ENGRAIS project

Auteur :
M. Gabriel FREITAS OLIVEIRA

Version du
January 15, 2020

Chapter 1

Problematic

The project Engrais has one problematic and objective: be able to navigate a robot in a plantation field, ignoring weeds and not running over the crops. To do it, we must choose how the robot will see the environment. The GPS signal is not precise enough to be able to control the robot granting that it will not run down any crops. The gyroscope sensors are not robust to control the robot fast enough, meaning that it can damage some crops. Thus, the chosen solution is to use 2 Lidar sensors (one at the front and the other at the back of the robot) to scan its surroundings, giving us a cloud of points. A great way to visualize everything discussed so far is depicted in figure 1.1, for aesthetic reasons only the front Lidar was represented.

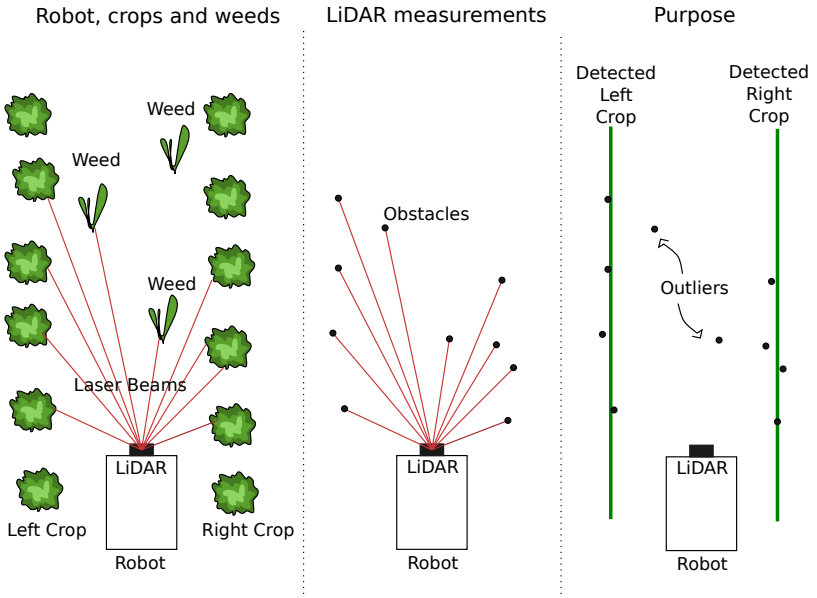

Figure 1.1: Robot and field visualization

The problem now is how to extract the best possible lines in a cloud of points. A good starting point is the Pearl algorithm [?] that will be explained in later sections, receiving a cloud of points as an input, and outputing n the best possible lines. However, this algorithm is very general and work for every cloud of points. To make it more fast and robust, we changed it to match our problematic, since we suppose that the field will have crops planted in a line, and every planted line will be parallel to each other. Once the best lines have been found, the navigation algorithm takes place. This algorithm receives the lines found, calculates the target velocities to the robot's wheels and sends it every 250 ms. Also, it takes in consideration the geometry of the field with the supposition that the lines distance are consistent between every adjacent line, and the robot will be in the center and parallel to 2 lines at the beginning (to be able to do the initialization step). Finally, the robot will move, and the control structure can continue sending target velocities. The General functioning of the robot is depicted in figure 1.2.

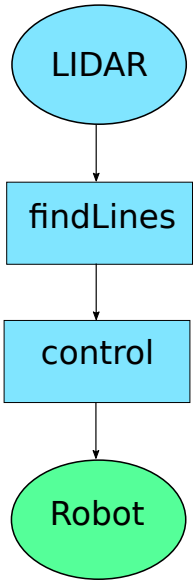

Figure 1.2: System flow chart

Finally, the last problem we have to define is how to calculate the wheels' velocity, having the the field's lines as an input. Since the input is a set of models (in the $y = ax + b$ form), a MIMO classic controller, such as a PID, would be very complex to implement and configure. Because of that, we chose to use a Fuzzy controller [?] that will be explained later. This fuzzy controller receives 2 inputs (intercept ratio and angle) and outputs 2 values (left wheel velocity, and right wheel velocity).

Chapter 2

Findlines

This chapter will explain and give a mathematical background to Pearl algorithm as well as the modifications made to our problem.

2.1 PEARL

PEARL was found to be promising due to the fact that this method usually converges in less iterations than RANSAC [**?**]. This is very interesting for real time applications such as the one at hand. It does so by using the last iteration when computing the new one, unlike RANSAC that starts over every time waiting for the residuals to be under a threshold.

2.1.1 The essentials of PEARL

PEARL is a method that aims at minimizing a function, called energy (Equation 2.1). This function represents a *score* for a set of models (lines) according to a data set of points. It allows to compare two sets of models for the same data points and thus select the one that best fit the data. The energy E is defined by

$$E(\mathcal{L}_i) = \sum_p ||p - L(p)|| + \lambda \cdot \sum_{(p,q)\in\mathcal{N}} w_{pq} \cdot \delta(L(p) \neq L(q)), \tag{2.1}$$

where:

- $\mathcal{L}_i = \{L_j\} \bigcup L_\emptyset$ is the current set of models, $L_j : f_j(x) = a_j x + b_j$ is the j^{th} model (line) and $L_\emptyset$ the empty model. The empty model is used for points that are not associated to a line (outliers). Figure 2.1 presents an illustration of points associated to models ;
- $p \in \mathbb{R}^2$ is a point extracted from the LiDAR sensor data and $L(p) \in \mathcal{L}_i$ is the model associated to the point p ($L : p \rightarrow L_j \in \mathcal{L}_i$);

- $||p - L(p)||$ is the euclidean distance between the point and its associated line ;
- $\mathcal{N}$ is the set of neighbor points and an element $(p, q) \in \mathcal{N}$ corresponds to two points p and q in the same neighborhood such that p is associated to the model $L(p)$ and q is associated to the model $L(q)$;
- $\delta(L(p) \neq L_q)$ equals 1 if $L(p)$ and $L(q)$ are not the same model, 0 otherwise ;
- $\lambda \cdot \sum_{(p,q)\in\mathcal{N}} w_{pq} \cdot \delta(L(p) \neq L(q))$ is a penalty for the placement of close points in different models. This penalty is weighted using

$$w_{pq} = \exp \frac{-||p - q||^2}{\zeta^2}, \tag{2.2}$$

  with $||p - q||$ being the euclidean distance between the points p and q ;
- ζ and λ are two constants chosen heuristically [**?**].

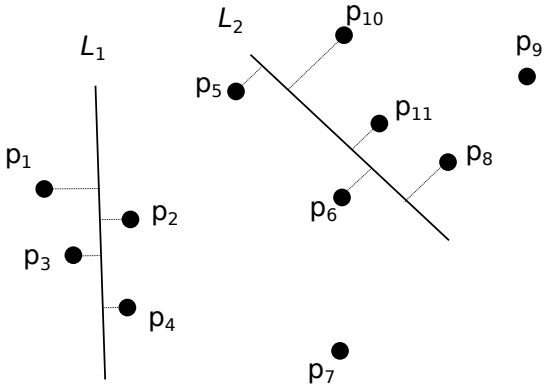

Figure 2.1: Labels example: $\{p_1, p_2, p_3, p_4\}$ are labeled with the model L_1, $\{p_5, p_6, p_8, p_{10}, p_{11}\}$ are labeled with the model L_2 and $\{p_7, p_9\}$ are labeled with the model $L_\emptyset$. For instance $L(p_2) = L_1$ and $L(p_8) = L_2$.

Outliers are points that are too far from any computed models according to a threshold. It corresponds to LiDAR points that are generated by an obstacle in the middle of the field (a weed for instance) and does not belong to any crop. On the other hand, inliers are points associated with a model.

2.1.2 PEARL Algorithm

Here we present the original PEARL algorithm [**?**]. This algorithm, detailed in Figure 2.2, searches for models (lines in our case) in a data set (LiDAR points).

The detailed of the algorithm steps:

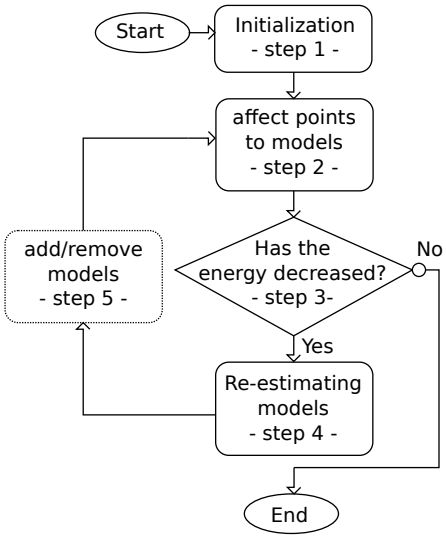

Figure 2.2: PEARL algorithm for model fitting.

1. At initialization, the algorithm randomly samples data to get $\mathcal{L}_0$, which is the first set of models, according to a defined number of initial models. It may also add the empty model $L_\emptyset$ for points that are considered as outliers ;

2. Run α-expansion [**?**] for energy described by Equation 2.1 and $\alpha \in \mathcal{L}_i$. This step appends each data point to the closest model $L_j \in \mathcal{L}_i$, or, if it is further than a threshold from all current models, into $L_\emptyset$. The label of each data point is altered, and the final set up allows for the energy calculation of that iteration (Equation 2.1) ;

3. If the calculated energy from step 2 does not decrease with respect to the previous iteration, then the procedure ends ;

4. Else, one solves Equation 2.3 and obtains a new set of models $\mathcal{L}'_i$ by replacing each model $L_j \in \mathcal{L}_i$ by a model $L_j{}'$ that better fits the points of the model according to

$$L'_j = \arg\min_{L_j} \sum_{p \in P(L_j)} ||p - L_j||, \tag{2.3}$$

   where

   - $P(L_j) = \{p | L(p) = L_j\}$ is the set of points p that are associated to the model L_j.

5. Sample more models from the points that belong to $L_\emptyset$, or merge/split current models in $\mathcal{L}'_i$ (details in Section 2.1.3). Go to step 2.

2.1.3 Implementing the step 5

It was observed that, because of the characteristics of cropping fields, the inclusion of optional step 5, described in [?], was very likely to provide better results. The basic PEARL approach that was tested during the analysis part of this article did include part of this optional step.

Merging models that are too close seems very reasonable, considering that we are treating 3-dimensional plants that do not represent a single point in space, but a small cluster. In order to extract the best possible model for a row, PEARL is expected to have a single model for a single row, this step prevents PEARL from assuming that there are two very close models if using different points of the same plant.

Furthermore, searching for new models in the outlier pool allows the algorithm to retrieve rows that may exist, but were not yet probed or had been discarded during previous iterations.

2.2 Ruby

The first improvement made to Pearl was called Ruby and only 2 little things change for this version. The first one is using the assumption that the crops will be planted in lines that are parallel to each other, thus, models that have parallel counterparts will be more valued in the next iteration. The second modification is using the assumption that the robot will move in a continuous manner, meaning that the models that were selected in the past will likely appear again in the near future. This way, differently than Pearl, this algorithm keeps the models outputed in the last call to be used again in the next call. In every other way, this algorithm behaves just like Pearl.

2.3 Ruby Genetic

The Pearl and Ruby algorithms are very similar to genetic algorithms, but not quite one. This version aims to implement a genetic Ruby algorithm, hoping it will make it more robust. The basics of a genetic algorithm are: random population generation, a fitness function to quality each individual, a reproduction method and a mutation one. Since we are focused on a set of best lines, this version only implements the population and quantification of each model. This version changes how each model is "killed" from the set while keeping changes made in Ruby. In previous versions we use a threshold to decide whether a model will stay to the next iteration or not, in Ruby genetic we use the fitness equation in 2.4, where b is the model's intercept, E_m is the model's energy, p is the number of parallel lines found and n is the number of points in the model.

$$fitness = \frac{b^2 + 10E_m}{p * n^2} \tag{2.4}$$

The final program have the following behaviour :

Algorithm 1 Ruby Genetic

```
1:  procedure RUBY GENETIC
2:      outliers ← Lidar points array
3:      E_old ← ∞
4:
5:      while it < 10 do
6:          models ← (re)Populate Population(in: outliers)
7:          models ← fuse models that are close to each other(in: models)
8:          models ← α-expansion (in: models, outliers)
9:          models, outliers ← remove models that have few points (in: models)
10:         models ← use linear fit in each model (in: models)
11:         models, outliers ← keep the 6 best models and erase the rest (in: models)
12:         E_new ← calculate new energy (in: models)
13:
14:         if E_new < E_old then
15:             bestConfig ← save best configuration(in: models, outliers)
16:             E_old ← E_new
17:         else
18:             models, outliers ← restore last best configuration (in: bestConfig)
19:
20:         it ← it + 1
21:
        return bestConfig
```

2.4 Ruby Genetic One Point

This version makes a small change, yet it makes all the difference. We suppose that the objects in the real world will give multiple points, but not really more information about the lines, this way if we concentrate a bunch of close points into a "center of mass" point, we well have significantly less inputs but no information lost, making the algorithm much faster. Besides the extra velocity, since the algorithm picks 3 random points in the field, it can select 3 points generated by the same object, making models that are usually bad, and consuming points that might be useful otherwise. To do this, we define a center of mass point as:

$$p_{cm} = \frac{\sum_{i=1}^{n} p_i}{n} \forall p_i \text{ as long } |p_i - p_{i+1}| < \varepsilon$$

Where n is the number of points summed, and ε a heuristic threshold that was set to 10cm in this project. This way we are able to calculate each object's center of mass, and use it to speed up our algorithm as well as make it more robust. Since the only thing that change is how to outliers are populated, the algorithm is very similar to Ruby Genetic. To make it easier, only the underlined bolded line was changed from Ruby Genetic to this version.

Algorithm 2 Ruby Genetic One Point

```
1:  procedure RUBY GENETIC ONE POINT
2:      outliers ← Calculate center of mass for each object in points array
3:      E_old ← ∞
4:
5:      while it < 10 do
6:          models ← (re)Populate Population(in: outliers)
7:          models ← fuse models that are close to each other(in: models)
8:          models ← α-expansion (in: models, outliers)
9:          models, outliers ← remove models that have few points (in: models)
10:         models ← use linear fit in each model (in: models)
11:         models, outliers ← keep the 6 best models and erase the rest (in: models)
12:         E_new ← calculate new energy (in: models)
13:
14:         if E_new < E_old then
15:             bestConfig ← save best configuration(in: models, outliers)
16:             E_old ← E_new
17:         else
18:             models, outliers ← restore last best configuration (in: bestConfig)
19:
20:         it ← it + 1
21:
        return bestConfig
```

2.5 Ruby Genetic One Point Positive/Negative

The next version implements a difference in the point selection step. The way the other versions do is just to select n random points in the field, but this can generate some bad models even if the field doesn't have weeds and the robot is parallel to the lines, as shown in figure 2.3.

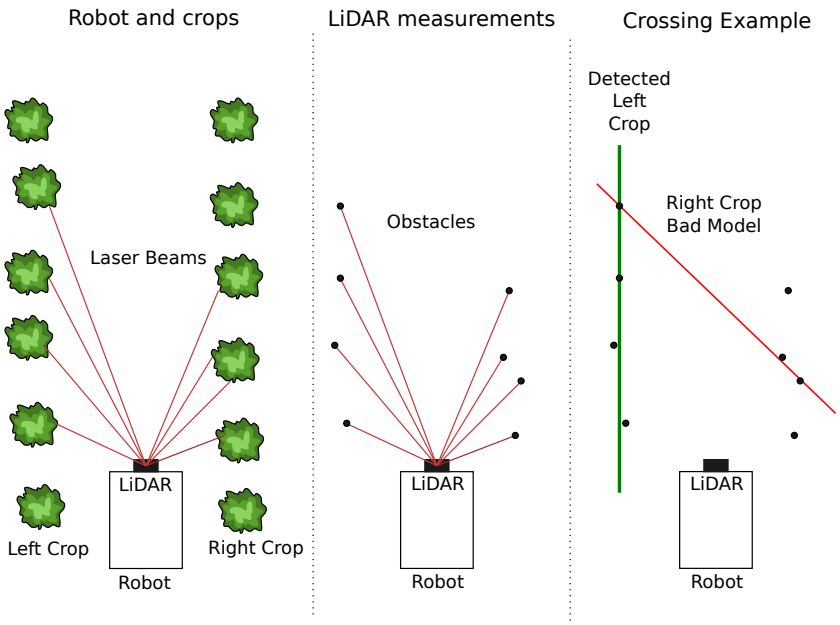

Figure 2.3: Crossing example

In this example the right model was found first, removing the 3 added points from the outliers set, and then the left model used the 3 remaining points from the left crop, leaving the 2 remaining points as outliers. Since there are few outliers, the 2 models have very closed points, and its energy will be small, it is very possible that this configuration will be returned as best models in this particular case. Even though this scenario is somewhat unlikely, and this example have very few points, this reasoning can be used to predict problems in some iterations. To reduce the probability of this kind of crossing, this version limits which points can be selected during the (re)populate models step. Separating each model into positive ($b \leq 0$) and negative ($b < 0$), and points into positive ($p_y \leq 0$) and negative ($p_y < 0$), instead of searching k models, we search for $\frac{k}{2}$ positive and negative models, and those models can only pick positive and negative points, respectively. This way, the probability of crossing models is reduced significantly, and even reduced to 0 if there are no weeds and the robot is parallel to the crops. As in the last time, the pseudo algorithm is very similar, with the changed being in bold and underlined.

Algorithm 3 Ruby Genetic One Point Positive/Negative

```
1: procedure RUBY GENETIC ONE POINT POSITIVE/NEGATIVE
2:     outliers ← Calculate center of mass for each object in points array
3:     E_old ← ∞
4:
5:     while it < 10 do
6:         models ← (re)Populate Population, separating positive from
                    negative models(in: outliers)
```

```
7:      models ← fuse models that are close to each other(in: models)
8:      models ← α-expansion (in: models, outliers)
9:      models, outliers ← remove models that have few points (in: models)
10:     models ← use linear fit in each model (in: models)
11:     models, outliers ← keep the 6 best models and erase the rest (in: models)
12:     E_new ← calculate new energy (in: models)
13:
14:     if E_new < E_old then
15:         bestConfig ← save best configuration(in: models, outliers)
16:         E_old ← E_new
17:     else
18:         models, outliers ← restore last best configuration (in: bestConfig)
19:
20:     it ← it + 1
21:
    return bestConfig
```

2.6 Ruby Genetic One Point Positive/Negative Infinity

The last idea came from the same example depicted in figure 2.3. As one can see, a bad model can consume good points, and since the points can only be in one model, this means they cannot be used again to find a better model. Using this logic, this version made not consumable points that can be used by multiple models, hoping that the algorithm would be more robust by doing so. However, this modification is slower as a side effect, since all points will be checked at all times. The pseudo code this time is quite different, and can be expressed as:

Algorithm 4 Ruby Genetic One Point Positive/Negative Infinity

```
1: procedure Ruby Genetic One Point Positive/Negative Infinity
2:     points ← Calculate center of mass for each object in points array
3:     E_old ← ∞
4:
5:     while it < 10 do
6:         models ← (re)Populate Population separating and reusing points (in: points)
7:         models ← fuse models that are close to each other(in: models)
8:         models ← α-expansion (in: models, points)
9:         models ← remove models that have few points (in: models)
10:        models ← use linear fit in each model (in: models)
11:        models ← keep the 6 best models and erase the rest (in: models)
12:        E_new ← calculate new energy (in: models)
13:
```

14: **if** $E_{new} < E_{old}$ **then**
15: $bestConfig \leftarrow$ save best configuration(in: $models$)
16: $E_{old} \leftarrow E_{new}$
17: **else**
18: $models \leftarrow$ restore last best configuration (in: $bestConfig$)
19:
20: $it \leftarrow it + 1$
21:
return $bestConfig$

Now that is clear how the findlines box works, we can move to the navigation and control algorithm.

Chapter 3

Navigation and Control

This algorithm receives all models found by findlines, and has to select which ones are more suitable to the field. Once the lines are selected, we use a state machine and a fuzzy controller to navigate the robot in the field.

3.1 Initialization

The navigation algorithm has first an initialization process. We first assume that all crops lines distance is consistent between each consecutive row, the robot is in the center of 2 lines and parallel to both at the beginning. This process listens to the findlines node and filters all models, searching from the closest pair of nodes that are equidistant and have slope close to 0. Once the a pair is found, the distance between two models are calculated as shown in 3.1

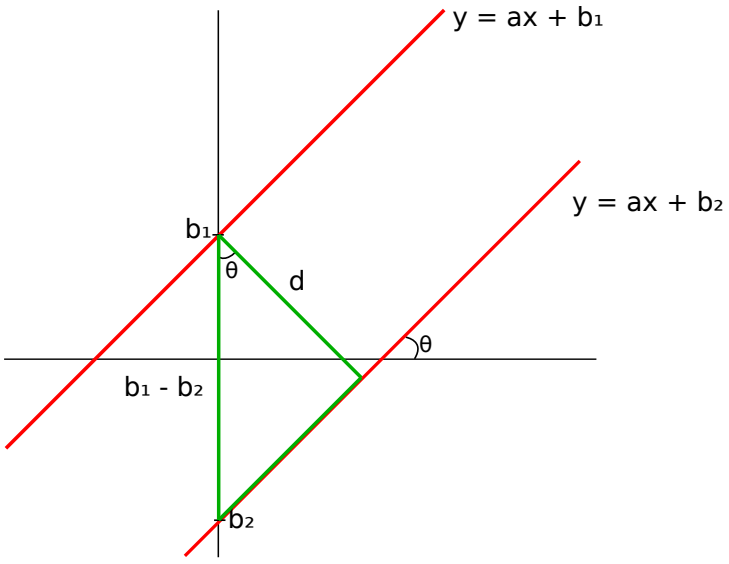

Figure 3.1: Distance between two parallel lines

This way we can very easily calculate the distance d, knowing that $tg(\theta) = a \rightarrow \theta = arctan(a)$

$$cos(\theta) = \frac{d}{|b_1 - b_2|}$$
$$d = cos(\theta) * |b_1 - b_2|$$
$$d = cos(arctan(a)) * |b_1 - b_2|$$

Knowing $cos(arctan(x)) = \frac{1}{\sqrt{x^2+1}}$

$$d = \frac{|b_1 - b_2|}{\sqrt{a^2+1}}$$

Repeating this process 10 times, we are able to have an average distance between two consecutive lines. This information is then stored and cannot be changed after the initialization process. The next step is to navigate the robot through the field using this stored information.

3.2 Predicting and Detecting Lines

After the initialization process, we populate n models where $\frac{n}{2}$ is to the left and right of the robot (we chose $n = 4$ as a default value). This way we are able to greatly increase our accuracy in selecting the models. The way we do this is by filtering the models sent by the findlines that are close to the models in the last iteration. This reasoning is valid as long as the robot moves in a linear, relatively slow manner. Since we cannot predict how much the robot will move, the delta limit is increased if nothing is found, until an upper limit is reached (in this case the robot will stop its movement, since it was not able to find any lines). Once a single model is found, we can use the distance information stored in the initialization process to calculate the other models, and proceed with the navigation. The pseudocode of how it works is :

Algorithm 5 Model Selection

```
1: procedure MODEL SELECTION
2:     candidateModels ← Models sent by findlines
3:     models ← Models from last iteration
4:     rModels ← Empty set
5:     d ← Distance between 2 adjacent lines stored in initialization
6:
```

```
7:  for delta = 0.05; modelsFound == 0 and delta ≤ 0.3; delta+ = 0.05 do
8:      for i = 0; i < 4; i = i + 1 do
9:          for each model m in candidateModels do
10:             slopeDelta ← calculates |model_i − m| for slope
11:             interceptDelta ← calculates |model_i − m| for intercept
12:
13:             if slopeDelta ≤ delta and interceptDelta ≤ 1.2 ∗ delta then
14:                 rModels_i ← m
15:                 modelsFound ← modelsFound + 1
16:
17: for int i = 0; i < 4; i = i + 1 do
18:     if rModel_i is empty then
19:         rModel_i ← Calculates model using the found models and d
20:
    return rModels
```

For visualization purposes we set flags to tell if the model was calculated or found, sending their visualization in yellow or green, respectively. With this decision, we can predict models (calculated models are said to be "predicted" as the algorithm predicts it should be there, even if it doesn't find it) and better filter the models coming from findlines.

3.3 Fuzzy controller

A fuzzy controller is a different approach to control a dynamic system. Classic control theory needs a mathematical model, often in Laplace or Z transform, to be able to configure a good controller. This complexity only gets worse when the controller have multiple inputs and multiple outputs. Since this problem would be extremely difficult to model, we chose to use a fuzzy controller instead of a classic one, skipping this process and not having to worry with MIMO modeling. To configure a fuzzy controller we just need to define the inputs, the fuzzy rules, and how to obtain the outputs. How a fuzzy controller works can be better understood in [**?**]. Knowing this, firstly we have to define the input and their membership functions.

3.3.1 Inputs

The inputs that the controller needs to receive are the distance from the center of the 2 lines, and the angle the robot makes with the center line. The angle is easy to get, we just need to calculate $\frac{\sum arctg(a_i)}{n}$ using all models. The distance to the center is a little more complicated to define. Using absolute values such as "1m is very far away" cannot be done, since we don't know the distance between two lines a priori. The solution we came for this problem is to use a "ratio" between -1 and 1, where -1 is above the left model, 0 is the

center, and 1 is above the right model, this way we can define the ratio in function of the closest right and left models' intercept, respectively named b_r and b_l,

$$ratio(b_l, b_r) = \begin{cases} \frac{|b_l|}{|b_r|} - 1 & \text{if } |b_l| < |b_r| \\ \\ 1 - \frac{|b_l|}{|b_r|} & \text{otherwise} \end{cases}$$

Remember that, in the robot's perspective, it is at rest and is the world that is moving, meaning that if the robot is above the left crops it will calculate $b_l = 0$ and $b_r = -d$ where d is the distance between the two lines, assuming the robot is parallel to them. Having both inputs defined, we can now proceed to their membership function as shown in figure 3.2.

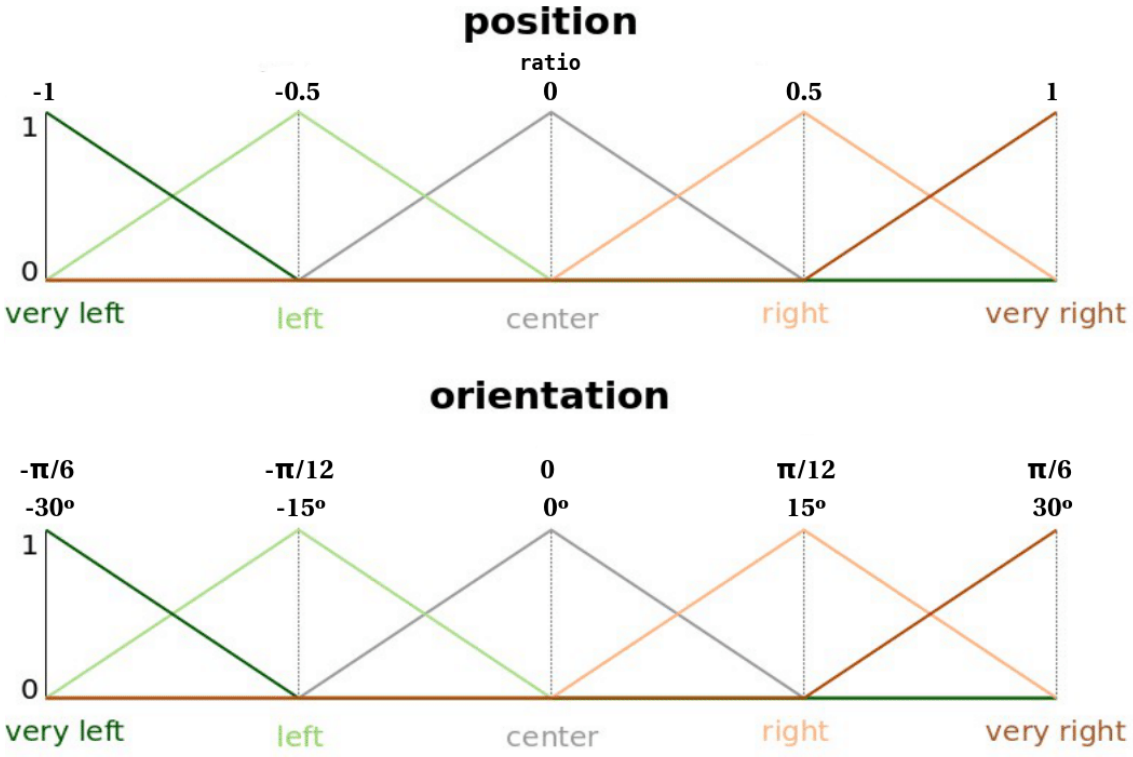

Figure 3.2: Input Membership Functions

3.4 Outputs

The outputs are really straight-forward. The fuzzy controller receives its inputs and then calculates the target velocity for the left and right wheel, their membership functions are the same and they were defined as depicted in figure 3.3.

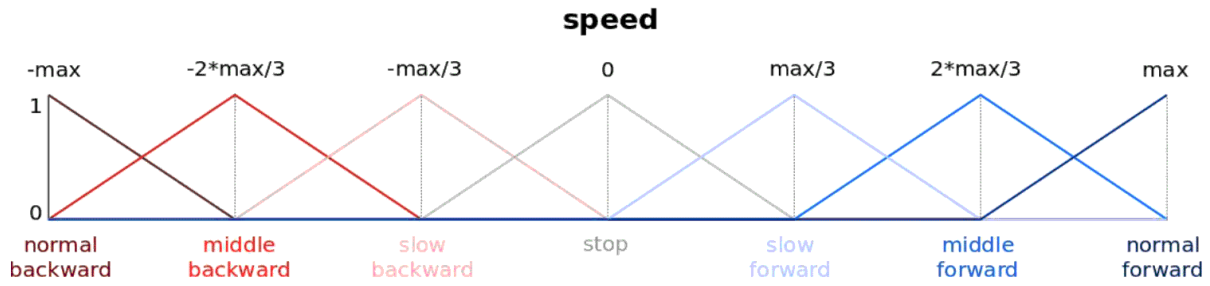

Figure 3.3: Output Membership Functions

3.5 Rules

The last part of the fuzzy controller is to define its rules. These rules aim to keep the robot always at the center and parallel of the two closest lines. Heuristically, we came up with the chart shown in figure 3.4.

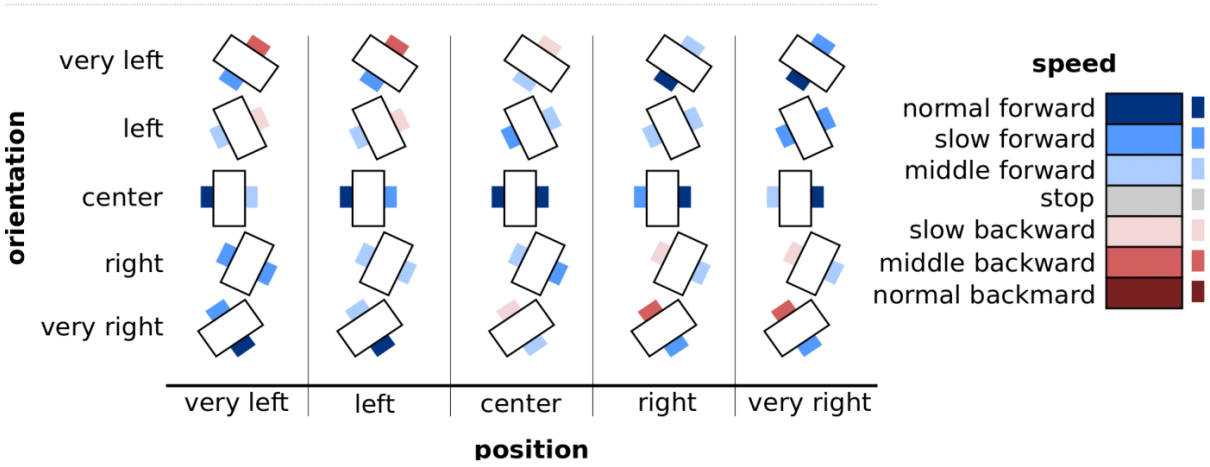

Figure 3.4: Rules Chart

This way, we have the following rules for each wheel:

Left side

orientation \ position	very left	left	center	right	very right
very left	middle forward	middle forward	slow forward	normal forward	normal forward
left	slow forward	slow forward	middle forward	slow forward	middle forward
center	normal forward	normal forward	normal forward	middle forward	slow forward
right	middle forward	slow forward	slow forward	slow backward	slow backward
very right	middle forward	slow forward	slow backward	middle backward	middle backward

Right side

orientation \ position	very left	left	center	right	very right
very left	middle backward	middle backward	slow backward	slow forward	middle forward
left	slow backward	slow backward	slow forward	slow forward	middle forward
center	slow forward	middle forward	normal forward	normal forward	normal forward
right	middle forward	slow forward	middle forward	slow forward	slow forward
very right	normal forward	normal forward	slow forward	middle forward	middle forward

Figure 3.5: Fuzzy Rules

Lastly, we have to define witch operations to use. In our case we used minimum as *And* operator, maximum as the *Or* operator, Algebraic Product as the Implication Method, Maximum as the Aggregation Method, and Centroid as the Defuzzification method. Having the fuzzy controller properly configured, the robot is now capable of navigating itself within the field, and move without running crops over. The next step is to be able to change rows when the robot reaches the end of the crops, and for this we use a finite state machine.

3.6 Finite State Machine (FSM)

The finite state machine have to decide when the robot needs to stop its linear motion, and start doing the maneuver. To do this, we implemented a Meanly state machine which receives the selected models as explained in section 3.2 and its transition outputs the target velocity for each wheel. The first state is called **Initial**, and it stops the robot until the FSM receives enough information to decide if the robot has to move forwards or backwards. Once the FSM receives a set of models, and it recognizes the positive-most point of any of them is positive, it makes a transition to **Forward** state, otherwise it goes to the **Backward** state. Remember that the models given to the FSM contain all the points in the field that are close to them.

Once the robot is making a linear movement (i.e. **Backwards** or **Forwards** states), it uses the Fuzzy controllers configured in section 3.3 to keep the robot parallel and at the middle of the two nearest lines. If the robot is going forwards, once it detects the positive-most point of all models are negative, it knows the robot has reached the end of the line and it has to start the maneuver, sending a transition to turn the robot. For simplicity sake, we will only explain the forward and left turn maneuvers, since every other

state is equivalent to one of them, but with a slightly modification. Keep in mind that all transitions have an intermediate state to stop its current movement, so that the robot has no overlapping velocities.

Knowing that, once the transition in made to **Turn_Begin** state, the robot start turning clockwise until the field's average angle is about 30°. Once this threshold is reached, a transition is made to **Turn_Mid**, moving the robot linearly either forward of backwards depending if the robot has to turn to the left or right. After selecting the sense, the FSM receives 4 models (2 to the right and 2 to the left), and then selects the two closest right, the two closest left, or the closest models depending on the selected sense. Suppose that the robot will turn to the left, in this scenario the FSM will receive b_1, b_2, b_3 and b_4 and select b_1 and b_2. Knowing that the intercept b is positive if it's to the left of the robot, and negative if to the right, we can calculate $|b_1 + b_2|$ and if this is close to 0, it means the robot has moved enough and that it is in the middle of the two closest left models. Next, the robot makes a transition to **Turn_Merge** and makes a clockwise turn until the field's average slope is around -18° and finally makes a transition to **Backwards** or **Forwards**, always going on the opposite way as before. This simplified FSM can be depicted in figure 3.6. Note that the states painted in red will transition to stop its linear motion, and the states painted in blue will transition to stop its angular motion. The output is in the form *condition / (Left Wheel Velocity, Right Wheel Velocity)*.

Having all navigation algorithms configured, we now proceed to test if this will work in our simulations.

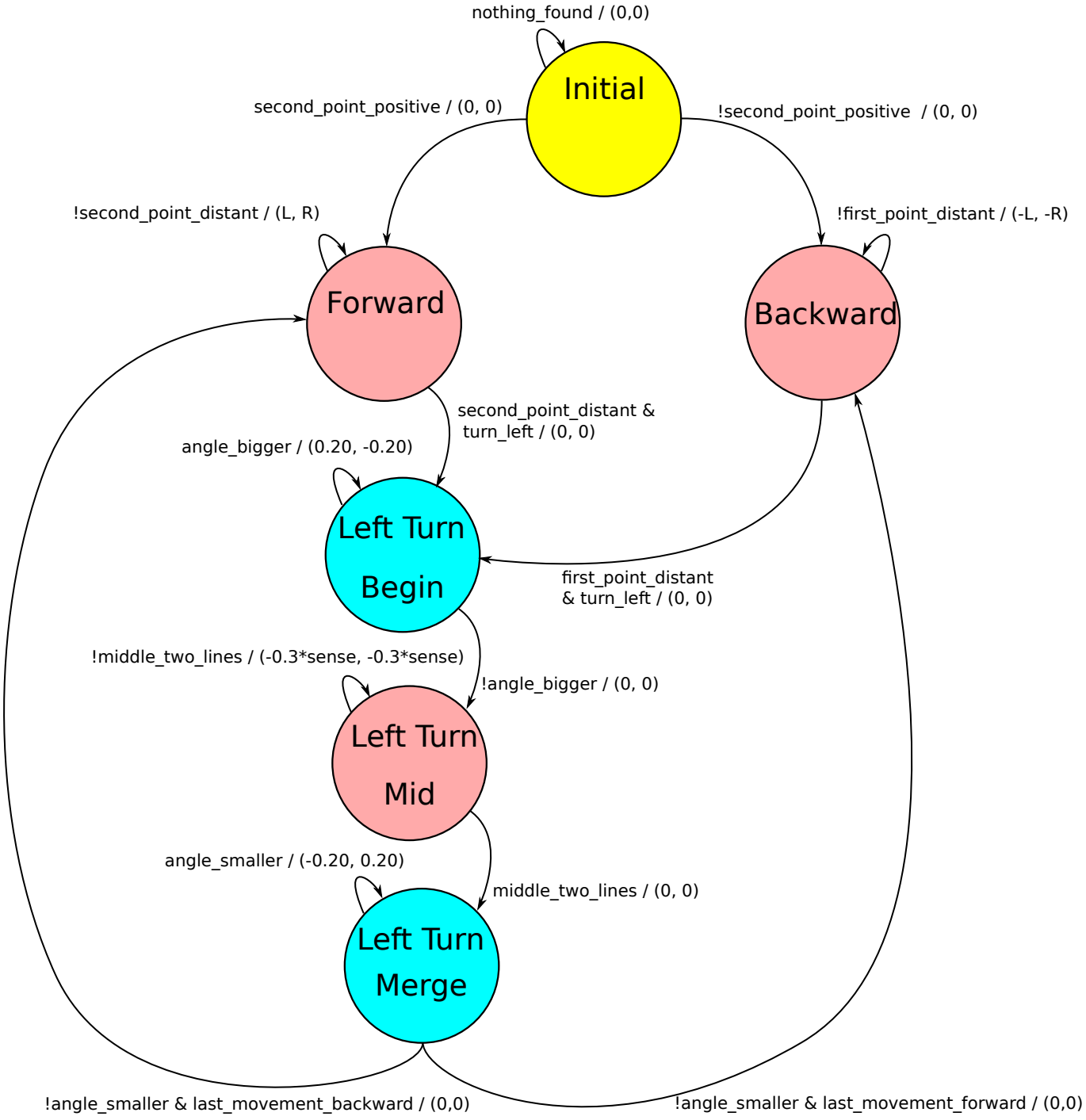

Figure 3.6: FSM flow chart

Chapter 4

Software installation

This chapter will describe how to install all the necessary tools to run the simulation and how the to setup the network in order to communicate and control the robot. First, the reader is invited to clone our Github repository. This repository have different packages for each functionality of the system, but to be used at all some packages must be installed first. In the /istiaENGRAIS/scripts there is a script named install.sh that will do all the installation and compiling process, but it is wise to follow the installation instructions in this document.

4.1 Software and Libraries

This section will show to to properly setup one's computer to run our algorithm

4.2 ROS and Fuzzylite Installation

During our development, we chose to use ROS to make the multi-thread and communication parts easier, and we use Fuzzylite library to implement our fuzzy controller described in **??**. For this reason, we must install and compile everything.

1. Install Robot Operationg System (ROS), curently the latest version is the Melodic Morenia. Close the terminal after installing.
2. Create a `catkin_ws` folder in your home directory with a folder named `src` inside it. Run the command **catkin_make** to create your workspace.
3. Copy or clone our Github repository inside `~/catkin_ws/src`.
4. In the folder `~/catkin_ws/src/istiaENGRAIS/engrais_control/src/.fuzzylite-6.0/` you will find *fuzzylite-6.0.zip*, extract it outside of istiaENGRAIS, e.g. `~/Downloads`.

5. Navigate to `~/Downloads/fuzzylite-6.0/fuzzylite/` and run **./build.sh all**
6. Copy the content inside `~/Downloads/fuzzylite-6.0/fuzzylite/release/bin` and paste it inside `~/catkin_ws/src/istiaENGRAIS/engrais_control/src/.fuzzylite-6.0/bin/`
7. Go to `~/catkin_ws` and run **catkin_make**
8. Finally, add the lines:

   source /opt/ros/melodic/setup.bash

   source $HOME/catkin_ws/devel/setup.bash

   IP=`hostname -I`

   IP=`echo "$IP" | cut -d' ' -f 1`

   export ROS_MASTER_IP=$IP

   export ROS_MASTER_URI=http://$ROS_MASTER_IP:11311

   export ROS_IP=$IP

   to the end of `~/.bashrc` and reopen your terminal.

Now ROS is all set in your machine, but we still have to install the simulation environment

4.3 Gazebo Install

The next step to run the simulation is to install Gazebo environment and some packages

1. Follow the instructions in Gazebo's site, the last version to this date is the 9.0
   If "gazebo: symbol error" try running **sudo apt upgrade libignition-math2**
2. Install the control packages

   sudo apt install ros-melodic-gazebo-ros

   sudo apt install ros-melodic-controller-manager

   sudo apt install ros-melodic-velocity-controllers

   sudo apt install ros-melodic-controller-interface

   sudo apt install ros-melodic-joint-state-controller

   sudo apt install ros-melodic-effort-controllers

4.4 Running the algorithm

To make life easier, we developed launch files to run our distributed architecture, and this subsection will quickly explain what each launch file does. To more information, go to our Github repository to consult its READ_ME. In general, launch files are very useful to run a bunch of node with common arguments. To run a ros launch command one can use

roslaunch <package> <launch_file_name> <parameter>:="<p>"

e.g. roslaunch engrais system.launch max_velocity:="2.0"

Note that the parameter is optional, since the launch file contains all necessary default values, to find the names and what each argument does, the reader is invited to open the launch file, and read the READ_ME.pdf inside engrais package. The launch files available for each package are :

4.4.1 engrais_control

- findlines.launch : launches a findlines node that receives a cloud of points and use one of the algorithms explained in section **??** to calculate the best lines
- control.launch : launches the control node that receives the best lines found by findlines and calculates the wheels' velocities

4.4.2 engrais_gazebo

- engrais_world.launch : launches gazebo simulation with robot model in place

4.4.3 engrais

- system.launch : launches 2 findlines nodes to treat front and back points data, and the control node to navigate the robot.
- simulation.launch : launches gazebo simulation, 2 findlines nodes to treat front and back points data, the control node to navigate the robot, and the node to save the robot's trajectory.

For a better simulation automation, inside istiaENGRAIS we created a *scripts* folder, having a shell script called simulate.sh, that will automatically run the simulation, making the robot navigate 5 times through each of the environments described in **??** using all algorithms in section **??**, saving the results in a Results folder 2 directories above it. After simulating, we can use the analysis.sh to draw the trajectories and have some statistics about the simulations, but to do this we have to install some new packages.

4.5 Drawing Script Setup

To Draw the trajectories, we have to install Miniconda, **remember to say "no" when the installer asks "Do you wish the installer to initialize Miniconda3 by running conda init?"**. Miniconda interferes with ROS, thus they cannot run at the same time, meaning that every time one wants to use miniconda's tools, he/she must run the command **export PATH="$HOME/miniconda3/bin:$PATH"**, changing miniconda's path if the user chose to install in another directory. After this, you must install these libraries:

sudo apt-get install python-pip

conda install -c plotly plotly=4.2.1

conda install -c plotly plotly-orca psutil requests

sudo apt -y install libgconf2-4

sudo apt install libcanberra-gtk-module libcanberra-gtk3-module

pip install plotly

pip install cufflinks

With everything installed and set up, you will be able to run ./analysis and get the statistics that are put inside the Results fonder.

Chapter 5

Simulations and Methodology

5.1 Simulations

The first thing to set up for our simulations is the environment. We chose to use the Gazebo as our environment, because of its compatibility with ROS and ease with model and simulation setup. Next, we made a robot model as shwon in figure 5.1 containing 2 wheels, 2 Lidar sensors (40 readings / second, 4 meter range, 180° angle, and 360 laser beams), 2 Wheels (for balance) and a body.

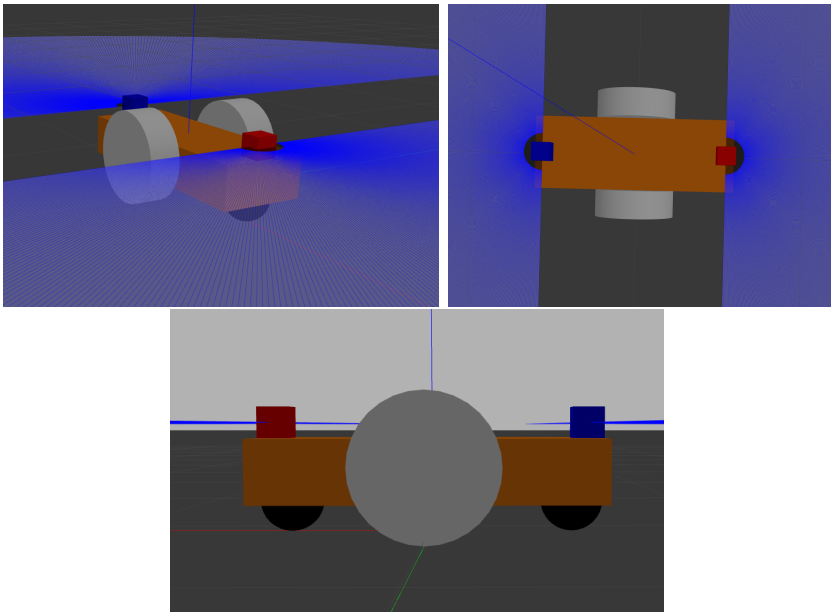

Figure 5.1: Robot Gazebo Model

This model simulates the "Lidar" and "Robot" parts depicted in figure 1.2. Using Robot Operating System, we are able to communicate with gazebo, receiving information

and sending the target velocity to both wheels. We chose to implement a communication architecture as shown in figure 6.2, where green nodes are managed by gazebo, and blue nodes implement our navigation algorithm.

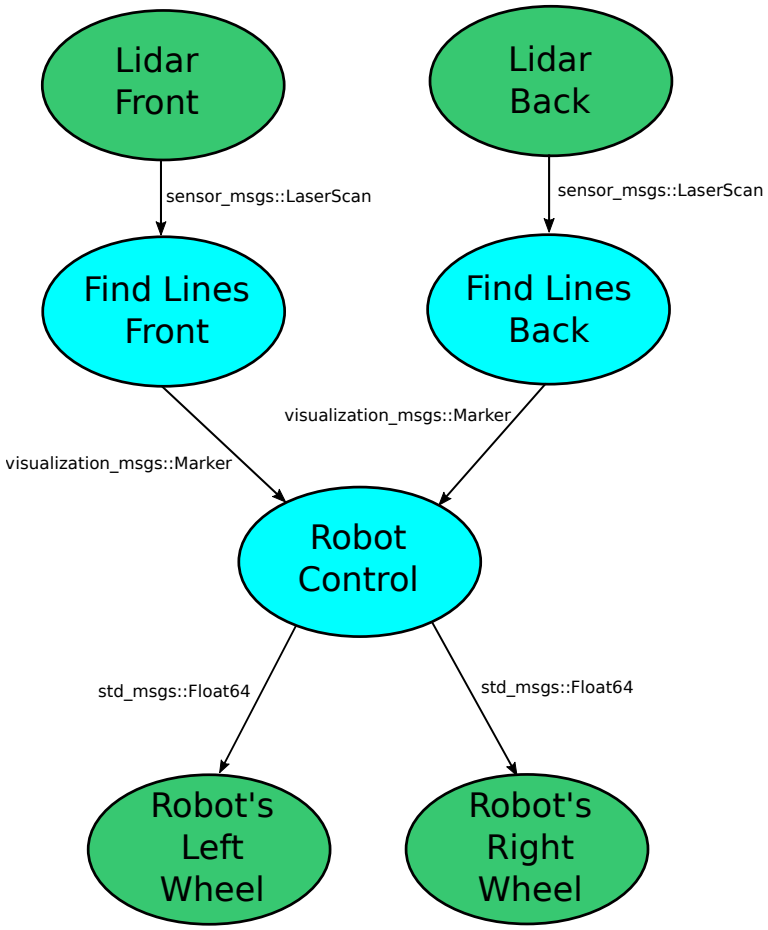

Figure 5.2: ROS nodes

Since Gazebo model has 2 Lidar sensors, we have no way to know when the messages will come, thus we had to develop a way to synchronize the information coming from back and front Lidar. The solution to this problem is the **Control** node that receives information from both Lidar sensors, and rotate and translate the points and models depending on the node that send the information for 250 ms, fusing models that appear multiple times and updating the points in the field.

Now that the robot model is set up and the navigation algorithm is capable to move the robot, we have to define some environment fields. The first field is the easy one and can be visualized in figure 5.3, having 5 rows of equally spaced crops, without weeds.

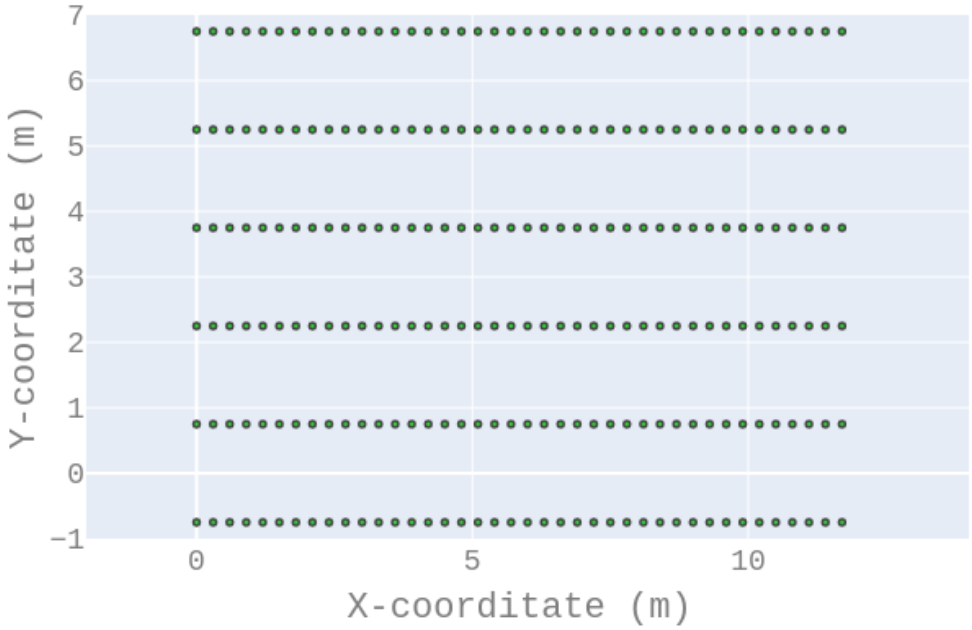

Figure 5.3: Engrais field

The Medium difficulty is called engrais2 and is depicted in figure 5.4, having 5 rows of randomly spaced crops, without weeds. Note that the distance between each row and its length changed between the two fields, testing if the algorithm is not biased and can navigate in previously unknown fields.

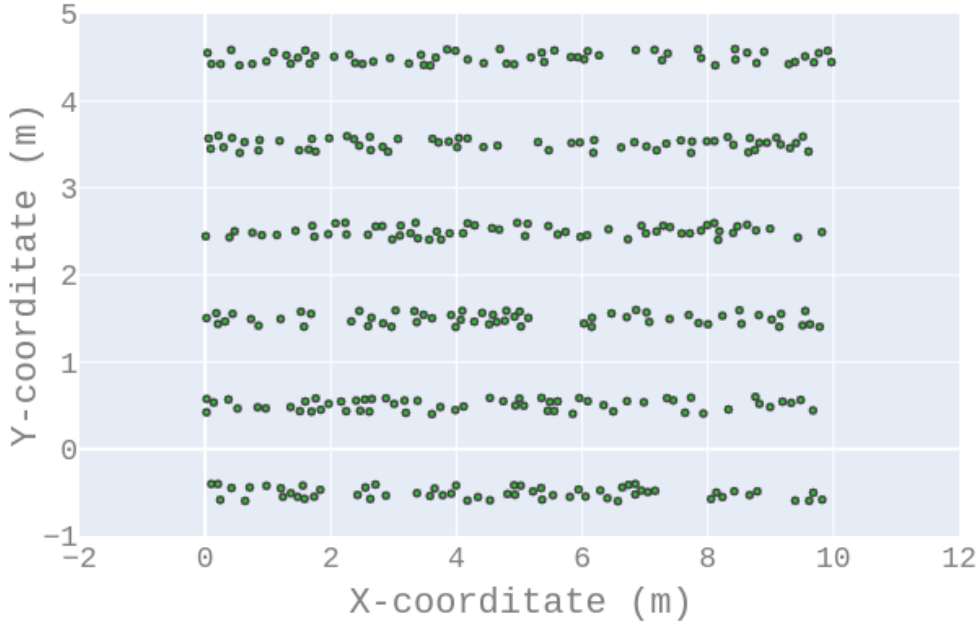

Figure 5.4: Engrais2 field

The next field is the Medium-Hard one, is called engrais3 and is depicted in figure 5.5, having 5 rows of randomly spaced crops, with 50 weeds placed randomly in the field represented as the red plants.

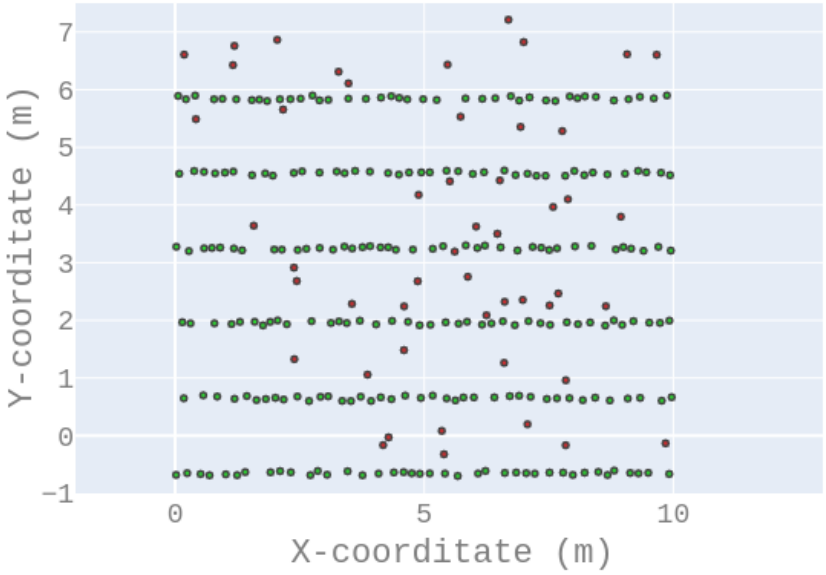

Figure 5.5: Engrais3 field

Finally, the last field is the Hard one, is called Engrais4 and depicted in figure 5.6, having 5 rows of randomly spaced crops, with big holes to simulate dead crops, a very lengthy line, and 100 weeds placed randomly.

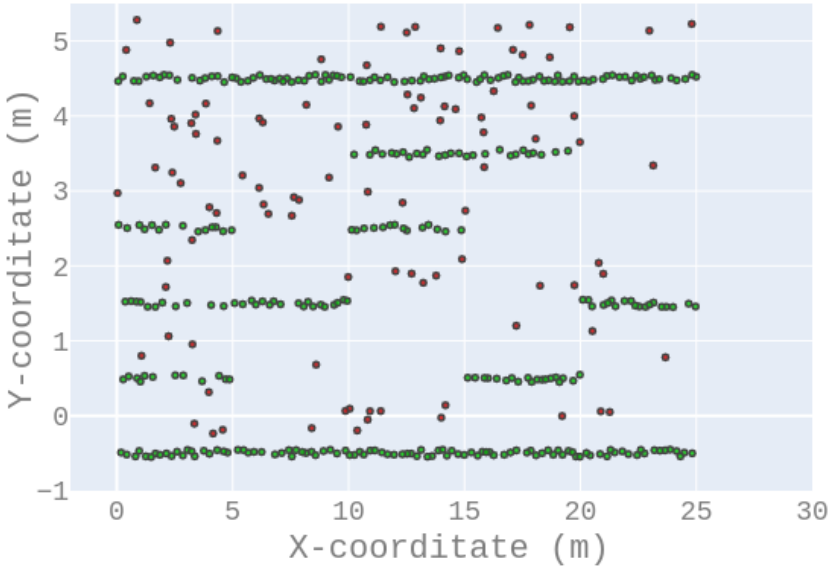

Figure 5.6: Engrais4 field

Having everything ready, the way we chose to run the simulation is to run it 5 times on each field, for every findlines algorithm described in chapter 2, getting the robot's position and the algorithm's execution time to better analyse our results.

5.2 Results

After all the simulations are completed, we first drew all trajectories for all fields and algorithms. These trajectories can be found in the appendix, but for simplicity's sake, we will just analyze the results table. The succession rate is measured from 0 to 5 complete trajectories (from start to end) without crushing plants.

Results					
Algorithm \ Environment	Engrais	Engrais2	Engrais3	Engrais4	Average(%)
Pearl	100%	60%	100%	60%	80%
Ruby	80%	80%	100%	20%	75%
RG	80%	80%	40%	0%	50%
RGOP	100%	100%	100%	100%	100%
RGOPPN	100%	100%	100%	80%	95%
RGOPPNI	100%	100%	100%	0%	75%

Mean Squared Error (m^2)					
Algorithm \ Environment	Engrais	Engrais2	Engrais3	Engrais4	Average
Pearl	5.93E-5	3E-3	1.3E-4	2.3E-3	1.3E-3
Ruby	2.9E-3	3.6E-4	1.2E-4	5.5E-3	2.2E-3
RG	8.7E-3	3.1E-3	8.9E-5	9.64E-3	5.3E-3
RGOP	6.46E-5	4.3E-4	6.66E-5	7.5E-4	3.2E-4
RGOPPN	3E-4	7E-4	1.4E-4	2.24E-3	8.4E-4
RGOPPNI	6.7E-5	3.1E-4	2.6E-4	6E-2	0.2

Average Number of Points					
Algorithm \ Environment	Engrais	Engrais2	Engrais3	Engrais4	Average
Pearl	90.8	179.35	122.79	120.29	128.3
Ruby	91	179.3	122.54	120.36	128.3
RG	91.16	178.1	124.53	120	128.4

RGOP	40.1	73.12	54.18	47.16	53.64
RGOPPN	40	73.15	54.19	47.12	53.61
RGOPPNI	40.08	73.96	54.19	47.17	53.85

Average Execution Time (ms)					
Environment / Algorithm	Engrais	Engrais2	Engrais3	Engrais4	Average
Pearl	4.09	20.89	9.06	9.27	10.82
Ruby	4.14	21.15	9.39	9.61	11.07
RG	5.25	8.12	6.56	6.11	6.51
RGOP	2.35	3.44	2.9	2.7	2.84
RGOPPN	1.58	3.57	2.8	2.37	2.58
RGOPPNI	3.3	6.37	4.42	4.05	4.5

Execution Time /100 Points					
Environment / Algorithm	Engrais	Engrais2	Engrais3	Engrais4	Average
Pearl	4.5	11.64	7.37	7.7	7.8
Ruby	4.54	11.79	7.66	7.98	7.99
RG	5.75	4.55	5.26	5.09	5.11
RGOP	5.86	4.7	5.35	5.7	5.4
RGOPPN	3.95	4.88	5.16	5.02	4.75
RGOPPNI	8.2	8.61	8.15	8.58	8.38

Since the robot's movement is not linear, the reference changes in every row, and during the change rows maneuver it is impossible to define a reference, the mean squared error is calculated only inside the rows $1m \leq err \leq end - 1\,m$. The average number of points and execution time were calculated in the first 500 iterations of the algorithm. Since the robot spends a lot of time to do the maneuver, the Lidar doesn't send a lot of points, reducing the number of points, the execution time, and the difference between each algorithm. By only calculating the first 500 iterations, we guarantee the Lidar will capt the maximum of points.

5.3 Conclusion

By analyzing the results and trajectories, we noticed a clear upgrade in later versions when compared to Pearl. The obvious winner is the Ruby Genetic One Point version, being able to perfectly navigate through all fields, while having the second fastest execution time in all cases and the smallest mean squared error. However, it is important to note that only 5 iterations were made during the simulations, meaning that 1 error would decrease the accuracy in 20%. From the trajectories and the results, it is reasonable to say that both Ruby Genetic One Point and Positive / Negative are interchangeable. Another positive aspect of the newer versions can be inferred with the execution time / 100 Points table. Note that Pearl and Ruby versions increase their time as the number of input (points) increases, while Ruby Genetic and later oscillate around the average value, without spikes or a clear sign of non linearity. This is a strong indication that the Pearl and Ruby algorithms have a polynomial complexity, probably $O(n^2)$, while Ruby genetic and newer versions have something similar to $O(n)$.

The greatest achievement of these results are the high rate of success, even in hard and medium hard fields, meaning that the navigation and control algorithms are very robust having as high as 100% success rate in engrais4, an extremely hard field for navigation without previous knowledge. These results show a notorious advancement from our debut point being inspired by [**?**], making the robot much more robust in fields that are way harder to navigate, while improving drastically the findlines execution time, and implementing a tuning function that is capable of changing rows in a very consistent way.

Chapter 6

Real Robot Implementation

This chapter aims to explain and document the configurations needed to make the real robot work. The approach we used to this problem is a distributed system, where 2 raspberry Pi receive the readings from the 2 Lidar (1 Raspberry for the front and one Raspberry for the back Lidar), possibly filter these points, and calculates the best possible lines using a findlines algorithm. These lines will be sent to a central computer, in our case a Nvidia Jetson TX-2, where the navigation algorithm takes place. Once the target velocities have been decided, the remaining 2 Raspberry Pi will receive the target velocity, and send them to the correct wheels.Trying to imitate the simulations, we built a robot as shown in figure 6.1 containing 2 wheels (Ez-Wheel, an electronic wheel that can be controlled using serial communication), 2 Lidar sensors, 4 raspberry pi, and a NVIDIA Jetson TX-2.

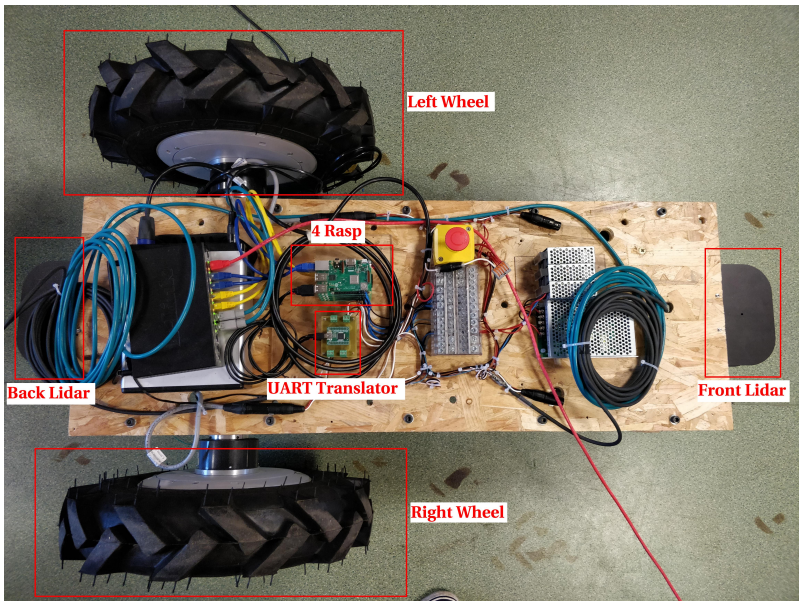

Figure 6.1: Real Robot

The first step is to install the corret OS in each component. For the PC, an Ubuntu 18.04.03 LTS was used. For the Raspberry Pi, we used a Ubuntu Rasberry 18.04.02 LTS, and for the Jetson-Tx2 a Nvidia's 18.04.3 Ubuntu. For the jetson, the installation is a bit more complex, so one will need to

1. In your PC, download Nvidia SDK manager at developer.nvidia.com/nvidia-sdk-manager (you will have to create a developer account, unfortunately).
2. Install the .deb file and run 'sdkmanager' at the terminal.
3. Log in into your account and change Target Hardware to your product (in our case Jetson TX2 (P3310)) and click continue
4. Accept the terms, click continue and put the sudo password
5. When the "SDK Manager is about to flash your Jetson TX2" screen pops up, change to Manual Setup. Click the power up button (4th one from left to right) two leds should show up. Then, hold the 3rd button, click and release the 1st button, then release the 3rd button. Finally, click the Flash button on your PC
6. After the flash is done, your Jetson Should boot up ubuntu and your PC will ask to Install SDK components on your Jetson TX2. Don't touch your PC and finish Ubuntu setup on your Jetson
7. After completing Ubuntu's setup, put jetson's Username and Password in your PC.
8. After installing everything, restart your jetson and it will be done!

After setting up the OS, we need to define one IP address for each component connected to the network, as shown in the following table :

Table 6.1: Machines Infos

Component Informations				
Component	Machine Name	Username	Password	IP
Router	—	admin	admin_siargne	192.168.10.001
Lidar Back	—	—	—	192.168.10.111
Lidar Front	—	—	—	192.168.10.112
Rasp 1	engrais-pi	engrais	engrais	192.168.10.101
Rasp 2	engrais-pi	engrais	engrais	192.168.10.102
Rasp 3	engrais-pi	engrais	engrais	192.168.10.103
Rasp 4	engrais-pi	engrais	engrais	192.168.10.104
Jetson TX-2	jetsontx2	jetson-tx2	jetsontx2	192.168.10.105
PC WIFI	usrlocal	usrlocal	usrlocal2019	192.168.10.110

6.1 Network Setup and Remote ROS

To be able to run ROS in a remote machine, some configurations must be made. The first one is having a secure SSH connection between the host and the remote machine. To do this, use the command

ssh -oHostKeyAlgorithms='ssh-rsa' user@IP

Changing "user" and "IP" acconrding to table 6.1. After doing this, remember to have in the host machine (all machines involved is recommended) the following lines in `~/.bashrc`

```
IP=`hostname -I`
IP=`echo "$IP" | cut -d' ' -f 1`
export ROS_MASTER_IP=$IP
export ROS_MASTER_URI=http://$ROS_MASTER_IP:11311
export ROS_IP=$IP
```

The final configuration needed to run a ROS node remotely is to have a configuration file in the remote machine. Since roslaunch doesn't run `~/.bashrc` in the remote machine, all the configuration made in it will not be set. The solution ROS found to this problem is to have a configuration file called "environment file" in the remote machine. This file must contain the following lines

```
export ROS_IP=`hostname -I`
. ~/catkin_ws/devel/setup.sh
exec "$@"
```

This file exports the machine's ROS_IP, run the ROS setup script, and every argument passed to it. Of course, if one needs more settings, it can be added in this file. **IMPORTANT :** the host machine will send its ROS_MASTER_URI environment variable to the remote one, so it is very important to set it up as described previously, otherwise the host machine will send "http://localhost:11311". If the remote machine receives this ROS_MASTER_URI it will try to connect to itself, not the host machine, making impossible to correctly launch the nodes.

The best solution found to this problem is to have in the ~/.bashrc the ROS_MASTER_URI be this machine's ip, this way every time one opens the terminal, ROS_MASTER_URI will be set, for example "http://192.168.10.110:11311" if connected to the PC, and when a remote note launch is called, this ROS_MASTER_URI will be transferred to the remote machine, having its ROS_MASTER_URI to be "http://192.168.10.110:11311" as well, making ROS connects to the machine having the IP 192.168.10.110. The same reasoning can be made for all other devices, making possible to launch remote nodes from every machine.

To launch a node remotely, it is necessary to define a "machine" tag in the launch file. The machine tag contains the following elements:

```
<machine   name="<arbitrary_name>"
           address="<remote_machine_IP>"
           user="<remote_machine_username>"
           password="<remote_machine_ssh_password>"
           env-loader="<path_to_environment_file>"
           default="true | false" />
```

If the default tag is true, all nodes after this tag will be used in this machine instead. For more information, see ROS documentation on machine tag. Then, to launch a node in this machine, one just needs to explicit it in the "node" tag.

```
<node   name="<arbitrary_name>"
        pkg="<package>"
        type="<node_to_be_ran>"
        machine="<remote_machine_name>"
```

From our testings, the host machine does not need to have the package nor the compiled node if the launch file only contains simple node tags. However, if the launch file contains a "include", usually to launch another launch file, the host will need to have the package, bot not necessarily the complied node (if using C++).

6.2 Distributed Architecture and Differences

With everything configured and ready to run, let's take a look at how the robot implements the architecture depicted in figure . Since the Lidar sensors have a huge range and many points, it can return some points that are not useful to our algorithm, so the robot had to implement a filter node to suppress these points before they reach the findlines algorithm. As explained later, the flow of information is the following: First, a Lidar sends its information to the network, and one rasp will be responsible to find the best possible lines in it (Rasp 3 takes care of the back Lidar, while Rasp 4 of the front Lidar). Once a rasp receives the raw points, it goes through a filter node, where all the undesirable points will be erased. Then, the filtered points will be sent to another node in this same Rasp and fed to the findlines algorithm. Once the process is complete, the best possible lines found by the algorithm will be sent to the network. To obtain the maximum amount of information possible, the findlines algorithm will loop continuously until 70% of period time is elapsed, giving us many executions of the same data-set and much more lines to work with.

After this process is done, the central machine (whether the PC or the Jetson TX2) will receive all lines found by Rasp 3 and 4 during a period of 250 ms. When this time has

passed, the central machine will do the navigation process as explained in chapter 3. Next, the central machine will send the target velocity for both wheels through the network, and wait for another 250 ms. Finally, Rasp 1 and 2 (responsible to send the target velocities to right and left wheels, respectively) will receive this information and send them to the wheels. Since the wheels must receive a target velocity every 100ms (maximum), Rasp 1 and 2 will imitate a zero order hold and send the same data over and over each 20 ms, and then change the data sent once new information comes from the central.

One vital difference from the simulation, is that we have to move the robot manually sometimes, and the real robot must have an emergency break. To solve the first problem, we implemented a node that receives directions from a PS4 controller, where the analog and the D-Pad will move the robot in that direction, while the buttons trigger or toggle the mode. A description of what which button does is depicted in figure6.2.

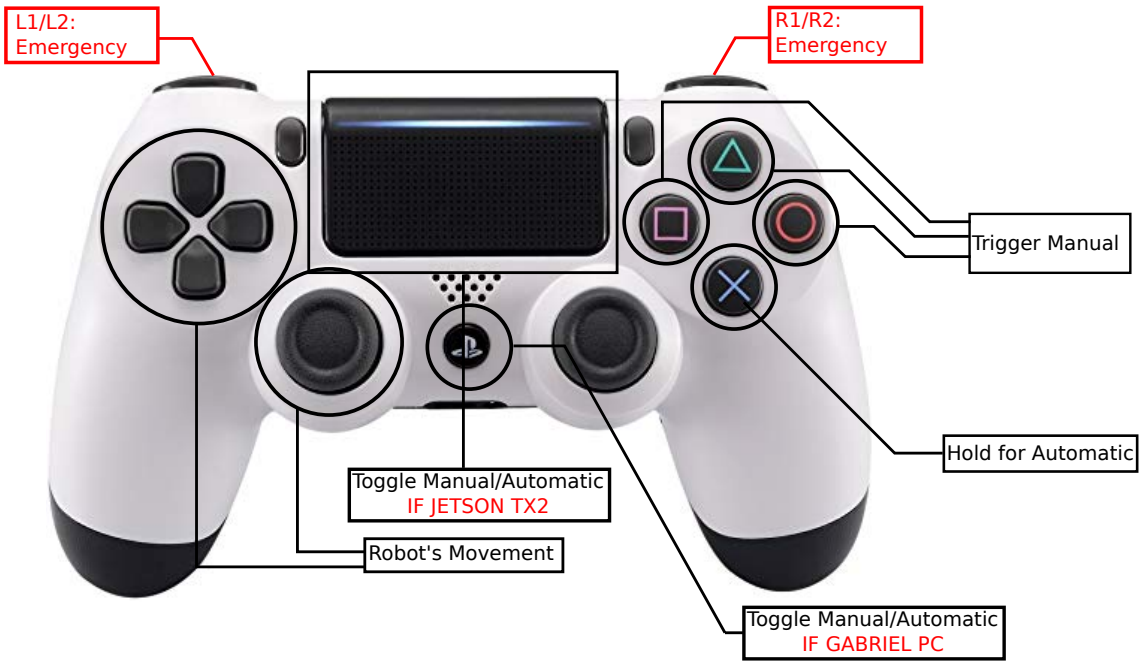

Figure 6.2: Controller Schema

The emergency brake problem was solved making every node receive a flag frequently to continue its execution. This flag has to be sent "false" at an arbitrary frequency, but if the nodes don't receive it in a 500ms interval, it will assume communication was lost and stop. Similarly, if a flag "true" is sent (i.e. when the emergency is triggered by the controller) every node will also stop. In manual mode, the node that receives the controller's data will send the flag automatically every 20ms. In automatic node this function is given to the central node, and it will send the flag every 20 ms as well. Similarly, a node can be used by a PC to control the robot using the keyboard's arrows, and change modes using the space bar, but due to its limitations it is vastly inferior to the controller. All these

process is depicted by figure 6.3, where green nodes come from peripherals outside our direct control (Lidar sensors and Controller), yellow nodes are processed by Rasp 4, brown nodes by Rasp 3, the blue node by PC/Jetson TX2, the purple node by Rasp 2, and the light red node by Rasp 1.

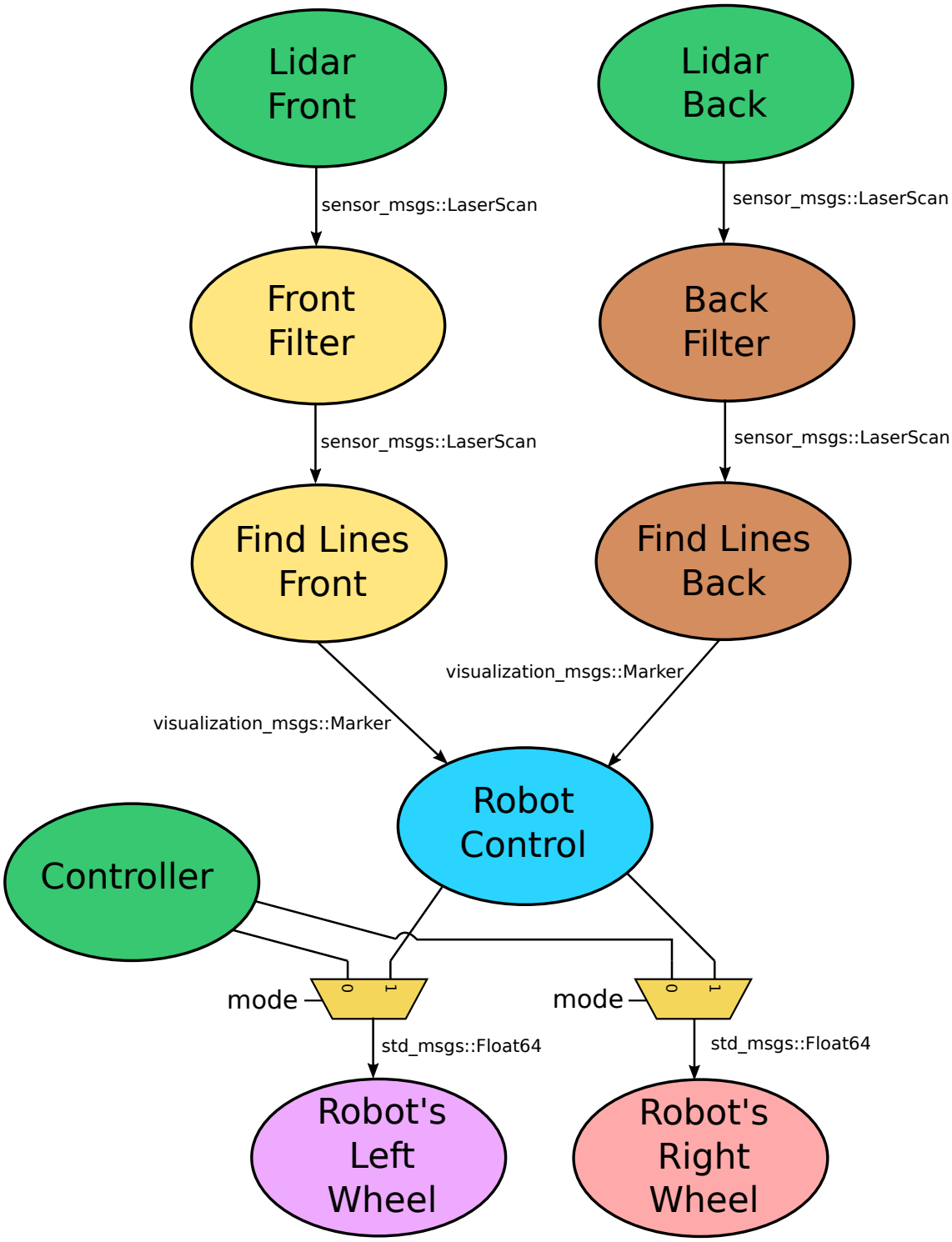

Figure 6.3: Real Robot Architecture

Lidar Filter

The one thing not explained yet is how the filtering works. Since our Lidar have more than 180° coverage, it returned the robot's plank as points and this was making our algorithm less precise. Our solution was then to filter some points depending on some properties. The filter works with a box defined by one interval in axis x and one interval in axis y, a box with intervals $x \in [-\infty, \infty]$ and $y \in [-\infty, \infty]$ will obviously cover all XY plane. Alongside with this box, we implemented an annulus to filter points that are too close or too far, obviously $r \in [-\infty, \infty]$ will also cover all XY plane. After defining it, one can also define the filtering operation, being union, where if a point is contained by either one of the two areas it will not be filtered, and the other being intersection, where a point muset be contained by both areas to not be filtered. This filter can be depicted by figure 6.4 where all points contained in areas painted in light green will not be filtered.

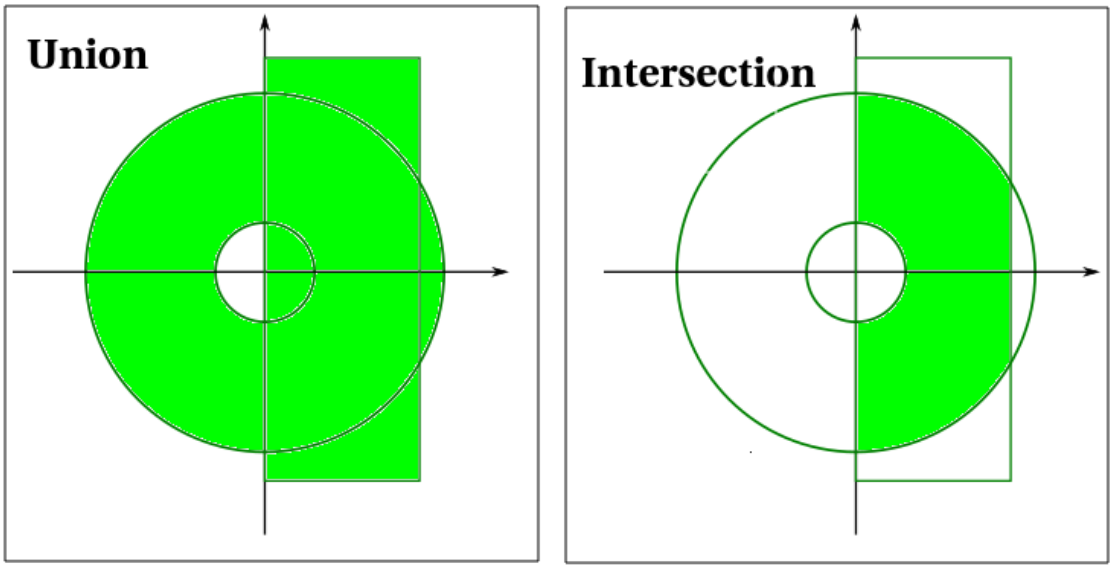

Figure 6.4: Filtering Methods

6.3 Wheels Command

To control the wheel, we must connect to the them using a serial communication. Since the raspberry Pi don't have a UART translator, we used an external one that can connect to two wheels, to visualize them, use the command $ls\ /dev/ttyUSB*$. Since the wheel's motor spin either clockwise or counter clockwise, it is important to note that the clockwise motion is seen from the back of the wheel, as shown in figure 6.5.

Figure 6.5: Clockwise Motion

All frames use a Cyclic Reduntant Check (CRC) that is defined by the sum of every Byte outside of the two first ones and a session ID that has to be incremented each successful communication. The CRC is described in equation 6.1, the session ID ranges from 0x10 and 0x1F, when the upper limit is reached it must be reset to 0x10, this ID will be replicated by the response frame (whether it is an error or a normal frame) to be used in debugging and synchronization.

$$CRC = (\sum_{n=2}^{size-2} frame\_byte_n) \mod 256 \tag{6.1}$$

Data Table			
Data	Data Addresses	Data Size	Type
Absolute Velocity	0x00 0x00 0x04 0x08	2	Read
Battery Level	0x01 0x00 0x06 0x06	1	Read
Set Velocity	0x00 0x00 0x04 0x07	4	Write
Get Wheel Structure	0x00 0x00 0x04 0x06	4	Read

6.3.1 Reading Data

Read Data Request			
	Byte	Description	Value (hex)
	0	Synchronization	0xAA
	1	Frame Size	0x0F
	2	Quick Session Frame Type	0x08
	3	Session ID	0x1X
	4	Frame ID	0x01
	5	Reserved	0x0A
	6	Reserved	0x00
	7	Reserved	0x00
	8	Reserved	0x00
	9	Request Type	0x01
	10	Reserved	0x05
	11-14	Data Address	See Data Table
	15	Data Size	See Data Table
	16	CRC	See CRC

Correct Response Frame			
	Byte	Description	Value (hex)
	0	Synchronization	0xAA
	1	Frame Size	0x07 + Data Size
	2	Run Session Frame Type	0x01
	3	Session ID	0x1X
	4	Frame ID	0x00
	5	Reserved	0x02 + Data Size
	6	Get Request Type	0x41
	7	Data Size	See Data Table
	8...	Data	See Data Decoding
	8 + Data size	CRC	See CRC

Error Response			
	Byte	Description	Value (hex)
	0	Synchronization	0xAA
	1	Frame Size	0x07
	2	Run Session Frame Type	0x01
	3	Session ID	0x1X
	4	Frame ID	0x00
	5	Reserved	0x02

	6	Error Request Type	0xFF
	7	Reserved	0x00
	8	CRC	See CRC

To decode data from the response frame:

Absolute Velocity Decode

This data structure contains 2 bytes (bytes 8 and 9) where the least significant bits are in byte 8. If one receives

Data Example			
	Byte	Description	Value (hex)
	8	Least Significant Bits	0x42
	9	Most Significant Bits	0x01

The useful data is then 0x0142 in hexadecimal, or 322 in decimal. Since the resolution is 0.1 Km/h, this means the wheel is rotating at 32.2 Km/h

Battery Level Decode

Since this data structure contains only one byte, it is very straight forward

Data Example			
	Byte	Description	Value (hex)
	8	Data	0x42

The useful data is then 0x42 in hexadecimal, or 66 in decimal. The resolution of this information is 1%, meaning the wheel has 66%.

Wheel Structure

The wheel structure gives a lot of information, but is quite complex to decode. This data structure contains the direction of rotation, if the wheel is in the speed mode, if the wheel has engine brake active, an estimation of the torque, the absolute speed, if an error has occurred, if the charger is connected, and the battery status. Imagine that one concatenates all 4 bytes received as data back to back in the form | Data 4 | Data 3 | Data 2 | Data 1 |, meaning the final data sample will have 32 bits long, the following table will describe what each of these bits mean, from the least significant to the most.

Measure	Size in bits	Meaning
Direction	2	0 = No motion 1 = Clockwise 2 = Counter-clockwise
Speed Mode	1	1 = Speed mode active
Reserved	1	0
Engine Brake	1	0 = Not active 1 = Engine brake active
Not Used	1	0
Toque approximation	10	Without unit
Speed	10	Speed in 0.1 km/h
Error	1	0 = Error
Charger Status	2	0 = Charger not plugged 1 = Charger Detected 2 = Charging
Battery Status	2	0 = Battery OK 1 = 50% 2 = 30% 3 = 0%
Not used	1	1

The following table will gave an example of response, and then how to decode it. Remember that LSB means Least Significant Bits, and MSB Most Significant Bits.

Data Example			
	Byte	Description	Value
	8	Data 1	0xC5 | 0b1100 0101
	9	Data 2	0x08 | 0b0000 1000
	10	Data 3	0x0A | 0b0000 1010
	11	Data 4	0x84 | 0b1000 0100

And for decoding

Byte 8 Decode (Binary)					
Torque(LSB)	Not Used	Engine Break	Reserved	Speed Mode	Direction
11	0	0	0	1	01

Byte 9 Decode (Binary)
Torque(MSB)
0000 1000

Byte 10 Decode (Binary)
Velocity(LSB)
0000 1010

Byte 11 Decode (Binary)				
Not Used	Battery Level	Charger Status	Error	Speed (MSB)
1	00	00	1	00

Finally, we have the $direction = 0b01 = 1 \rightarrow$ clockwise, $Speed\,mode = 1 \rightarrow$ Active, $Engine\,Brake = 0 \rightarrow$ Engine brake active, $torque = 0b0000100011 = 35$, $Speed = 0000001010 = 10 \rightarrow 1$ km/h, $Error = 1 \rightarrow$ OK, $Charger\,Status = 0 \rightarrow$ Charger not detected, and $Battery\,Level = 0 \rightarrow$ Battery full. **WARNING**: During tests, we noticed the 4th byte does not return the correct data, so please ignore it.

6.3.2 Writing Data

Write Data Request			
	Byte	Description	Value (hex)
	0	Synchronization	0xAA
	1	Frame Size	0x0F + Data Size
	2	Quick Session Frame Type	0x08
	3	Session ID	0x1X
	4	Frame ID	0x00
	5	Reserved	0x0A + Data Size
	6	Reserved	0x00
	7	Reserved	0x00
	8	Reserved	0x00
	9	Request Type	0x02
	10	Reserved	0x05
	11-14	Data Address	See Data Table
	15	Data Size	See Data Table
	16...	Data	See Data
	16 + Data Size	CRC	See CRC

Expected Response			
	Byte	Description	Value (hex)
	0	Synchronization	0xAA
	1	Frame Size	0x07

	2	Run Session Frame Type	0x01
	3	Session ID	0x1X
	4	Frame ID	0x00
	5	Reserved	0x02
	6	Get Request Type	0x42
	7	Reserved	0x00
	8	CRC	See CRC

Error Response			
	Byte	Description	Value (hex)
	0	Synchronization	0xAA
	1	Frame Size	0x07
	2	Run Session Frame Type	0x01
	3	Session ID	0x1X
	4	Frame ID	0x00
	5	Reserved	0x02
	6	Error Request Type	0xFF
	7	Reserved	0x00
	8	CRC	See CRC

Velocity Data Structure

To send a velocity command, the user must send 4 bytes of data, the first byte defines the direction of rotation, where 0x15 means clockwise and 0x16 means counter-clockwise. The second byte is not used, and the final two bytes defines the target velocity, where byte 18 have the least significant bits, and byte 19 have the most significant bits. Since only 10 bits can be used in velocity target, only the first 2 bits of byte 19 can be used. The velocity is passed with a resolution of 0.1 km/h, thus a 10 km/h target velocity requires a message data equals to 100.

Velocity Data Example			
	Byte	Description	Value (hex)
	16	Rotation Direction	0x15
	17	Torque Mode (Not Used)	0x00
	18	Vel. Least Significant Bits	0x64
	19	Vel. Most Significant Bits	0x01

With this example, the wheel is going to spin clockwise at 0x164 in hex, or 356 in decimal, meaning 35.6 km/h

6.4 Launching Nodes

This section will explain what each launch file does for each package

6.4.1 engrais_control

- findlines.launch : launches a findlines node, the sick tim node, and the Lidar filter node locally that receives a cloud of points and use one of the algorithms explained in section **??** to calculate the best lines.
- control.launch : launches the control node locally that receives the best lines found by findlines and calculates the wheels' velocities.

6.4.2 engrais_motors

- engrais_motor.launch : launches a node locally to communicate with the wheels using serial communication.

6.4.3 engrais

- Single Launch
  - central.launch : Launches the navigation node remotely, in the ROS_MASTER machine.
  - rasp1.launch : Launches the engrais_motor node remotely in Rasp 1 for the right wheel
  - rasp2.launch : Launches the engrais_motor node remotely in Rasp 2 for the left wheel
  - rasp3.launch : Launches the sick tim, filter, and findlines nodes in Rasp3 to receive back lidar's points
  - rasp4.launch : Launches the sick tim, filter, and findlines nodes in Rasp4 to receive front lidar's points
- Fragmented Launch
  - control.launch : Launches central.launch, rasp3.launch, and rasp4.launch
  - findlines.launch : Launches rasp3.launch, and rasp4.launch
  - motors.launch : Launches rasp1.launch, and rasp2.launch
- controller_move.launch : Launches rasp1.launch, rasp2.launch, and a node in ROS master to receive instructions from the PS4 controller.

- keyboard_move.launch : Launches rasp1.launch, rasp2.launch, and a node in ROS master to receive instructions from the keyboard.
- system_controller.launch : Launches rasp1.launch, rasp2.launch, rasp3.launch, rasp4.launch, central.launch, and a node in ROS master to receive instructions from the PS4 controller.
- system_keyboard.launch : Launches rasp1.launch, rasp2.launch, rasp3.launch, rasp4.launch, central.launch, and a node in ROS master to receive instructions from the keyboard.

Appendix

6.5 Engrais

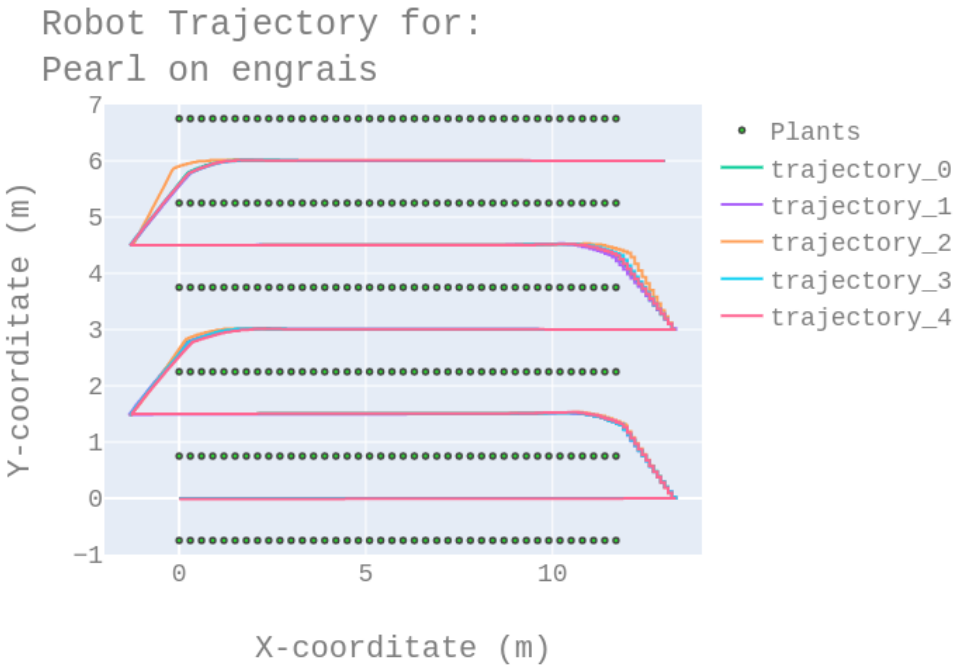

Figure 6.6: Pearl Algorithm on Engrais

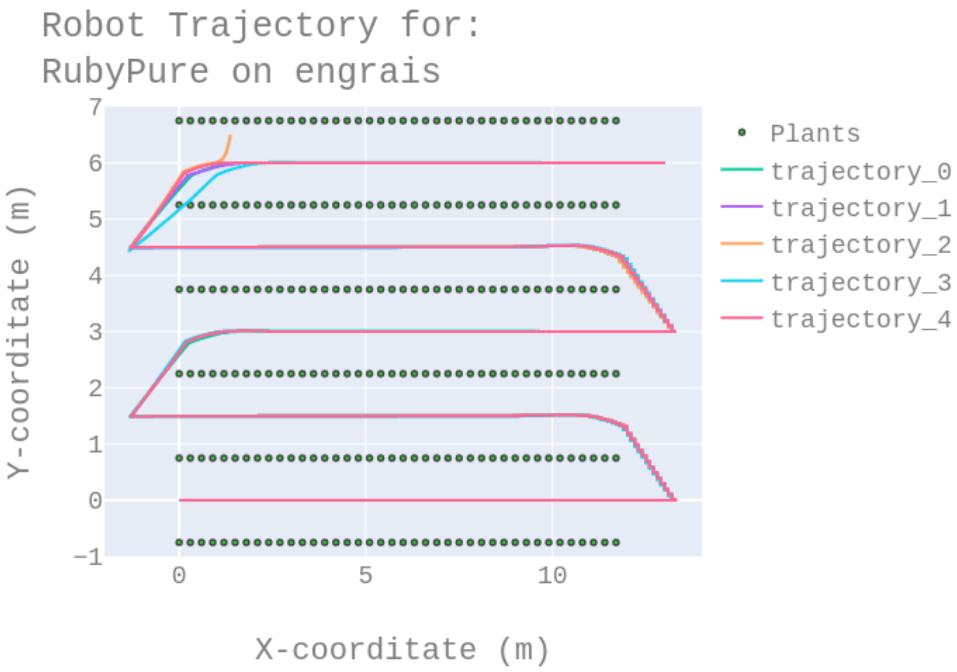

Figure 6.7: Ruby Algorithm on Engrais

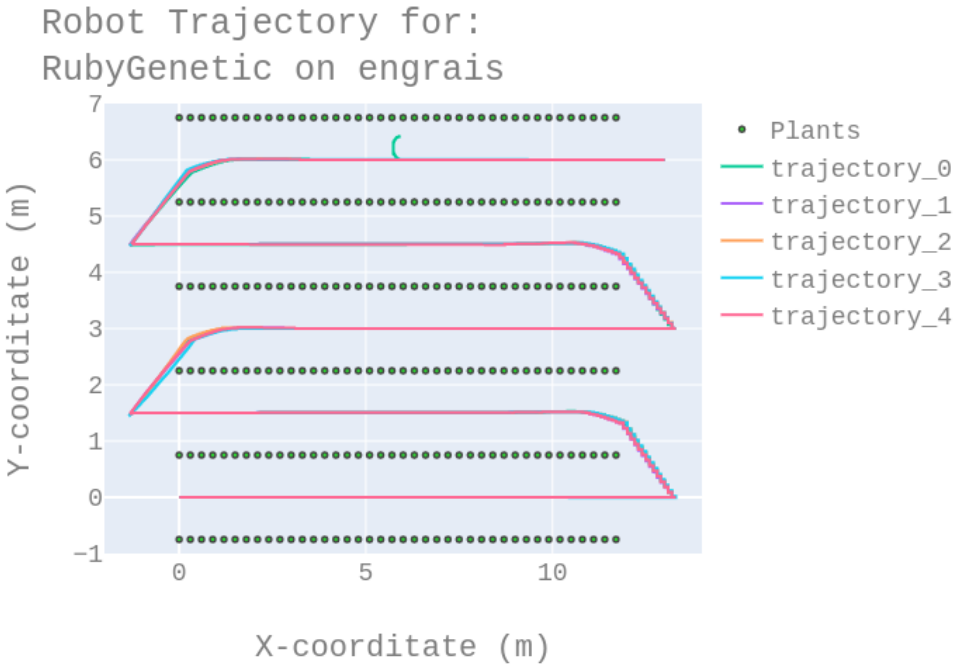

Figure 6.8: Ruby Genetic Algorithm on Engrais

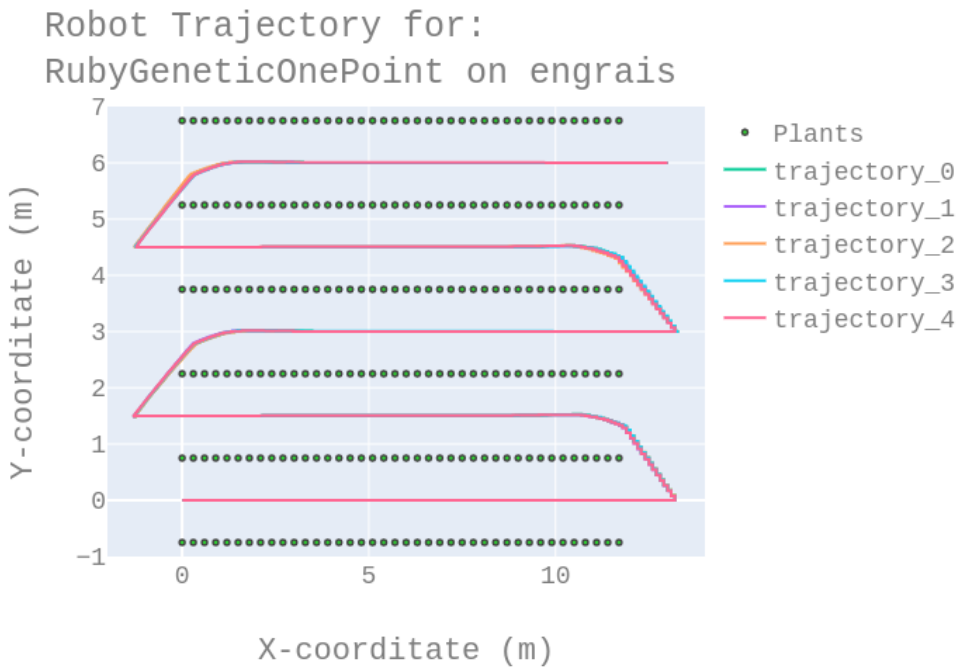

Figure 6.9: Ruby Genetic One Point Algorithm on Engrais

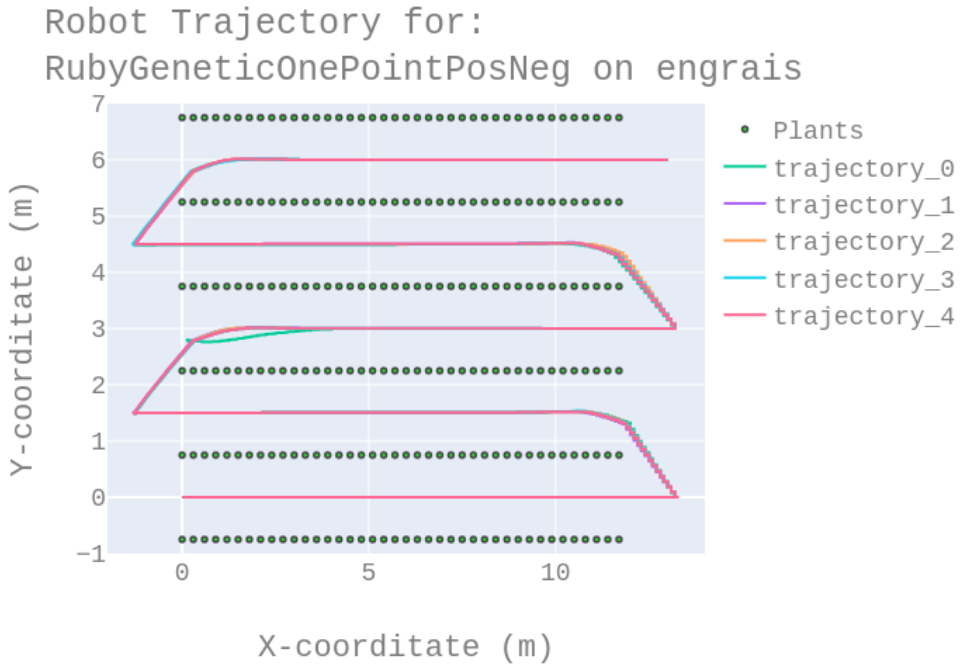

Figure 6.10: Ruby Genetic One Point Positive / Negative Algorithm on Engrais

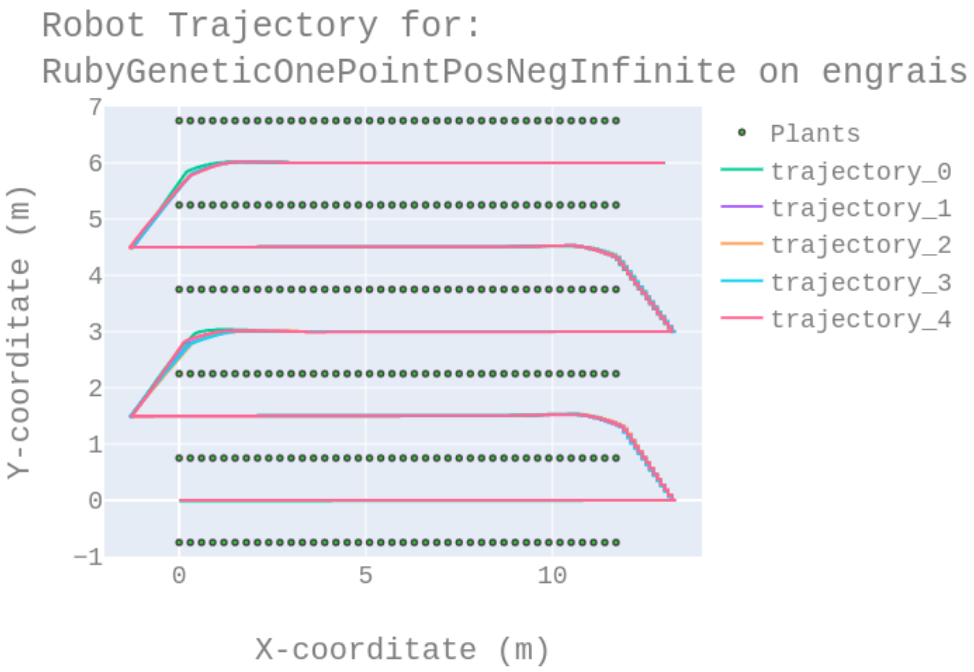

Figure 6.11: Ruby Genetic One Point Positive / Negative Infinite Algorithm on Engrais

6.6 Engrais2

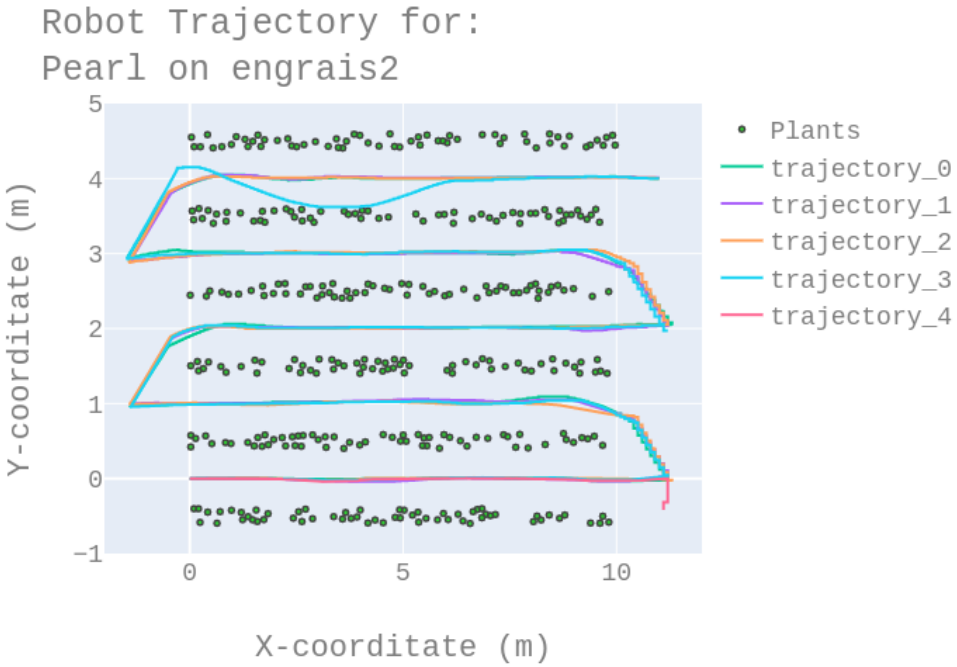

Figure 6.12: Pearl Algorithm on Engrais2

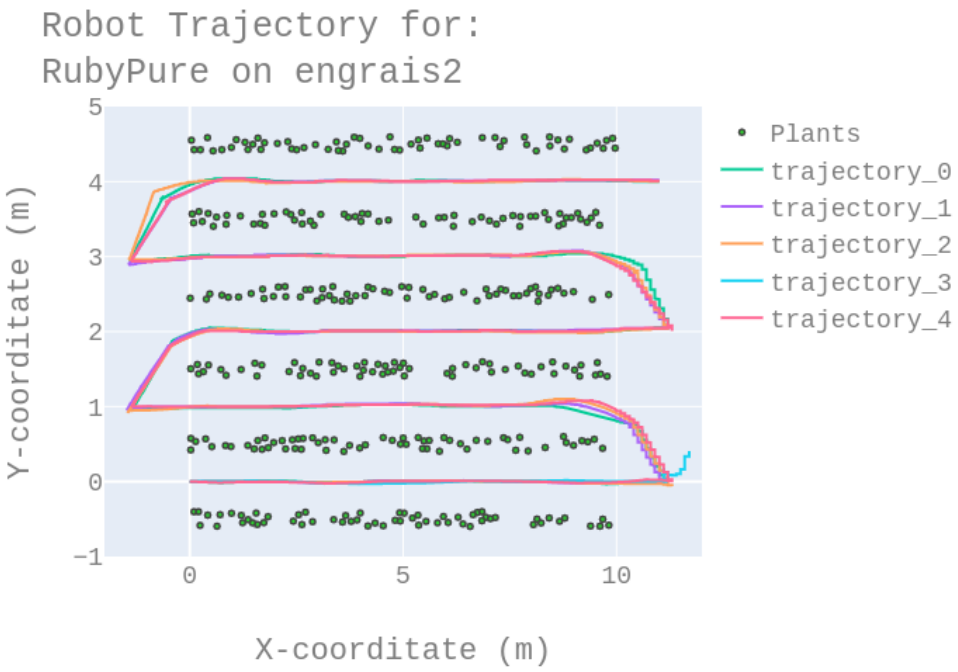

Figure 6.13: Ruby Algorithm on Engrais2

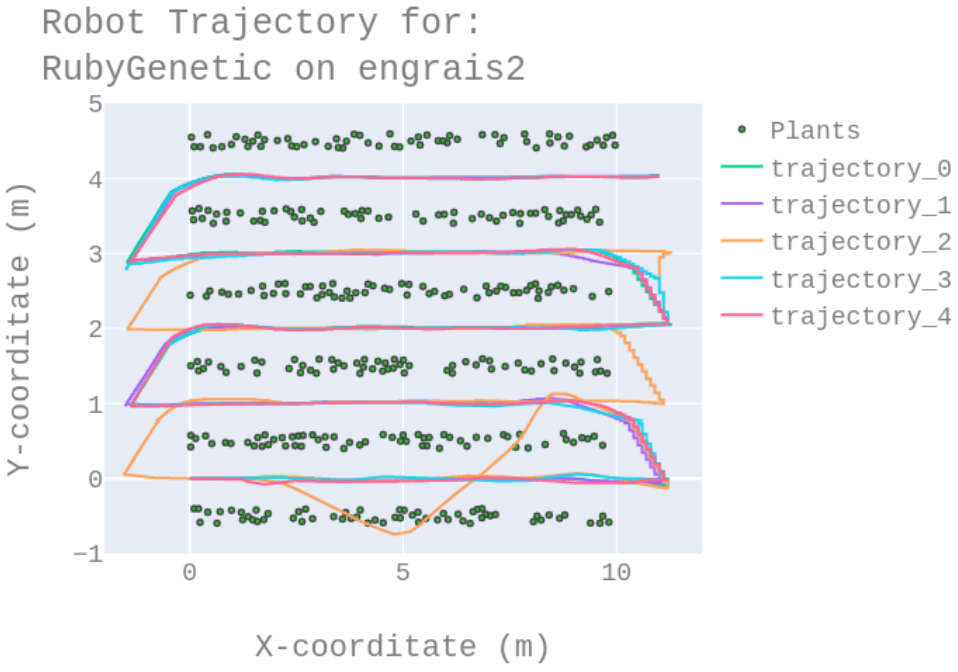

Figure 6.14: Ruby Genetic Algorithm on Engrais2

Figure 6.15: Ruby Genetic One Point Algorithm on Engrais2

Figure 6.16: Ruby Genetic One Point Positive / Negative Algorithm on Engrais2

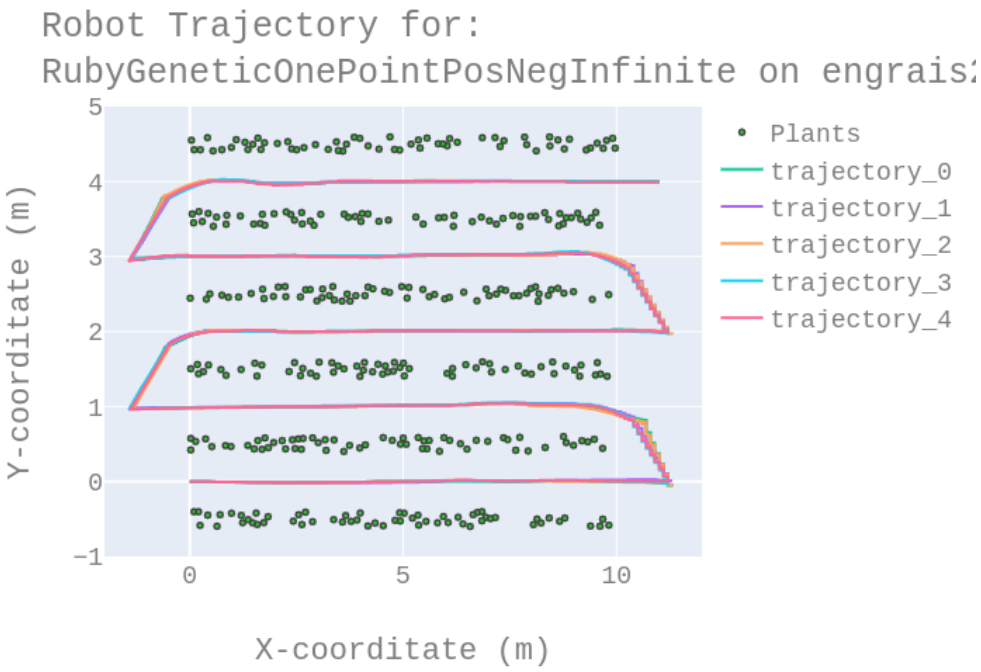

Figure 6.17: Ruby Genetic One Point Positive / Negative Infinite Algorithm on Engrais2

6.7 Engrais3

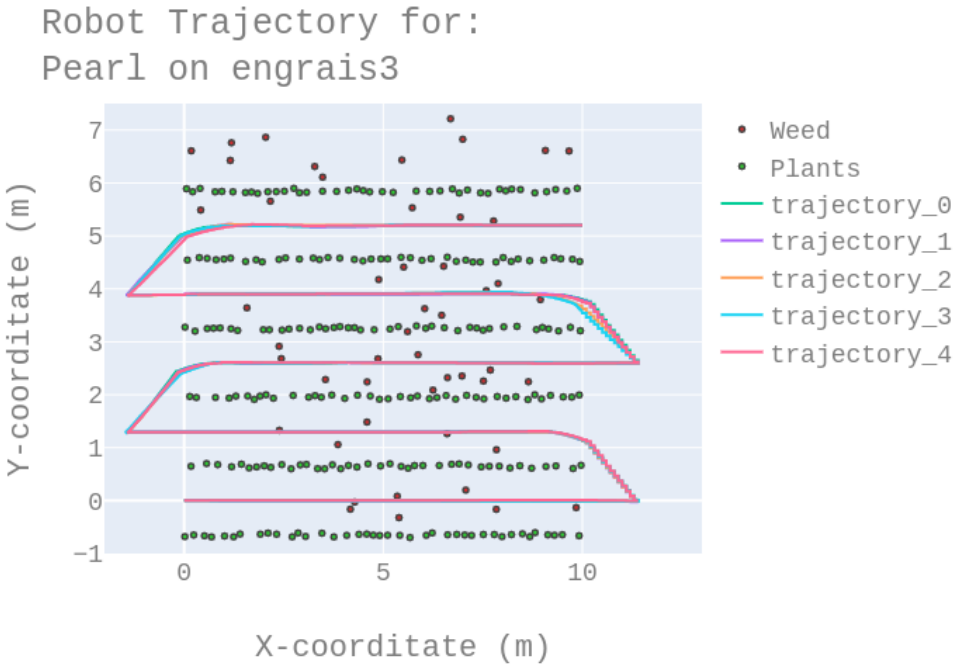

Figure 6.18: Pearl Algorithm on Engrais3

Figure 6.19: Ruby Algorithm on Engrais3

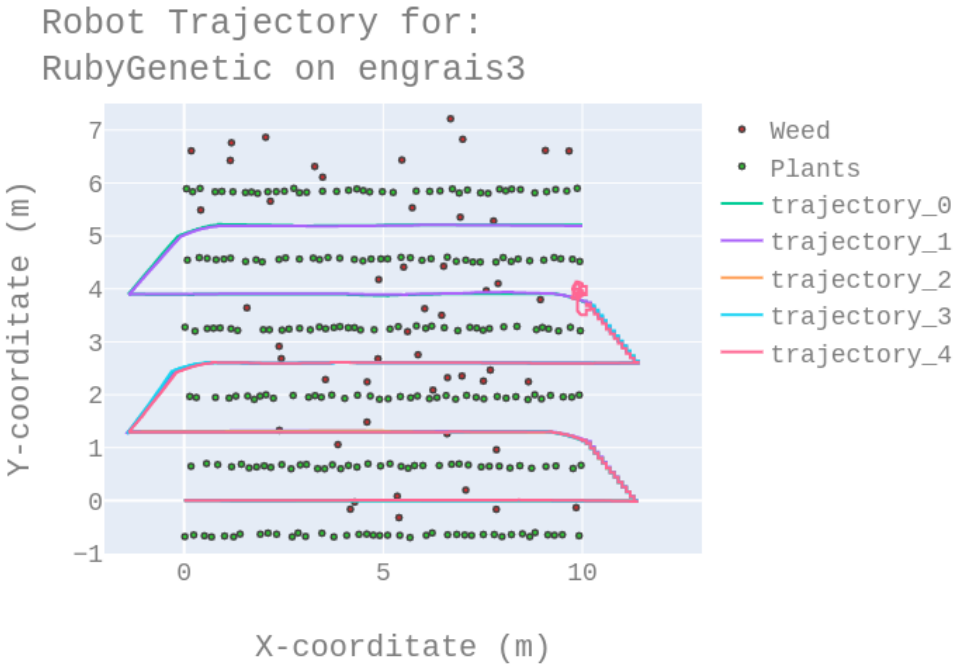

Figure 6.20: Ruby Genetic Algorithm on Engrais3

Figure 6.21: Ruby Genetic One Point Algorithm on Engrais3

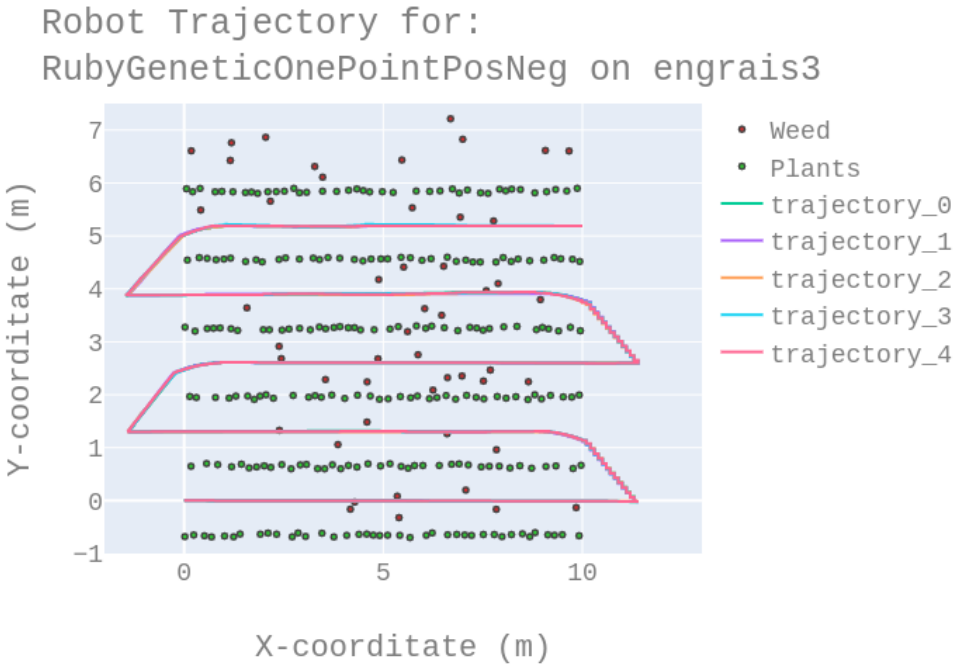

Figure 6.22: Ruby Genetic One Point Positive / Negative Algorithm on Engrais3

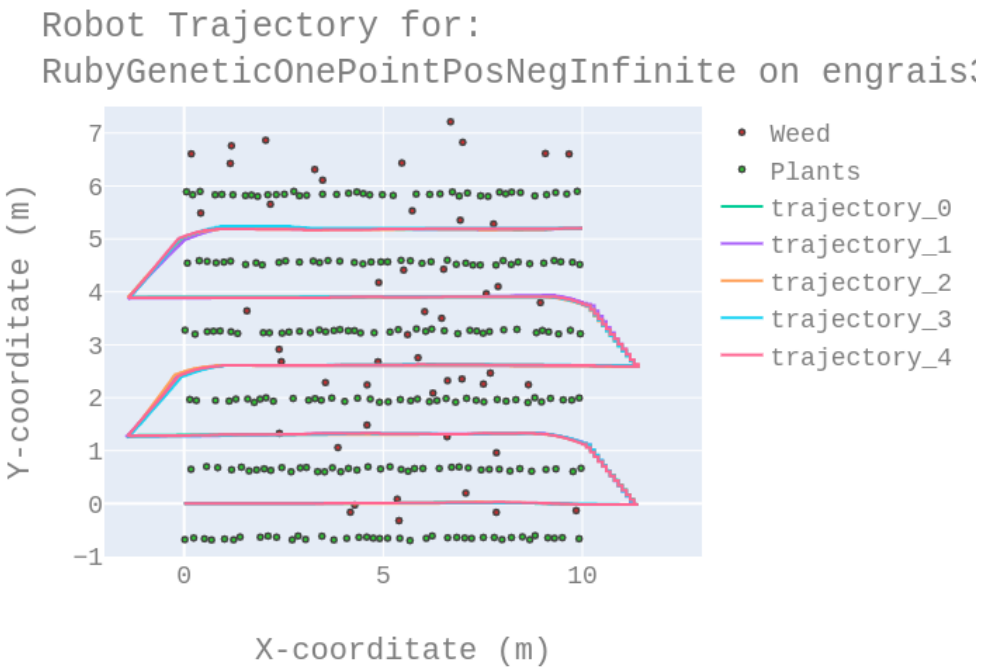

Figure 6.23: Ruby Genetic One Point Positive / Negative Infinite Algorithm on Engrais3

6.8 Engrais4

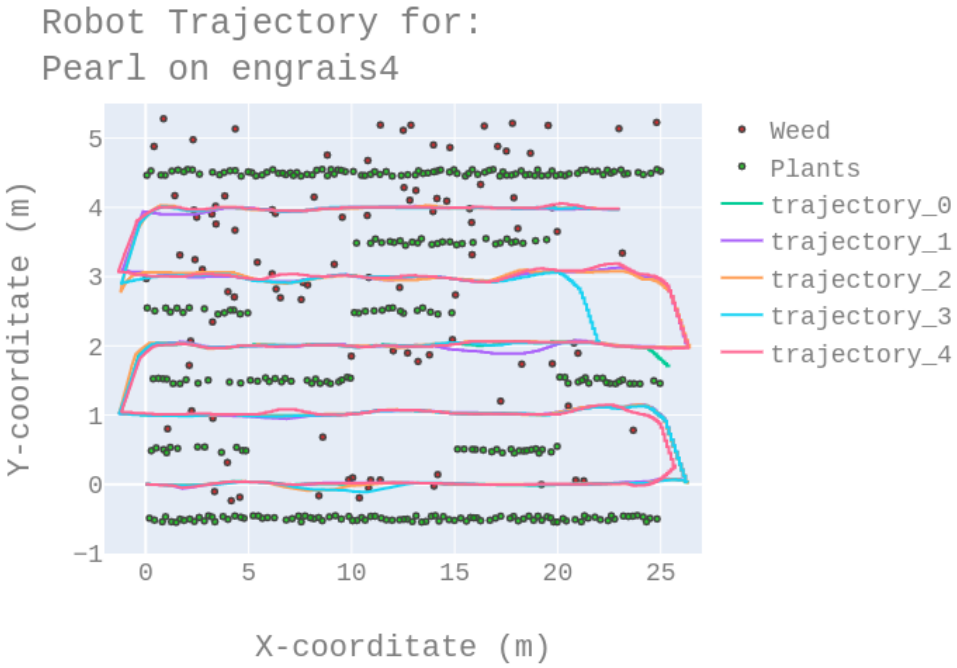

Figure 6.24: Pearl Algorithm on Engrais4

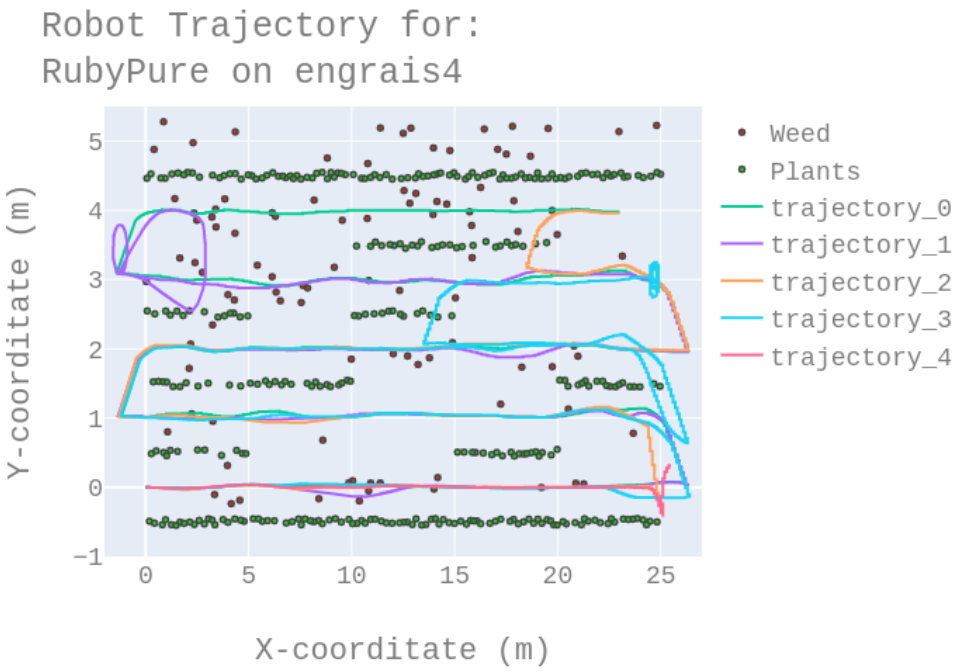

Figure 6.25: Ruby Algorithm on Engrais4

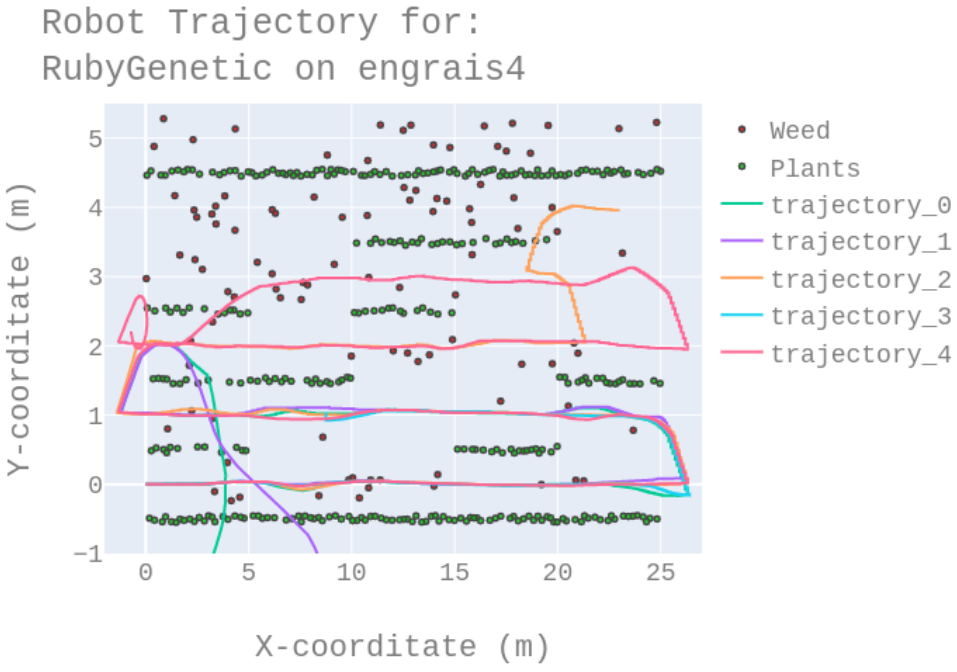

Figure 6.26: Ruby Genetic Algorithm on Engrais4

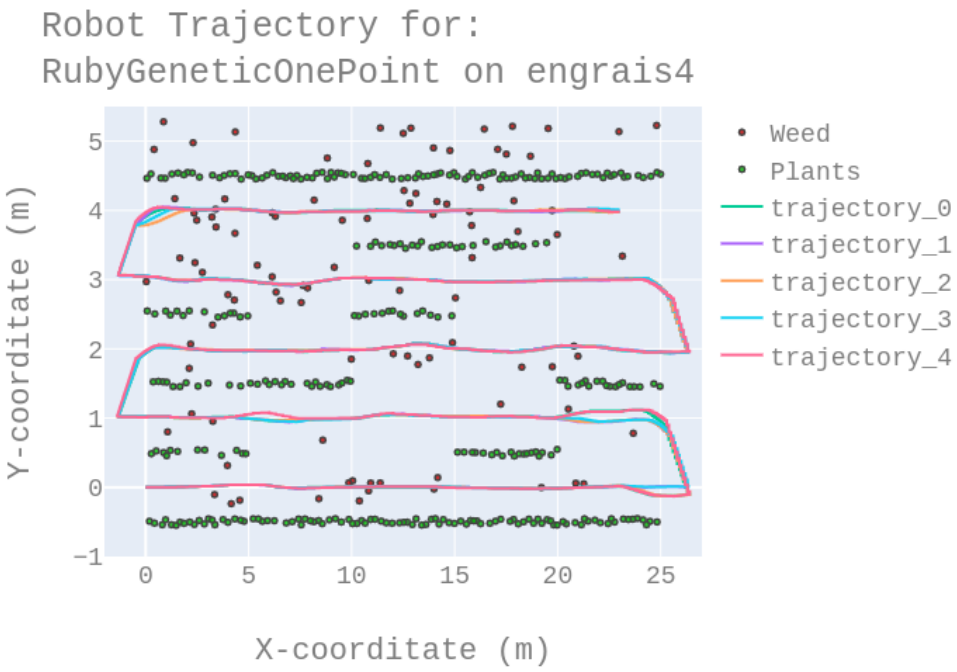

Figure 6.27: Ruby Genetic One Point Algorithm on Engrais4

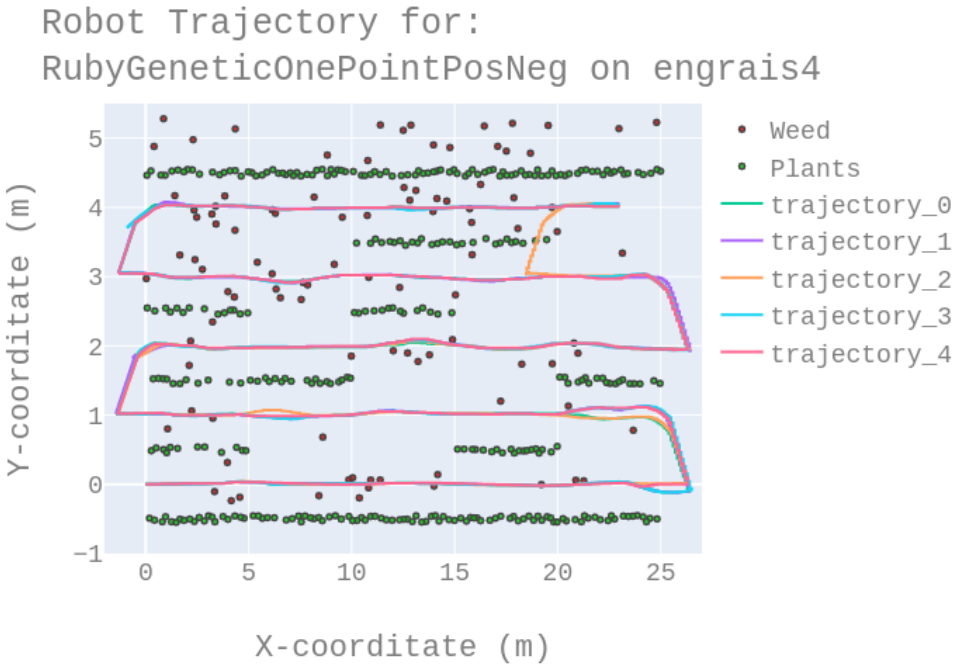

Figure 6.28: Ruby Genetic One Point Positive / Negative Algorithm on Engrais4

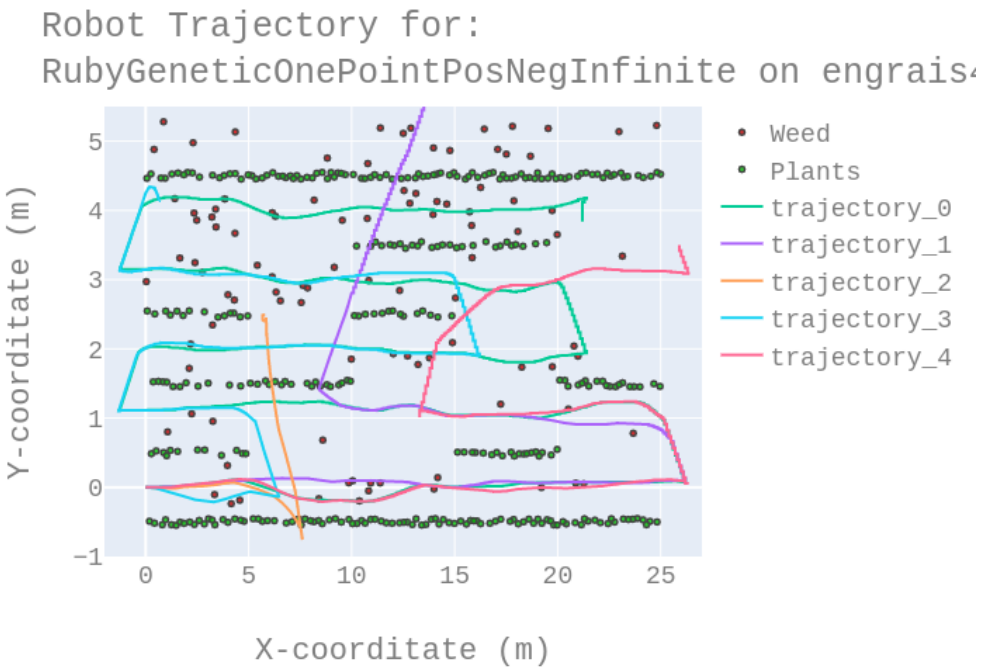

Figure 6.29: Ruby Genetic One Point Positive / Negative Infinite Algorithm on Engrais4

Abstract — This article aims to present the fundamentals of a Timed Event Graph (TEG - a petri net subclass) and apply it on industrial lines. Using a TEG, this article proposes an observer's feedback controller, based on classic theory, and a discussion of its benefits. Later, this calculus are exemplified using a simple system and the reader is able to follow the calculations (made by hand) to better understand the algebra. Finally, this controller is applied (using a software made by the author) into a real system, that simulates an industrial line. It aims to maintain a reference output while delaying the input as much as possible, in order to reduce overload in machines and lines, as well as other advantages.

Key words : Timed Event Graphs, Max-Plus, Control, State Observer, just-in-time.

ISTIA
62, avenue Notre Dame du Lac
49000 Angers