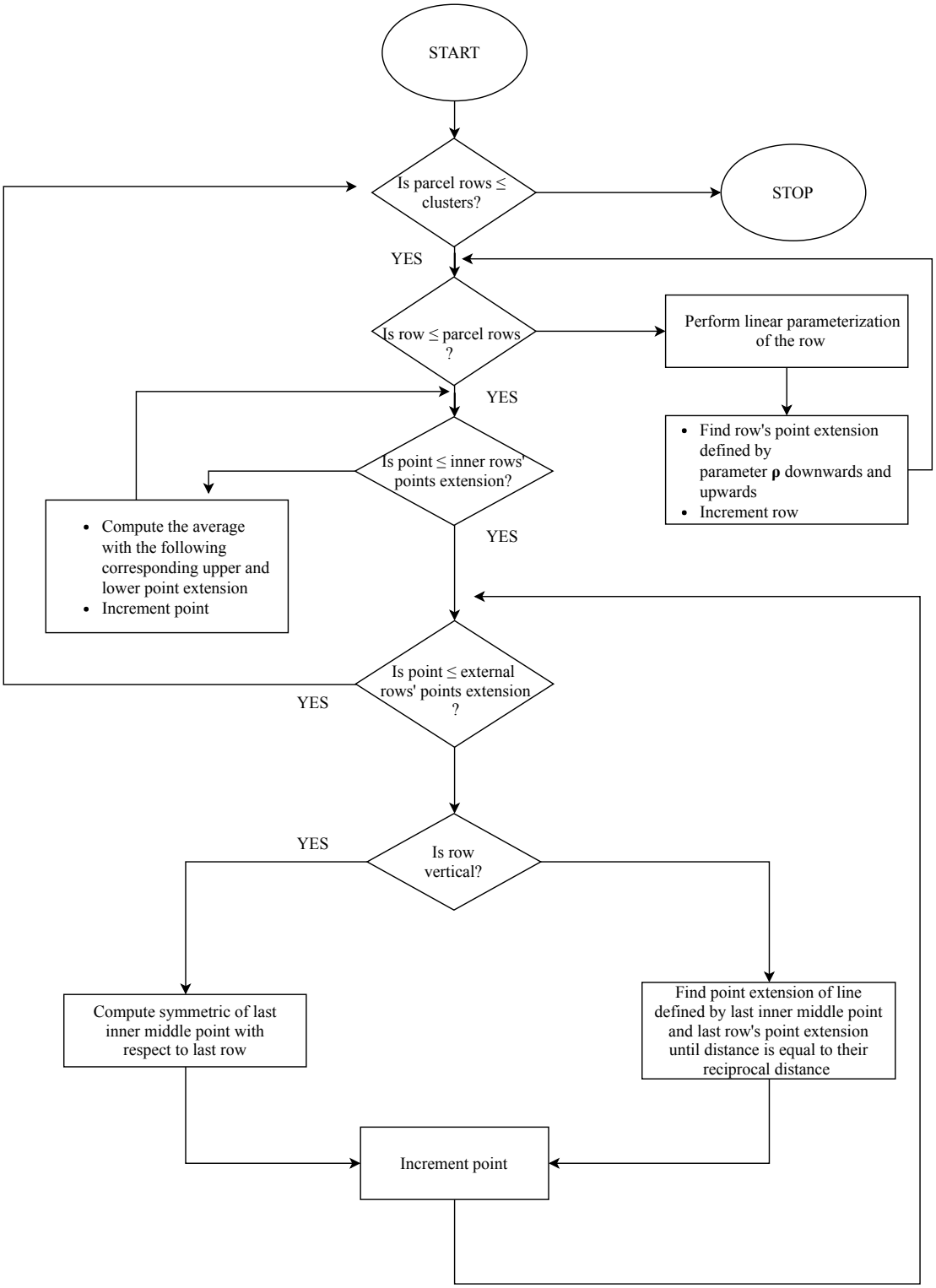


Figure 5.3: Steps generation flowchart

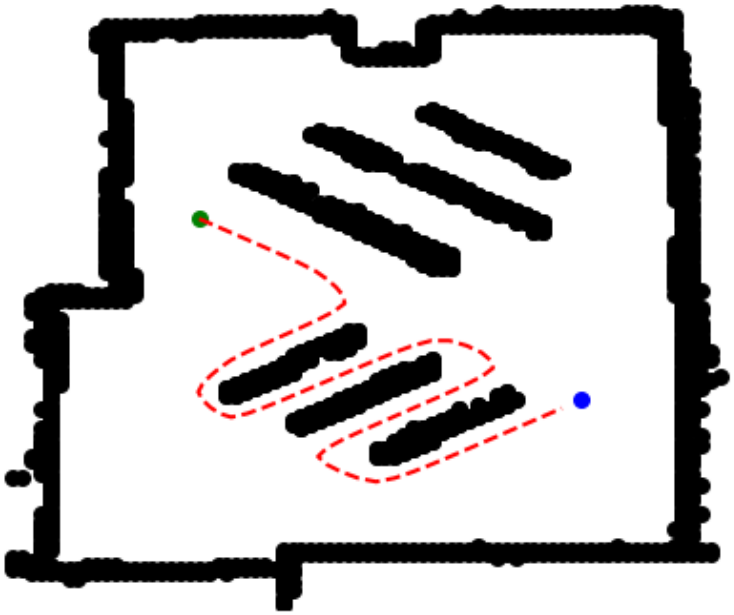


Figure 5.4

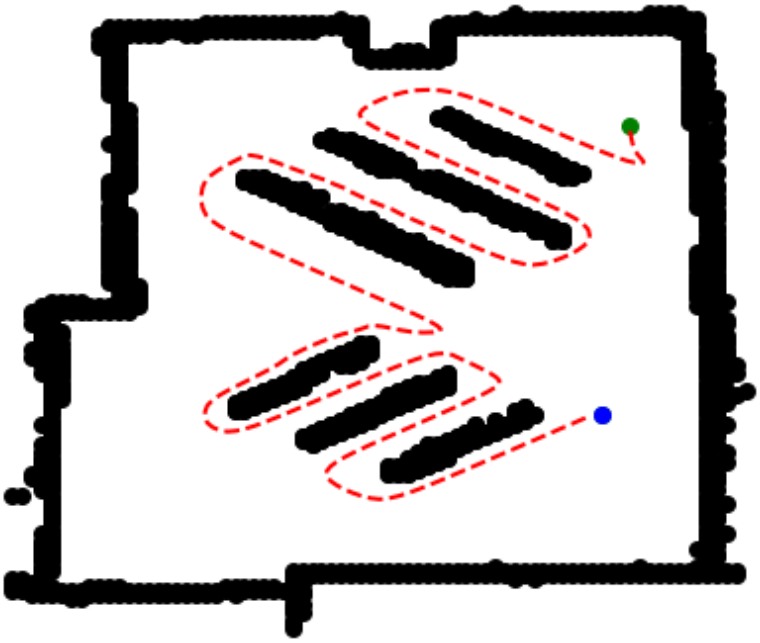


Figure 5.5

Chapter 6

Experimental Results and Conclusion

This section presents the obtained results of the work evaluated on a part of a real agricultural environment map taken by a UAV, as represented in figure 6.1.

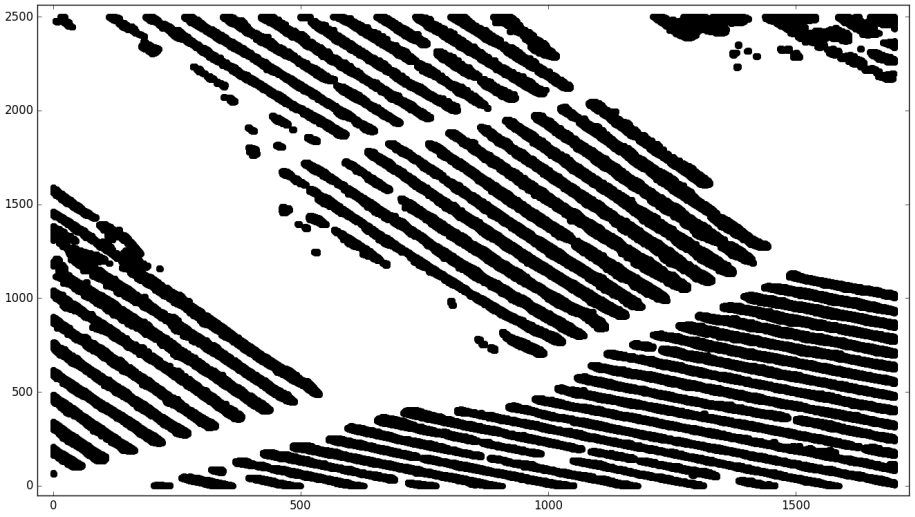


Figure 6.1: Environment Map

Firstly in figure 6.2 all the rows that compose the parcels were identified and represented with different colours in order to distinguish from each other. It can be seen that also small segments have been correctly detected

to be stand-alone clusters. Then the best line approximation for each row

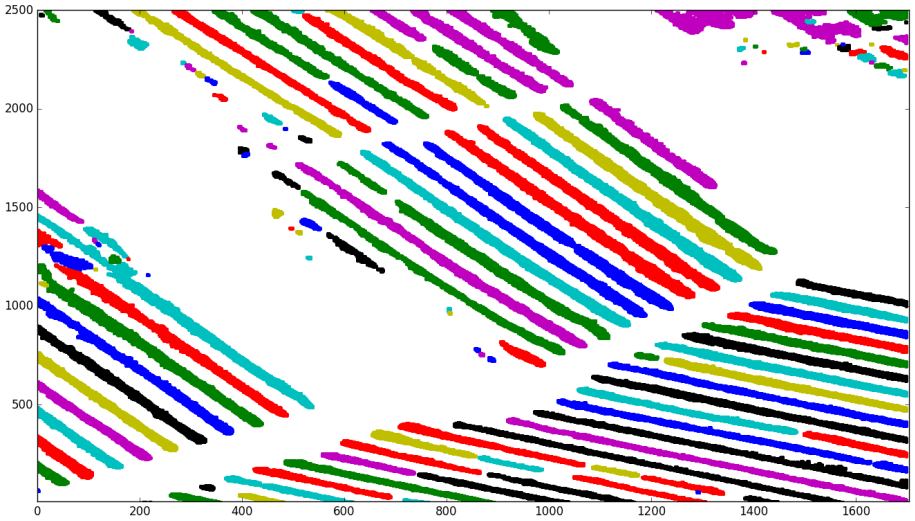


Figure 6.2: Rows Detection

was found and coloured in black in figure 6.3.

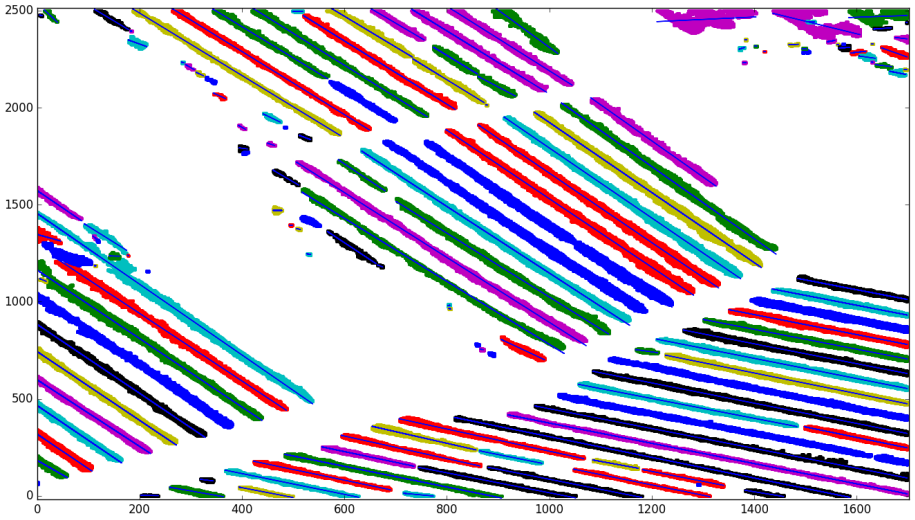


Figure 6.3: Rows Linearization

After this first preliminary processes, the parcels were recognized by means of the clustering *Dirichlet Process*. From figure 6.4 it is possible to notice that some misclassifications are present. In general they are correlated to small segments at the extremes of the parcels or in cases where two contiguous rows are split by missing plants or defects in the map.
To find a solution to this problem the *Cluster Correction* algorithm has been applied, obtaining finally the clustering in figure 6.5, where some errors still persist in particular when the row has a circular shape. In this case the linearization made through the *OLS* algorithm is not able to guarantee an adequate approximation.

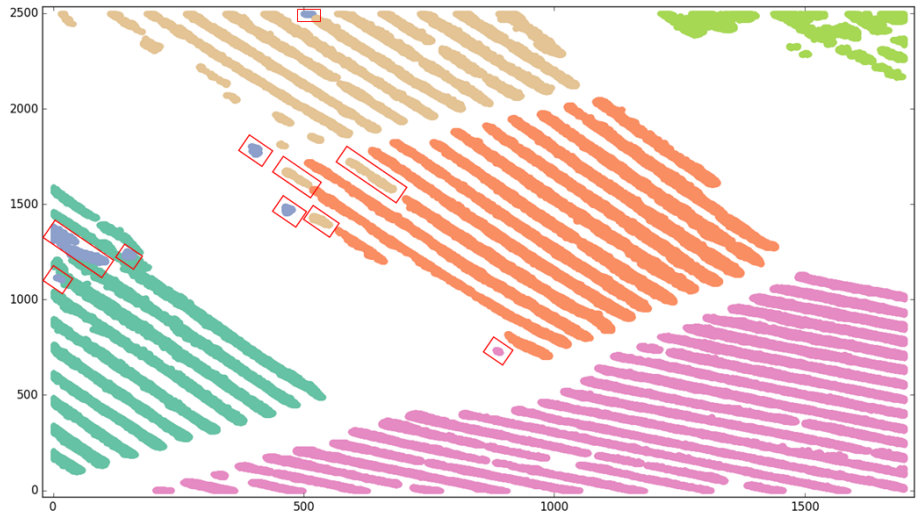


Figure 6.4: First Clustering. In red are framed the errors due to the clustering

At this stage, when the clustering has been performed, it is possible to recombine that segments of row which present defects. The figure 6.6 illustrates the impact of the *Defective Rows Correction* function, whereas in figure 6.7 it is possible to appreciate the final map which will be used by the robot.

Once the final map is obtained it can be used to find the steps (fig. 6.8) which allow the path planner to compute iteratively the robot route (fig. 6.9).

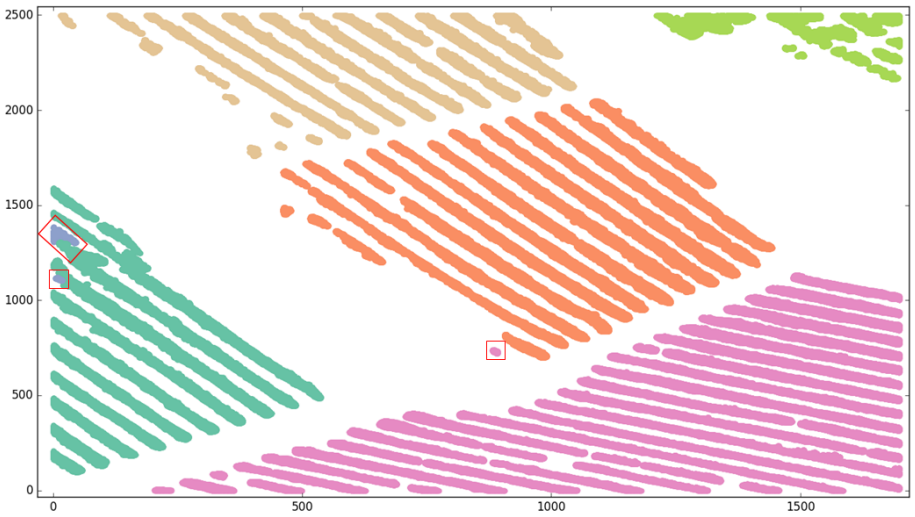


Figure 6.5: Clustering Correction. Persisting errors framed in red

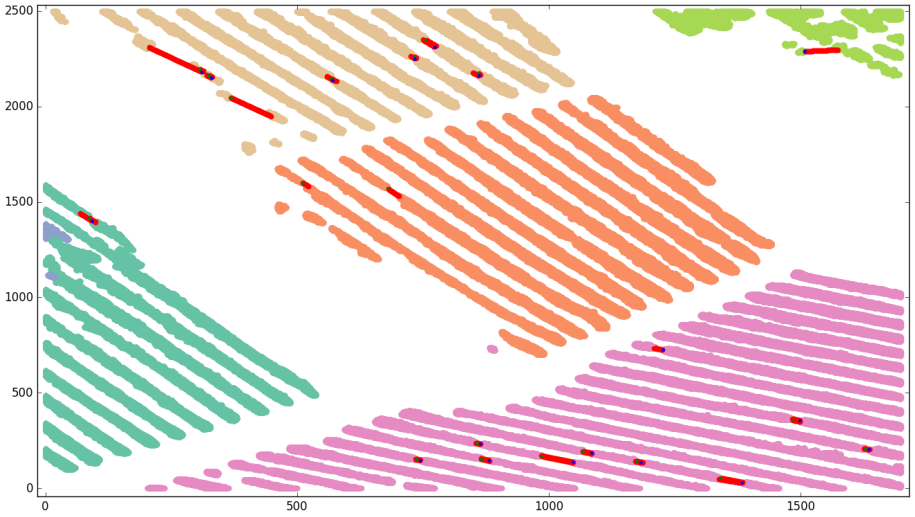


Figure 6.6: Defective Rows Correction

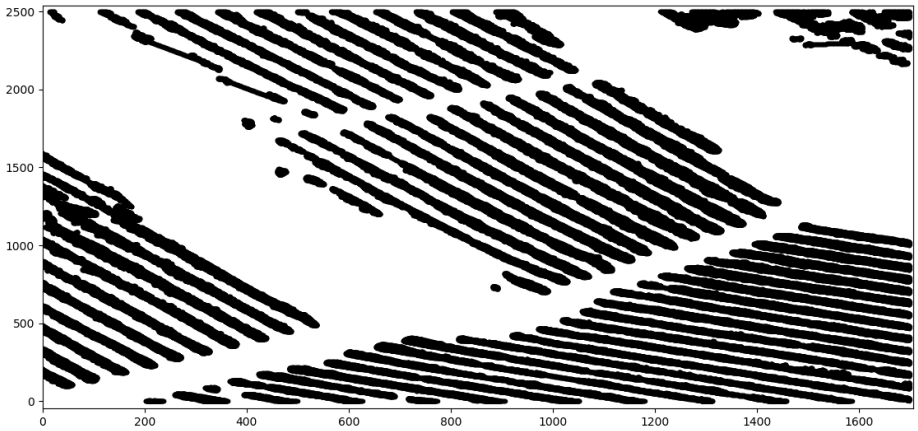


Figure 6.7: Final Map

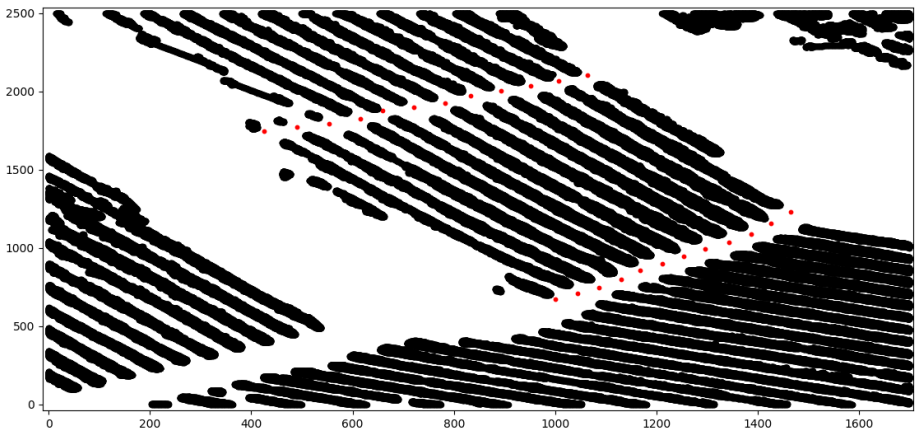


Figure 6.8: Step points for the path planner

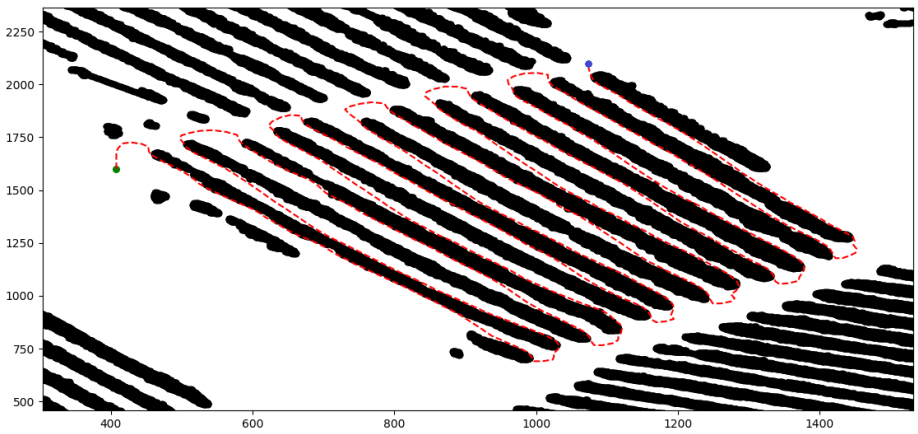


Figure 6.9: Parcel coverage

6.1 Conclusion and Future Work

The experimental results presented in this last chapter show that the work as a whole presents some weaknesses when dealing with environments that deviate significantly from the ones used in the development phase, which can be considered ideal cases where imperfections in parcels are due mainly to the presence of small areas where plants miss or defects in the image reconstruction.
For this reason the work developed can be considered a good starting point for a further researches, trying to understand how to remove the errors that come out when important environmental changes occur in the map.
Besides, it is of great importance, once all the path plan has been defined, to deploy these information in the robot in such a way that it can cover the desired environment. To do this, the future work has already been defined, consisting predominantly of developing all motion controls which will allow to obtain a *Motion Plan*, and equally important being able to localize the robot in the environment in order to apply the just mentioned motion commands.

Appendix A

*A**

```
from copy import deepcopy
import numpy

class A_star():

    def __init__(self, grid, goal, init, robot_dim):  # semiaxis dimensions

        self.dimx = len(grid)
        self.dimy = len(grid[0])
        self.grid = grid
        self.goal = goal
        self.init = init
        self.robotx = robot_dim[0]
        self.roboty = robot_dim[1]
        self.cost = [1, 1, 1, 1, 1.4, 1.4, 1.4, 1.4]
        self.step = 1
        self.reduction = 1
        self.path = []

    def heuristic(self):
        h = [[self.dimx - 1 for col in range(self.dimy)] for row in range(self.dimx)]
        x_init = self.goal[0]
        y_init = self.goal[1]
        if x_init >= self.dimx or y_init >= self.dimy:
            print "Goal out of limit"
            time.sleep(1)
```

```
            sys.exit(1)
        else:
            h[x_init][y_init] = 0
            for x in range(self.dimx):
                for y in range(self.dimy):
                    h[x][y] = round(math.sqrt((x — x_init) ** 2 + (y — y_init) ** 2), 2)
            return h

    def search(self):
        delta = [[—1, 0],  # go up
                 [0, —1],  # go left
                 [1, 0],  # go down
                 [0, 1], go right
                [—1, —1], # go up left
                [—1, 1 ], # go up right
                [ 1, —1], # go down left
                [ 1, 1 ]] # go down right
        closed = numpy.full((self.dimx, self.dimy), 0)
        closed[self.init[0]][self.init[1]] = 1
        expand = [[—1 for col in range(self.dimy)] for row in range(self.dimx)]
        action = [[—1 for col in range(self.dimy)] for row in range(self.dimx)]
        obstacle = False
        a = self.robotx
        b = self.roboty
        x = self.init[0]
        y = self.init[1]
        g = 0
        heuristic = self.heuristic()
        h = heuristic[x][y]
        f = g + h
        openm = [[f, g, h, x, y]]
        found = False  # flag that is set when search is complete
        resign = False  # flag set if we can't find expand
        count = 0
        while not found and not resign:
            if len(openm) == 0:
                resign = True
                return "Fail"
            else:
```

```
                openm.sort()
                openm.reverse()
                nextp = openm.pop()
                x = nextp[3]
                y = nextp[4]
                g = nextp[1]
                expand[x][y] = count
                count += 1
                if x == self.goal[0] and y == self.goal[1]:
                    found = True
                else:
                    for i in range(len(delta)):
                        x2 = x + self.step * delta[i][0]
                        y2 = y + self.step * delta[i][1]
                        if x2 >= 0 and x2 < self.dimx and y2 >= 0 and y2 < self.dimy:
                            if x2 - a >= 0 and x2 + a < self.dimx and y2 - b >= 0 and y2 + b
    < self.dimy:
                                truth_table = numpy.array(self.grid[x2 - a: x2 + a, y2 - b:
    y2 + b] == 0)
                                obstacle = False
                                if not truth_table.all():
                                    closed[x2][y2] = 1
                            else:
                                obstacle = True
                                closed[x2][y2] = 1
                            if closed[x2][y2] == 0 and not obstacle:
                                g2 = g + self.step * self.cost[i]
                                h2 = heuristic[x2][y2]
                                f2 = g2 + h2
                                openm.append([f2, g2, h2, x2, y2])
                                closed[x2][y2] = 1
                                action[x2][y2] = i
        path = []
        x = self.goal[0]
        y = self.goal[1]
        while x != self.init[0] or y != self.init[1]:
            x2 = x - self.step * delta[action[x][y]][0]
            y2 = y - self.step * delta[action[x][y]][1]
            path.append([x, y])
```

```
103             x = x2
104             y = y2
105         for i in range(len(path)):
106             n = i
107             self.path.append(path[n])
108         return self.path
109     def smooth(self, weight_data=0.2, weight_smooth=0.8, tolerance=0.000001):
110         path = []
111         for i in range(len(self.path)/self.reduction):
112             n = i * self.reduction
113             path.append(self.path[n])
114         newpath = deepcopy(path)
115         change = tolerance
116         while change >= tolerance:
117             change = 0.0
118             for i in range(1, len(path) - 1):
119                 for j in range(len(path[0])):
120                     v = newpath[i][j]
121                     newpath[i][j] += weight_data * (path[i][j] - newpath[i][j]) +
    weight_smooth * (
122                                 newpath[i + 1][j] + newpath[i - 1][j] - 2.0 * newpath[i][j])
123                     change = abs(v - newpath[i][j])
124         self.newpath = newpath
125         return newpath
```

Appendix B

Findrows

```
import numpy as np
matr = np.load('map.npy')
closed = np.full((len(matr), len(matr[0])), 1, dtype=np.uint8)
x = []
y = []
xw = []
yw = []
print type(matr)
map = np.empty_like(matr, dtype=np.uint8)
for i in range(len(matr)):
    for j in range(len(matr[0])):
        if matr[i][j] == 1:
            closed[i][j] = 0
        elif matr[i][j] == 0:
            closed[i][j] = 1
delta = [[-1, 0],  # go up
         [0, -1],  # go left
         [1, 0],  # go down
         [0, 1],  # go right
         [-1, -1],  # go up left
         [-1, 1],  # go up right
         [1, -1],  # go down left
         [1, 1]]  # go down right
ro = 1
start = [0, 0]
clusters = []
```

```
open = []
for i in range(len(matr)):
    j = 0
    found = False
    while j < len(matr[0]):
        while not found and j < len(matr[0]):
            if matr[i][j] == 1 and closed[i][j] == 0:
                found = True
                start = [i, j]
                closed[start[0]][start[1]] = 1
                open.append(start)
            else:
                j += 1
        cluster = []
        while len(open) > 0:
            next = open.pop()
            cluster.append(next)
            x = next[0]
            y = next[1]
            for r in range(1, ro + 1):
                for a in range(len(delta)):
                    x2 = x + r * delta[a][0]
                    y2 = y + r * delta[a][1]
                    if x2 >= 0 and x2 < len(matr) and y2 >= 0 and y2 < len(matr[0]):
                        if matr[x2][y2] == 1 and closed[x2][y2] == 0:
                            open.append([x2, y2])
                            cluster.append([x2, y2])
                            closed[x2][y2] = 1
        j += 1
        found = False
        if len(cluster) > 1:
            clusters.append(cluster)
```

Appendix C

Parcel Clustering

```
import numpy as np
from sklearn.mixture import BayesianGaussianMixture as DP

data = []
k = 0
for cl in clusters:
    xx = []
    yy = []
    for i in range(len(cl[1])):
        xx.append(cl[i][0])
        yy.append(cl[i][1])
    data.append([coeffs[k], sum(yy) / len(yy), sum(xx) / len(xx)])
    k += 1
data = np.array(data)
data = data.reshape(-1, 3)
clf = DP(n_components=len(data), n_init=len(data), covariance_type='full', tol=1e-12,
          weight_concentration_prior_type='dirichlet_process')
clf.fit(data)
labels = clf.predict(data)
nw_clusters = []
for j in range(len(labels)):
    nw_clusters.append([labels[j], clusters[j]])
```

Appendix D

Cluster Correction

```
import numpy as np
from copy import copy
from sklearn.linear_model import LinearRegression
from collections import Counter

def dist(x, y):
    d = (x[0] - y[0]) ** 2 + (x[1] - y[1]) ** 2
    return d
for i in range(len(clusters)):
    clusters_neighbours[i] = [[0, 0, 0, 0], [None, None, None, None], [None, None, None, None], [0, 0], False, [0, 0]]
### FIND CLUSTERS NEIGHBOURS AND THEIR LABELS
changed_labels = [0]
count = 0
start_time = time.time()
new_clusters = copy(clusters)
count = 0
while len(changed_labels) > 0:
    changed_labels = []
    clusters = copy(new_clusters)
    index = 0
        if count == 0:           ##DO THIS ONLY ONCE
      xx = []
      yy = []
      for cluster in new_clusters:
        for i in range(len(cluster[1])):
```

```
26          xx.append(cluster[1][i][0])
27          yy.append(cluster[1][i][1])
28            cluster_mean = np.mean(cluster[1], axis=0)
29            clusters_neighbours[index][3] = [float(cluster_mean[0]), float(cluster_mean[1])]
30            clusters_neighbours[index][5] = coef
31            coeff_perpendicular = -1.0 / coef[0]
32            q = cluster_mean[1] - coeff_perpendicular * cluster_mean[0]  # y = y0 - mx0 + mx
    ---> q = y0 + mx0
33            new_zero = False
34            exceeded_up = False
35            x_up = int(cluster_mean[0])
36            y_up = int(cluster_mean[1])
37            up_extension = [0, 0]
38            up_distance = 0
39            upper_limit = [0, 0]
40            lower_limit = [0, 0]
41            right_limit = [0, 0]
42            left_limit = [0, 0]
43            sign = np.sign(coef[0])
44            if sign < 0:
45                xmax = min(xx)
46                xmin = max(xx)
47            else:
48                xmax = max(xx)
49                xmin = min(xx)
50            yn = coef[0] * xmin + coef[1]
51            initial_point = [xmin, yn]
52            new_one = False
53            x = xmin
54            finished = False
55            while not new_one:
56                if sign > 0:
57                    x += -1
58                else:
59                    x -= -1
60                yn = coef[0] * x + coef[1]
61                x = int(x)
62                y = int(yn)
63                if x > 0 and x < len(map) and y > 0 and y < len(map[0]):
```

```
64                  if map[x][y] == 1:
65                      new_one = True
66                      finished = True
67                      lower_limit = [x, y]
68                  else:
69                      pass
70              else:
71                  new_one = True
72          yn = coef[0] * xmax + coef[1]
73          initial_point = [xmax, yn]
74          new_one = False
75          x = xmax
76          sign = np.sign(coef[0])
77          finished = False
78          while not new_one:
79              if sign > 0:
80                  x += 1
81              else:
82                  x -= 1
83              yn = coef[0] * x + coef[1]
84              x = int(x)
85              y = int(yn)
86              if x > 0 and x < len(map) and y > 0 and y < len(map[0]):
87                  if map[x][y] == 1:
88                      new_one = True
89                      upper_limit = [x, y]
90                      finished = True
91                  else:
92                      pass
93              else:
94                  new_one = True
95          new_zero = False
96          exceeded_right = False
97          x_right = int(cluster_mean[0])
98          right_start = [0, 0]
99          right_distance = 0
100         while not new_zero:
101             x_right += 1
102             y_right = int(x_right * coeff_perpendicular + q)
```

```
103                 if x_right > 0 and x_right < len(map) and y_right > 0 and y_right < len(map
    [0]):
104                     if map[x_right][y_right] == 0:
105                         new_zero = True
106                         right_start = [x_right, y_right]
107                     else:
108                         pass
109                 else:
110                     new_zero = True
111                     exceeded_right = True
112             if not exceeded_right:
113                 new_one = False
114                 exceeded_right = False
115                 while not new_one:
116                     x_right += 1
117                     y_right = int(x_right * coeff_perpendicular + q)
118                     if x_right > 0 and x_right < len(map) and y_right > 0 and y_right < len(
    map[0]):
119                         if map[x_right][y_right] == 1:
120                             new_one = True
121                             right_limit = [x_right, y_right]
122                             right_distance = dist(right_start, right_limit)
123                         else:
124                             pass
125                     else:
126                         new_one = True
127                         exceeded_right = True
128             new_zero = False
129             exceeded_left = False
130             x_left = int(cluster_mean[0])
131             left_start = [0, 0]
132             left_distance = 0
133             while not new_zero:
134                 x_left -= 1
135                 y_left = int(x_left * coeff_perpendicular + q)
136                 if x_left > 0 and x_left < len(map) and y_left > 0 and y_left < len(map[0]):
137                     if map[x_left][y_left] == 0:
138                         new_zero = True
139                         left_start = [x_left, y_left]
```

```
140                     else:
141                         pass
142                 else:
143                     new_zero = True
144                     exceeded_left = True
145             if not exceeded_left:
146                 new_one = False
147                 exceeded_left = False
148                 while not new_one:
149                     x_left -= 1
150                     y_left = int(x_left * coeff_perpendicular + q)
151                     if x_left > 0 and x_left < len(map) and y_left > 0 and y_left < len(map
    [0]):
152                         if map[x_left][y_left] == 1:
153                             new_one = True
154                             left_limit = [x_left, y_left]
155                             left_distance = dist(left_start, left_limit)
156                         else:
157                             pass
158                     else:
159                         new_one = True
160                         exceeded_left = True
161                 limits = [upper_limit, lower_limit, left_limit, right_limit]
162                 clusters_neighbours[index][0] = limits
163         if count >= 0:
164             limits = clusters_neighbours[index][0]
165             row_found = False
166             rows = []
167             for limit in limits:
168                 if limit == [0, 0]:
169                     vine_rows.append('None')
170             finished = False
171             k = 0
172             while k < len(clusters) and not finished:
173                 row = clusters[k][1]
174                 label = clusters[k][0]
175                 j = 0
176                 row_found = False
177                 while j < len(row) and not row_found:
```

```
178                     point = row[j]
179                     j += 1
180                     for i in range(len(limits)):
181                         if point == limits[i] and not row_found:
182                             clusters_neighbours[index][1][i] = k
183                             clusters_neighbours[index][2][i] = label
184                             rows.append(label)
185                             row_found = True
186                 k += 1
187             ### CHANGE POSSIBLE ERRORS IN LABELS
188             counted_labels = Counter(clusters_neighbours[index][2]).most_common()
189             most_common_label = (0, 0)
190             second_most_common_label = (0, 0)
191             i = 0
192             found = False
193             while i < len(counted_labels) and not found:
194                 if counted_labels[i][0] != None:
195                     most_common_label = counted_labels[i]
196                     found = True
197                 else:
198                     pass
199                 i += 1
200             found = False
201             while i < len(counted_labels) and not found:
202                 if counted_labels[i][0] != None:
203                     second_most_common_label = counted_labels[i]
204                     found = True
205                 else:
206                     pass
207                 i += 1
208             if most_common_label[1] > 1:
209                 if most_common_label[1] > second_most_common_label[1]:
210                     if cluster[0] != most_common_label[0]:
211                         cluster[0] = most_common_label[0]
212                         changed_labels.append([most_common_label, cluster_mean])
213                     else:
214                         pass
215                 elif most_common_label[1] == second_most_common_label[1] and
    clusters_neighbours[index][4] == False:
```

```
216                     clusters_neighbours[index][4] = True
217                     right_limit = clusters_neighbours[index][0][3]
218                     left_limit = clusters_neighbours[index][0][2]
219                     mean = clusters_neighbours[index][3]
220                     right_distance = dist(right_limit, mean)
221                     left_distance = dist(left_limit, mean)
222                     if right_distance < left_distance:
223                         right_neighbour = clusters_neighbours[index][1][3]
224                         label_neighbour = clusters_neighbours[index][2][3]
225                         second_neighbour = clusters_neighbours[right_neighbour][1][3]
226                         label_second_neighbour = clusters_neighbours[right_neighbour][2][3]
227                         if label_neighbour == label_second_neighbour:
228                             cluster[0] = label_neighbour
229                             changed_labels.append([label_neighbour, mean])
230                         else:
231                             pass
232                     else:
233                         left_neighbour = clusters_neighbours[index][1][2]
234                         label_neighbour = clusters_neighbours[index][2][2]
235                         second_neighbour = clusters_neighbours[left_neighbour][1][2]
236                         label_second_neighbour = clusters_neighbours[left_neighbour][2][2]
237                         if label_neighbour == label_second_neighbour:
238                             cluster[0] = label_neighbour
239                             changed_labels.append([label_neighbour, mean])
240                         else:
241                             pass
242                 else:
243                     pass
244         index += 1
```

Appendix E

Defective Rows Correction

```
new_map = np.zeros_like(map)
labels_value = np.unique(clusters[:, 0])
labeled_clusters = {}
for label in labels_value:
    labeled_clusters[label] = []
index = 0
for cluster in clusters:
    labeled_clusters[cluster[0]].append(index)
    point_connection[index] = [[], []]  # up, down
    index += 1
new_clusters = copy(clusters)
for key, val in labeled_clusters.items():
    for value in val:
        row = clusters_neighbours[value]
        coef = row[5]
        sign = np.sign(coef[0])
        upper_neighbour = row[1][0]
        label_upper_neighbour = row[2][1]
        lower_neighbour = row[1][1]
        label_lower_neighbour = row[2][0]
        if len(clusters[value][1]) > 100:
            if upper_neighbour in val and upper_neighbour != value \
                    and abs((coef[0] — clusters_neighbours[upper_neighbour][5][0])/ coef[0])
    < 0.3:
                xmax = 0
                if sign < 0:
```

```
26                  xmax = min(clusters[value][1])[0]
27              else:
28                  xmax = max(clusters[value][1])[0]
29              yn = coef[0] * xmax + coef[1]
30              initial_point = [xmax, yn]
31              plt.plot(xmax, yn, 'b.')
32              new_one = False
33              x = xmax
34              finished = False
35              connectionx = []
36              connectiony = []
37              while not new_one:
38                  if sign > 0:
39                      x += 1
40                  else:
41                      x -= 1
42                  yn = coef[0] * x + coef[1]
43                  connectionx.append(x)
44                  connectiony.append(yn)
45                  new_clusters[value][1].append([x, yn])
46                  x = int(x)
47                  y = int(yn)
48                  if x > 0 and x < len(map) and y > 0 and y < len(map[0]):
49                      if map[x][y] == 1:
50                          new_one = True
51                          upper_limit = [x, y]
52                      else:
53                          pass
54                  else:
55                      new_one = True
56              new_clusters[value][1] = new_clusters[value][1] + new_clusters[
    upper_neighbour][1]
57              new_clusters[upper_neighbour][1] = []
58              point_connection[value][0] = [connectionx, connectiony]
59              plt.scatter(connectionx, connectiony, color='r')
60          if lower_neighbour in val and lower_neighbour != value \
61                  and abs((coef[0] - clusters_neighbours[lower_neighbour][5][0])/ coef[0]) <
    0.3:
62              xmin = 0
```

```
63            coef = vine_row[5]
64            sign = np.sign(coef[0])
65            if sign < 0:
66                xmin = max(clusters[value][1])[0]
67            else:
68                xmin = min(clusters[value][1])[0]
69            yn = coef[0] * xmin + coef[1]
70            initial_point = [xmin, yn]
71            plt.plot(xmin, yn, 'g.')
72            connectionx = []
73            connectiony = []
74            new_one = False
75            x = xmin
76            finished = False
77            while not new_one:
78                if sign > 0:
79                    x += -1
80                else:
81                    x -= -1
82                yn = coef[0] * x + coef[1]
83                connectionx.append(x)
84                connectiony.append(yn)
85                new_clusters[value][1].append([x, yn])
86                x = int(x)
87                y = int(yn)
88                if x > 0 and x < len(map) and y > 0 and y < len(map[0]):
89                    if map[x][y] == 1:
90                        new_one = True
91                        lower_limit = [x, y]
92                    else:
93                        pass
94                else:
95                    new_one = True
96            new_clusters[value][1] = new_clusters[value][1] + new_clusters[lower_neighbour
    ][1]
97            new_clusters[lower_neighbour][1] = []
98            point_connection[value][1] = [connectionx, connectiony]
99            plt.scatter(connectionx, connectiony, color='r')
```

Appendix F

Steps generation

```
def lin_dist(x, y):
    d1 = abs(x[0] — y[0])
    d2 = abs(x[1] — y[1])
    return d1, d2
def line(p0, p1):
    x0 = p0[0]
    x1 = p1[0]
    y0 = p0[1]
    y1 = p1[1]
    xx = [x0, x1]
    yy = [y0, y1]
    coef = np.polyfit(xx, yy, 1)
    slope = coef[0]
    intercept = coef[1]
    return slope, intercept
def line_simmetry(p, m, q):
    x1 = p[0]
    y1 = p[1]
    x0 = (—2* m * q + x1 — x1 * m ** 2 + 2 * m * y1) / (1 + m ** 2)
    y0 = (2 * q + 2 * m * x1 — y1 + y1 * m ** 2) / (1 + m ** 2)
    return [x0, y0]
def axial_simmetry(p1, p2):
    x1 = p1[0]
    y1 = p1[1]
    mx = p2[0]
    my = p2[1]
```

```
    x0 = 2 * mx - x1
    y0 = y1
    return [x0, y0]
labels, indexes = np.unique(clusters[:,0], return_inverse=True)
labeled_clusters = {}
for label in labels:
    labeled_clusters[label] = []
i = 0
for index in indexes:
    if len(clusters[i][1]) > 0:
        labeled_clusters[labels[index]].append(clusters[i][1])
    i += 1
ro = 4
goals = []
q = 0
m = 0
for label, cluster in labeled_clusters.iteritems():
    cluster_limits = []
    for row in cluster:
        x = []
        y = []
        upper_limit = [0,0]
        lower_limit = [0,0]
        for point in row:
            x.append(point[0])
            y.append(point[1])
        if (isVertical):
            regressor.fit(-y, x)
            x_hat = regressor.predict(-y)
            plt.plot(x_hat, y, 'k')
            if sign < 0:
                xmax = min(x_hat)
                ymax = y[x_hat.tolist().index(xmax)]
                xmin = max(x_hat)
                ymin = y[x_hat.tolist().index(xmin)]
            else:
                xmax = max(x_hat)
                ymax = y[x_hat.tolist().index(xmax)]
                xmin = min(x_hat)
```

```
66                ymin = y[x_hat.tolist().index(xmin)]
67            Dx = xmax — xmin
68            Dy = ymax — ymin
69            slope = Dy/Dx
70            intercept = ymax — slope * xmax
71            coef[0] = slope
72            coef[1] = intercept
73        else:
74            regressor.fit(x, y)
75            y_hat = regressor.predict(x)
76            if sign < 0:
77                xmax = min(x)
78                ymax = coef[0] * xmax + coef[1]
79                xmin = max(x)
80                ymin = coef[0] * xmin + coef[1]
81            else:
82                xmax = max(x)
83                ymax = coef[0] * xmax + coef[1]
84                xmin = min(x)
85                ymin = coef[0] * xmin + coef[1]
86            slope = coef[0]
87            intercept = coef[1]
88        initial_point = [xmin, ymin]
89        stop = False
90        x = xmin
91        y = ymin
92        finished = False
93        if not isVertical:
94            while dist(initial_point, [x,y]) < ro ** 2 and not stop:
95                if sign > 0:
96                    x += —1
97                else:
98                    x —= —1
99                yn = slope * x + intercept
100               x = int(x)
101               y = int(yn)
102               if x > 0 and x < len(map) and y > 0 and y < len(map[0]):
103                   pass
104               else:
```

```
                    stop = True
            lower_limit = [x, y]
        else:
            x = xmin
            y = ymin — ro
            lower_limit = [x, y]
        initial_point = [xmax, ymax]
        stop = False
        x = xmax
        y = ymax
        finished = False
        if not isVertical:
            while dist(initial_point, [x, y]) < ro ** 2 and not stop:
                if sign > 0:
                    x += 1
                else:
                    x —= 1
                yn = slope * x + intercept
                x = int(x)
                y = int(yn)
                if x > 0 and x < len(map) and y > 0 and y < len(map[0]):
                    pass
                else:
                    stop = True
            upper_limit = [x, y]
        else:
            x = xmax
            y = ymax + ro
            upper_limit = [x, y]
        cluster_limits.append([upper_limit, lower_limit])
    clusters_goals = []
    for j in range(len(cluster_limits) — 1):
        xm = (cluster_limits[j][1][0] + cluster_limits[j + 1][1][0]) / 2.0
        ym = (cluster_limits[j][1][1] + cluster_limits[j + 1][1][1]) / 2.0
        xM = (cluster_limits[j][0][0] + cluster_limits[j + 1][0][0]) / 2.0
        yM = (cluster_limits[j][0][1] + cluster_limits[j + 1][0][1]) / 2.0
        limits = [[xM, yM], [xm, ym]]
        clusters_goals.append(limits)
    if len(clusters_goals) > 1:
```

```
144         first = [int(clusters_goals[0][0][0]), int(clusters_goals[0][0][1])]
145         second = [int(clusters_goals[1][0][0]), int(clusters_goals[1][0][1])]
146         m, q = line(first, second)
147         stop = False
148         x = first[0]
149         y = first[1]
150         initial_point = [x, y]
151         finished = False
152         limit = []
153         if abs(m) < 3:
154             while dist(initial_point, [x, y]) < dist(first, second) and not stop:
155                 x -= 1
156                 yn = m * x + q
157                 x = int(x)
158                 y = int(yn)
159             limit = [[x, y]]
160         else:
161             x = first[0]
162             y = first[1]
163             limit = [[x,y]]
164         first = [int(clusters_goals[0][1][0]), int(clusters_goals[0][1][1])]
165         second = [int(clusters_goals[1][1][0]), int(clusters_goals[1][1][1])]
166         m, q = line(first, second)
167         stop = False
168         x = first[0]
169         y = first[1]
170         initial_point = [x, y]
171         finished = False
172         if abs(m) < 3:
173             while dist(initial_point, [x, y]) < dist(first, second) and not stop:
174                 x -= 1
175                 yn = m * x + q
176                 x = int(x)
177                 y = int(yn)
178             limit.append([x, y])
179         else:
180             x = first[0]
181             y = first[1]
182             limit.append([x, y])
```

```
183         clusters_goals.reverse()
184         clusters_goals.append(limit)
185         clusters_goals.reverse()
186         first = [int(clusters_goals[-1][0][0]), int(clusters_goals[-1][0][1])]
187         second = [int(clusters_goals[-2][0][0]), int(clusters_goals[-2][0][1])]
188         m, q = line(first, second)
189         stop = False
190         x = first[0]
191         y = first[1]
192         initial_point = [x, y]
193         finished = False
194         limit = []
195         if abs(m) < 3:
196             while dist(initial_point, [x, y]) < dist(first, second) and not stop:
197                 x += 1
198                 yn = m * x + q
199                 x = int(x)
200                 y = int(yn)
201             limit = [[x, y]]
202         else:
203             x = first[0]
204             y = first[1]
205             limit = [[x, y]]
206         first = [int(clusters_goals[-1][1][0]), int(clusters_goals[-1][1][1])]
207         second = [int(clusters_goals[-2][1][0]), int(clusters_goals[-2][1][1])]
208         m, q = line(first, second)
209         stop = False
210         x = first[0]
211         y = first[1]
212         initial_point = [x, y]
213         finished = False
214         if abs(m) < 3:
215             while dist(initial_point, [x, y]) < dist(first, second) and not stop:
216                 x += 1
217                 yn = m * x + q
218                 x = int(x)
219                 y = int(yn)
220             limit.append([x, y])
221         else:
```

```
222             x = first[0]
223             y = first[1]
224             limit.append([x, y])
225         clusters_goals.append(limit)
226     else:
227         line_limits = cluster_limits[0]
228         m, q = line([int(line_limits[0][0]), int(line_limits[0][1])], [int(line_limits[1][0])
    , int(line_limits[1][1])])
229         if abs(m) < 3:
230             up = line_simmetry(clusters_goals[0][0], m, q)
231             down = line_simmetry(clusters_goals[0][1], m, q)
232         else:
233             up = axial_simmetry(clusters_goals[0][0], line_limits[0])
234             down = axial_simmetry(clusters_goals[0][1], line_limits[1])
235         clusters_goals.append([up, down])
236         clusters_goals.reverse()
237         line_limits = cluster_limits[1]
238         m, q = line([int(line_limits[0][0]), int(line_limits[0][1])], [int(line_limits[1][0])
    , int(line_limits[1][1])])
239         if abs(m) < 3:
240             up = line_simmetry(clusters_goals[1][0], m, q)
241             down = line_simmetry(clusters_goals[1][1], m, q)
242         else:
243             up = axial_simmetry(clusters_goals[1][0], line_limits[0])
244             down = axial_simmetry(clusters_goals[1][1], line_limits[1])
245         clusters_goals.append([up, down])
246     goals.append([label, clusters_goals])
247 for i in range(len(goals)):
248     for j in range(i + 1, len(goals)):
249         for k in range(len(goals[i][1])):
250             for l in range(len(goals[j][1])):
251                 for m in range(2):
252                     for n in range(2):
253                         d = dist(goals[i][1][k][m], goals[j][1][l][n])
254                         if d <= (2 * ro) ** 2:
255               mean = np.mean([goals[i][1][k][m], goals[j][1][l][n]], axis = 0)
256                             goals[i][1][k][m]= [mean[0], mean[1]]
257                             goals[j][1][l][n]= [mean[0], mean[1]]
```

Bibliography

[1] Charles E Antoniak. "Mixtures of Dirichlet processes with applications to Bayesian nonparametric problems". In: *The annals of statistics* (1974), pp. 1152–1174.

[2] David Blackwell, James B MacQueen, et al. "Ferguson distributions via Pólya urn schemes". In: *The annals of statistics* 1.2 (1973), pp. 353–355.

[3] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[4] Thomas S Ferguson. "A Bayesian analysis of some nonparametric problems". In: *The annals of statistics* (1973), pp. 209–230.

[5] Paolo Gay et al. "Vineyard detection from unmanned aerial systems images". In: 114 (June 2015), pp. 78–87.

[6] Donald B Gennery. "Traversability analysis and path planning for a planetary rover". In: *Autonomous Robots* 6.2 (1999), pp. 131–146.

[7] Zoubin Ghahramani and Thomas L Griffiths. "Infinite latent feature models and the Indian buffet process". In: *Advances in neural information processing systems*. 2006, pp. 475–482.

[8] Maya R Gupta. "A measure theory tutorial: Measure theory for dummies". In: (2006).

[9] Maya R Gupta, Yihua Chen, et al. "Theory and Use of the EM Algorithm". In: *Foundations and Trends® in Signal Processing* 4.3 (2011), pp. 223–296.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[11] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[12] Chris H Hugenholtz et al. "Geomorphological mapping with a small unmanned aircraft system (sUAS): Feature detection and accuracy assessment of a photogrammetrically-derived digital terrain model". In: *Geomorphology* 194 (2013), pp. 16–24.

[13] Hemant Ishwaran and Mahmoud Zarepour. "Exact and approximate sum representations for the Dirichlet process". In: *Canadian Journal of Statistics* 30.2 (2002), pp. 269–283.

[14] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed <today>]. 2001–. URL: http://www.scipy.org/.

[15] Olav Kallenberg. *Foundations of modern probability*. Springer Science & Business Media, 2006.

[16] In-So Kweon and Takeo Kanade. "High-Resolution Terrian Map from Multiple Sensor Data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992), pp. 278–292.

[17] Rasmus E Madsen, David Kauchak, and Charles Elkan. "Modeling word burstiness using the Dirichlet distribution". In: *Proceedings of the 22nd international conference on Machine learning*. ACM. 2005, pp. 545–552.

[18] Alessandro Matese et al. "Intercomparison of UAV, aircraft and satellite remote sensing platforms for precision viticulture". In: *Remote Sensing* 7.3 (2015), pp. 2971–2990.

[19] Thomas Minka. *Estimating a Dirichlet distribution*. 2000.

[20] Hans P Moravec and Alberto Elfes. "High resolution maps from wide angle sonar". In: *Proceedings of the IEEE Conference on Robotics and Automation*. 1985, pp. 19–24.

[21] Nils J Nilsson. *Shakey the robot*. Tech. rep. SRI INTERNATIONAL MENLO PARK CA, 1984.

[22] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. "Investigation of the*(star) search algorithms: Characteristics, methods and approaches". In: *World Applied Programming* 2.4 (2012), pp. 251–256.

[23] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[24] J Primicerio et al. “NDVI-based vigour maps production using automatic detection of vine rows in ultra-high resolution aerial images”. In: *Precision agriculture'15*. Wageningen Academic Publishers, 2015, pp. 693–712.

[25] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2016.

[26] Yee Whye Teh, Dilan Grür, and Zoubin Ghahramani. “Stick-breaking construction for the Indian buffet process”. In: *Artificial Intelligence and Statistics*. 2007, pp. 556–563.

[27] Yee W Teh et al. “Sharing clusters among related groups: Hierarchical Dirichlet processes”. In: *Advances in neural information processing systems*. 2005, pp. 1385–1392.

[28] Sebastian Thrun et al. “Robotic mapping: A survey”. In: *Exploring artificial intelligence in the new millennium* 1.1-35 (2002), p. 1.

[29] Thanh N Tran, Klaudia Drab, and Michal Daszykowski. “Revised DBSCAN algorithm to cluster data with dense adjacent clusters”. In: *Chemometrics and Intelligent Laboratory Systems* 120 (2013), pp. 92–96.

[30] Kiri Wagstaff et al. “Constrained k-means clustering with background knowledge”. In: *ICML*. Vol. 1. 2001, pp. 577–584.