UCLouvain

École polytechnique de Louvain

Tracking Objects with several cameras using Deep Learning on Raspberry Pi

Authors: **Antoine WINANT, Alexandros XENAKIS**
Supervisors: **Benoît MACQ, Kaori HAGIHARA, Dani MANJAH**
Reader: **Raphael JUNGERS**
Academic year 2019–2020
Master [120] in Electro-mechanical Engineering
Master [120] in Computer Science and Engineering

Abstract

This thesis is interested in multi-agent distributed tracking, robust to agent failures and target occlusions. First, the architecture of a network of communicating agents is conceptualized. It is composed of two types of agents: agent-cameras and an agent-user. The former constitute the intelligent sensors, which gather and filter data on the environment, while the latter assemble the filtered data for the end-user. The presence of multiple agents allows to compensate for occlusions by offering a variety of views on the targets. Then, as the observations collected by the agents are noisy, the problem of noise reduction is examined, with an implementation of a distributed Kalman filter. Experiments show a slightly improved noise reduction compared to the local Kalman filtering. Finally, an extension is considered, where the agents are allowed to move, in an attempt to improve their view of the targets they track. Depending on the use-case, this can improve the quality of the tracking.

Acknowledgments

First and foremost, we would like to express our deepest gratitude towards our supervisors, the Pr. Benoît Macq, Kaori Hagihara and Dani Manjah, who provided us guidance and feedback throughout the year. This thesis could not have been completed without their insightful advice and continuous involvement, always filled with enthusiasm.

We also wish to express our sincere appreciation to Antoine Aspeel, who provided invaluable advice and feedback.

Finally, we are extremely grateful to our family and friends, who supported us from the start and accompanied us in times of social distancing. A big thanks to each of you.

Contents

Chapter 1

Introduction

The objective of this thesis is to research and implement a multi-agent distributed system that robustly tracks objects in the presence of obstructions. Object detection software has recently been developed on single board computers [1], which constitutes an opportunity to research a new low-cost scalable multi-agent system. Causes of obstructions include physical obstacles, illumination changes, or anything hindering the sensors from perceiving the desired part of the environment.

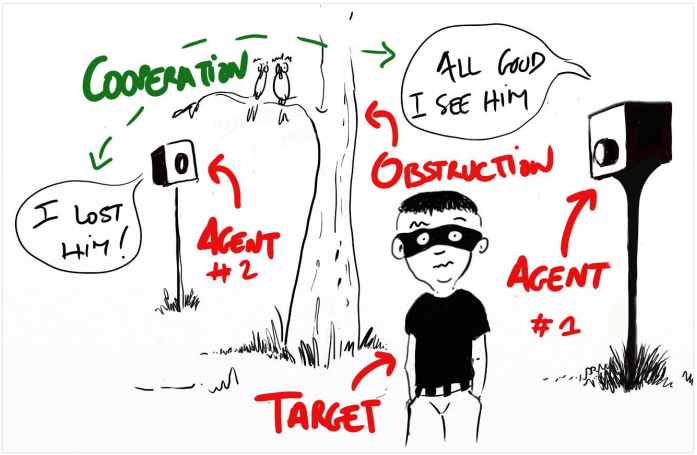


Figure 1.1: Main idea illustration

The work required for this thesis was divided into two main tasks. The first task consisted in research focused on conceptualising a multi-agent system for robust tracking in the presence of obstructions. The second task involved implementing a simulation in order to test and evaluate ideas.

1.1 Block decomposition of objective

This section describes the steps involved in the creation of a fully functional system, with no consideration for implementation detail. In the following paragraphs, (x) refers to a step depicted by a cloud numbered x in figure 1.2.

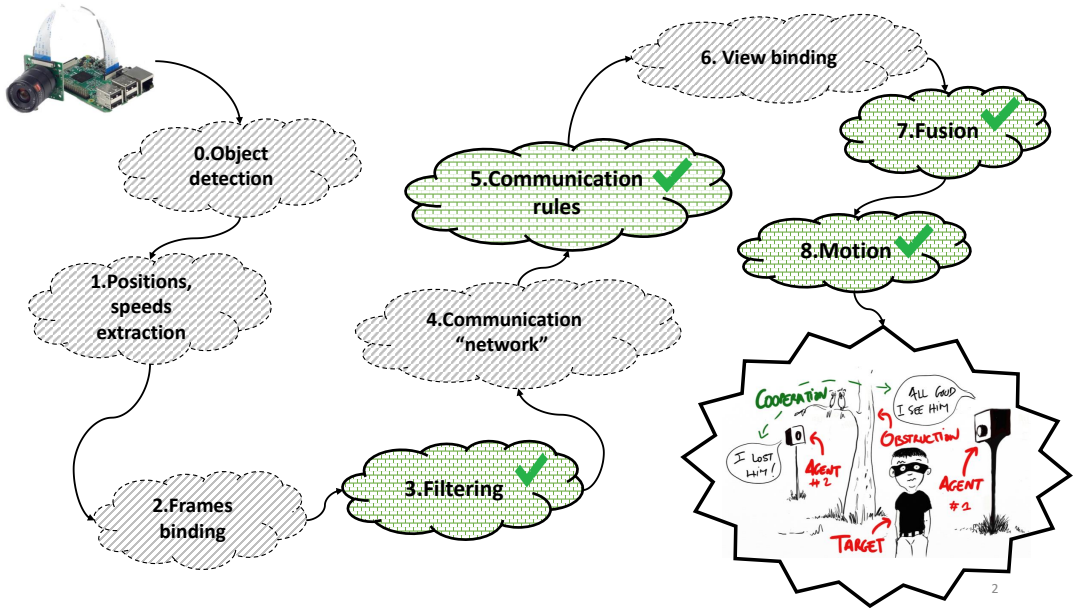


Figure 1.2: Steps before reaching the final goal

The first task (0) is to implement an object detection algorithm on a single board computer. This is possible in several ways, some of which are considered in appendix A. For many existing algorithms, pre-trained models exist, covering a wide range of objects. However, a specific use-case might require additional training.

The next sub-problem (1) is that of estimating the positions of the targets detected by the object detection algorithm. Object detection algorithms output a set of positions in the two-dimensional image plane. Therefore, this output has to be converted to a three-dimensional space. Furthermore, the bounding boxes (which define a rectangular region around the detected object) vary from frame to frame, even for still images. This is a source of noise, resulting in the need to ensure that the signal-to-noise ratio is sufficiently high for the data to be used.

The next problem to be resolved is that of frame binding (2). While tracking a single target within the frame stream is trivial, tracking several targets is both crucial and of increasing difficulty. The difficulty lies in linking targets that are common on several frames within the stream, i.e. connecting the bounding boxes of each object with its corresponding bounding boxes in previous frames, if applicable.

Once frame binding is completed, each single board computer - referred to as an agent - will have received a sequence of positions for each target in the field of view of its camera. Since cameras are imperfect physical detectors, these measurements are considered noisy, and will therefore need to be filtered (3).

After having the sequences of filtered positions, the agents should use this information to make up for obstructions and try to continuously track a maximum number of targets. To achieve this, the agents will likely need to communicate in some way (4, 5). A new identification problem appears at this stage and needs to be solved before going further (6). Humans can easily identify a person that appears on two different cameras with different angles. For a computer however, it is not straightforward and a way to recognize that the information exchanged is referring to the same target should be found. Finally, estimations of a target's position sent by different agents should be fused (7).

Optionally, one additional step can be added if the agents have at least one degree of freedom (8). Indeed, they can use the information they have gathered along with the exchanged information in order to move, in an attempt to improve measurements.

Only a subset of the steps are addressed (green clouds with plain outline in figure 1.2): chapter 2 describes the communication rules (5). The noise removal (3, 7) is covered by chapter 3 and the motion (8) by chapter 4. For these four steps, a simulation was developed to evaluate solutions, the other parts being modeled as black boxes. The execution flow is thus slightly simplified. Each agent receives noisy positions. It then filters it in a distributed way and exchanges relevant information with the other agents. Optionally, the agents finally try to position themselves in order to keep a clear view of the targets for as long as possible.

1.2 Simple use-case

This section provides an overview of how the software created as part of this thesis runs a simulation of a simple use case. The basic features of the software will be highlighted, illustrating how a real situation can be simulated. The main objective of the software is to create a highly scalable and autonomous system which minimises obstructions. Links to the generated data and to illustrated plots, as well as a video demonstrating the execution of the simulation of the simple use case are available in appendix H.

Scalability and modularity are the key ideas, therefore the simulation includes an *"environment creation tab"*, in which it is possible to model a specific use-case (more details in the section 5). The *"creation tab"* is presented in the figure 1.3, where one can observe two cameras facing each other (blue dot on the left and the yellow dot on the right) with their field of vision represented as a red triangle. Also, there are two fix targets (blue outer circle) and a moving target on the top along with its trajectory. The goal is to track this target, travelling from the top all the way down to the bottom.

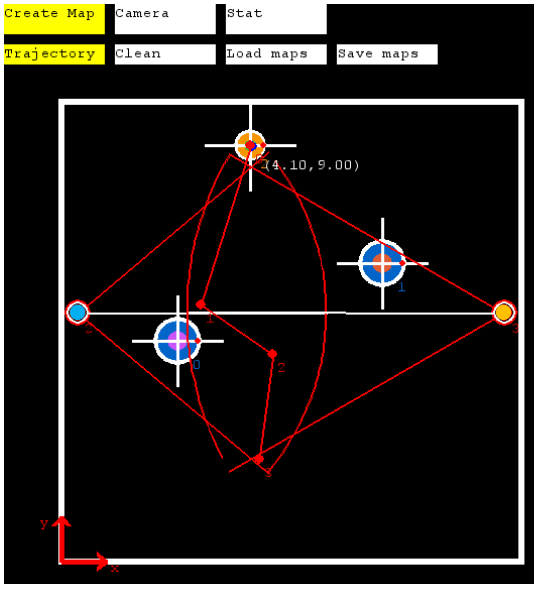


Figure 1.3: Map creation in the simulation (*"creation tab"*)
link to video available in appendix H

The situation in the figure illustrates a simplified reality. Imagine, at first, a target moving in a room, a park or anywhere else. There are generally many obstacles such as trees, pillars or other objects which could possibly hide the target. Therefore the target might sometimes disappear. From the camera's perspective, it will react to such a case either by moving or by relying on another camera with a clear view on the target.

Figure 1.4 shows the output after running the simulation on the map presented in the *"creation tab"*. The cameras are placed such that they have a different angle of view on the room, therefore they cooperate in a complementary way. Pillars or trees (or any fix obstruction) are depicted with blue circles and obstruct the camera field of vision. The blue camera on the left detects only the upper region whereas the yellow one only detects the lower region. The blue dots are measurements acquired by the blue camera and the yellow ones from the other one. As a matter of fact, having both cameras cooperating enables a complete coverage of the surveillance zone. At time $11,30$ s, one can observe a blue line between the blue camera and the tracked target. This line graphically shows that the camera is in charge of tracking the target.

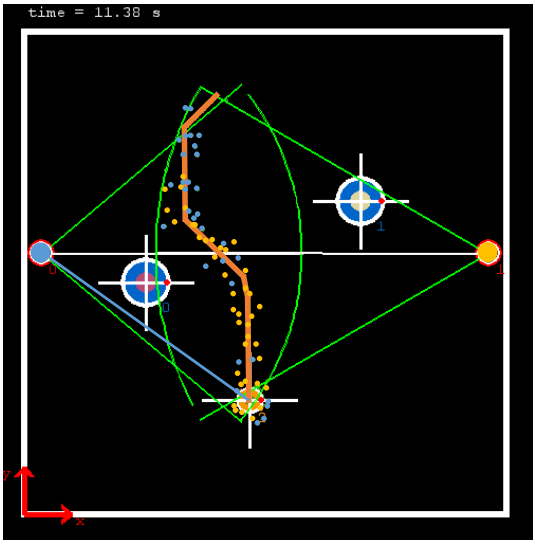


Figure 1.4: Simple case (*"simulation tab"*)

Figure 1.5 corresponds to the information received by the user. The user is an entity collecting information sent by the cameras. A comparison between generated and measured data is presented in the left part of the plot. Data is transmitted only for target 2, as the other targets are fix (such as pillar or trees). Furthermore, the color of the path refers to the time, which ranges from red to purple (about 10 s in total). Target 2 is thus moving from the top towards the bottom. The other targets are fix, as expect.

The second plot shows how agents cooperate. It is still the trajectory of target 2 and the color is, in this case, associated to the agent sending the information to the receiver. A purple color in figure 1.5 means that the camera 0 is sending the information, whereas a green color means that the camera 1 is sending the information. Therefore it becomes an evidence, when target 2 is obstructed by target 1 for the blue camera, the yellow camera starts sending information.

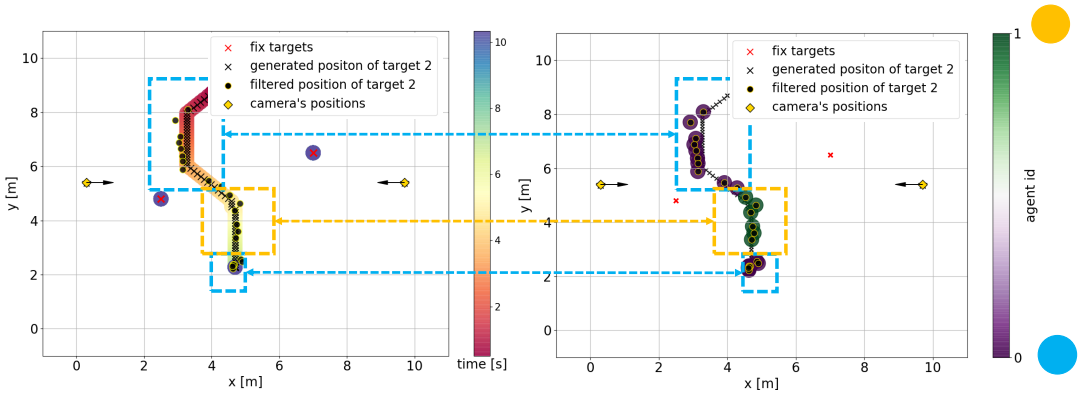


Figure 1.5: Information received by the user in terms of positions

Finally the figure 1.6 extracts the essential features from the above plots to summarize them. Three pieces of information are kept: the time the information was received, the sender and the target it is referring to (in this case there is only target 2). The red rectangle emphasizes a change of the camera that is in charge, allowing to continue tracking target number one. It is worth noting that from the user's side, the trajectory is uninterrupted and no actions were undertaken to coordinate the communications.

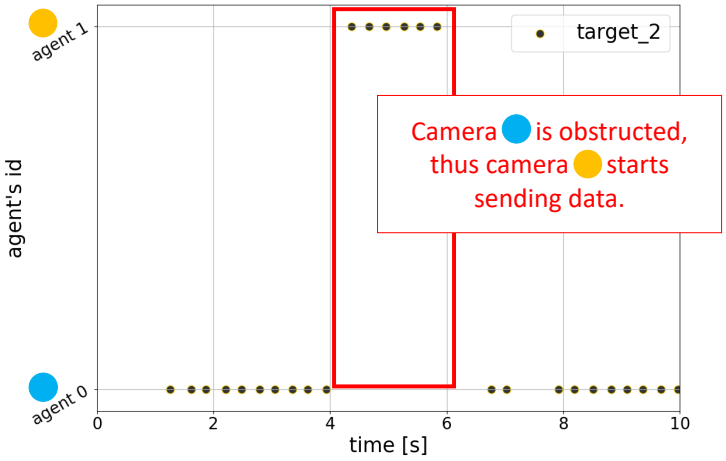


Figure 1.6: Information received by the user

Means and methods set up to achieve this cooperation between autonomous cameras are the object of this document. The second chapter describes the multi-agent architecture of the system. Then, the third chapter focuses on the two milestones, *"filtering"* and *"fusion"*, with a dedicated Kalman Filter implementation. The fourth chapter is about the motion of the agents. The motion is an additional feature of the system. It allows to modulate the coverage, however it brings complexity to the system, amongst others in the communication and different specific algorithms. The fifth chapter describes technical details about the simulation and its implementation in python. Principal results are presented and interpreted, in chapter 6. Finally future work is exposed in the chapter 7.

Chapter 2

Multi-agent System

There exist low-cost single board computers that are computationally powerful enough to be effectively coupled with an object detection algorithm [1]. The principal drawback is the limited computational power compared to a computer, which limits the algorithm's performances and requires more time to process an image (more information in appendix A). However this disadvantage can be compensated by the low price and the possibility to combine multiple smaller computers that cooperate.

Subsequently, single board computers such as the raspberry PI or the Jetson Nano, in association with object detection software, constitute an ideal primary element to conceptualize a distributed tracking system.

A mutli-agent approach presents the following advantages:

- The system is more robust: a failure in one part does not paralyze the whole system. For instance, one of the single board computer failing means that one sensor is lost, but does not prevent the others from cooperating in the way they were.
- The system is scalable: a new sensor can be easily added/removed to the existing network.

2.1 Definition

A **multi-agent system** is a system composed of a set of agents, located in a certain environment and interacting according to certain relationships.

In a multi-agent system, each **agent** presents the following features:

- **Autonomy :** Its actions are not dictated by an external entity; they are a result of the information gathered by the agent itself.
- **Sociability :** It communicates with other agents, sharing useful information.
- **Reactivity :** It acts according to its perceived environment.
- **Pro-activity :** It anticipates changes in the environment.

2.2 Overview

As depicted in figure 2.1, the system is composed of two types of agents: *"agent-cameras"* and *"agent-users"*.

The bottom layer in the figure corresponds to the physical cameras which collect the data (pictures of the environment). Each camera is a supplier to its associated *"agent-camera"*.

The *"agent-cameras"* (middle layer) act as filters and only transmit the relevant content to the upper layer, called *"agent-user"*.

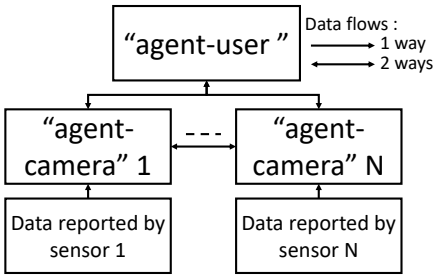


Figure 2.1: Global network overview

As opposed to the *"agent-cameras"*, the *"agent-user"* (third layer) plays a passive role. It neither takes measurement nor processes any data. It assembles the partial information provided by the *"agent-cameras"* in order to present a reconstructed trajectory of the tracked targets to the person using the software.

An important precision is that agents are equals from a hierarchical point of view and can communicate together at anytime. Their specific role is clarified in the following sections.

2.3 Agent-user

The *"agent-user"* combines its inputs (ie. outputs from *"agent-cameras"*) to present a summarized overview to the person using the software. In the implemented system, the *"agent-user"* plays a passive role, as it only acts when estimations are received and it cannot ask for them. *"Agent-cameras"* cluster around it and send estimates of the tracked targets' positions (this behaviour is explained it the section 2.6). The *"agent-user's"* output represents the overall system's output.

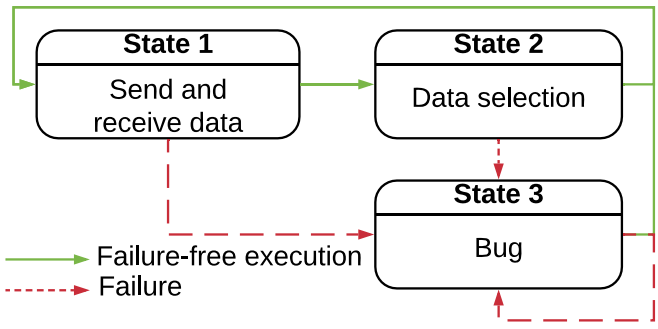


Figure 2.2: *"Agent-user"* FSM

The first task accomplished by the *"agent-user"* in a failure-free execution is the reception of data (State 1). This is presented in the section 2.5.

Situations could occur in which multiple *"agent-cameras"* send similar information at the same time (see section 2.7). For the person using the software, one of them is sufficient, therefore data selection (State 2) is a key aspect to keep the most updated estimate with the highest accuracy. A timestamp and an accuracy level are attributed to each of them as a criterion for the selection process.

Finally, state 3 models a failure, allowing to evaluate the robustness of the system.

2.4 Agent-camera

"Agent-cameras" are the principal actors in the system. Their role is to analyse data provided by their sensor (a camera) and to cooperate with the other *"agent-cameras"*. This cooperation mainly aims at reducing noise and avoiding sensor obstructions. Furthermore, the *"agent-cameras"* coordinate themselves to send the minimum necessary information to the *"agent-user"* they are clustered around. Thus, they actively participate to the information collection, processing and transmission. Finally, the *"agent-cameras"* include *"motion algorithms"*, attempting to dynamically optimize their view according to a set of targets.

One of the main considerations during the creation of the system was to limit the amount of exchanged messages. A decentralized architecture will always increase the number of messages, in comparison to a centralized one. Therefore, it is important to keep this drawback in mind, in order for the system to remain scalable.

The *"agent-cameras"* are autonomous, they are not coordinated by a central entity. A first task of the agents is to collect and filter measurements on their environment. A second task is to decide whether to send the information about the targets in its field of view to the *"agent-user"*. This is required because the *"agent-user"* should not receive the same information from more than a predefined number of agents, in order to reduce communications. Therefore, a criterion is defined, according to which the *"agent-cameras"* know if they should send the information to the *"agent-user"*. In the implemented system, the criterion is the distance to the target, assuming that the observation of the closest camera is likely to be the more accurate one. The information required by an agent to make this decision is the location of the other agents as well as the location of the targets, which are stored in the *"room-representation"* of the agent. This is a virtual representation of the surrounding environment based on information either collected by its sensors or by communicating with the others.

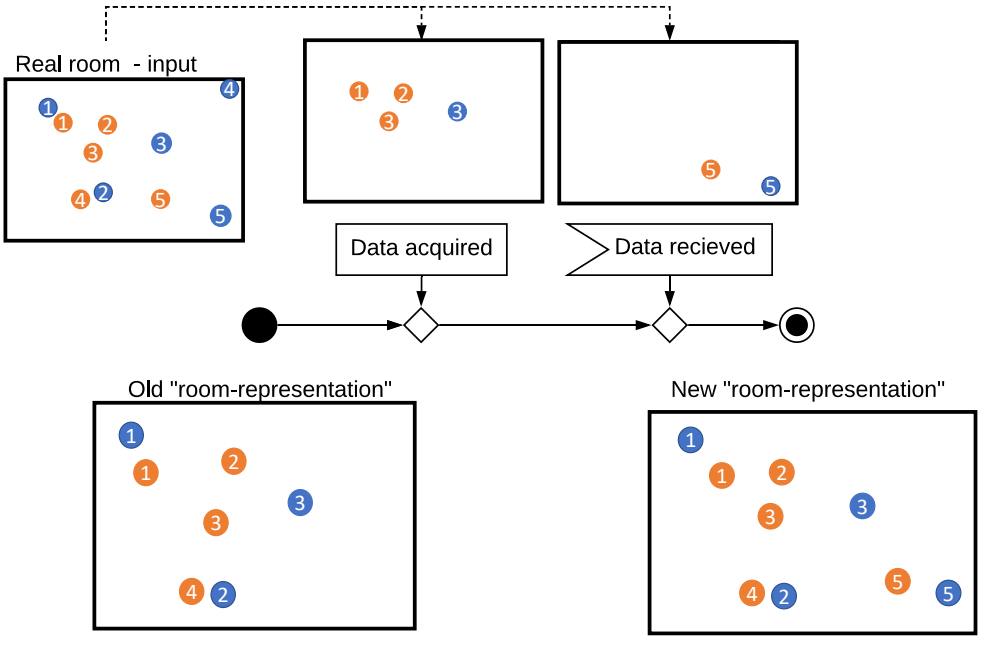


Figure 2.3: Description of belief

Figure 2.3 illustrates how the above principle works. The real environment is shown on the top left (Real room), where targets are represented in orange and cameras in blue. The perspective of camera 3 is arbitrarily chosen. Starting from its initial *"room-representation"* (Old *"room-representation"*), the goal is to describe how it is updated according to the new information gathered and received. The *"agent-camera"* receives information from its camera (Data acquired) as well as from other agents - in the figure, only from agent 5 - (Data received) and incorporates it, resulting in the new *"room-representation"* (New *"room-representation"*). In this example, several points are worth noting. First, agent 3 has not received information from the camera 4. This means that agent 4 has never communicated with agent 3. This could be due to the agent 4 being faulty. Then, in the new *"room-representation"*, agent 3 is aware of the agent 5, as he just received this information. Furthermore, as the agent 3 did not receive anything new from agents 1 and 2, those agents' positions were not updated in the new *"room-representation"*. Finally, the targets 1, 2 and 3 have updated positions in the new *"room-representation"*, as this information was collected.

As highlighted in figure 2.3, the *"room-representation"* contains information on the known targets as well as information on the other *"agent-cameras"*. This way, an *"agent-camera"* can put itself in the position of another *"agent-camera"* in order to predict the actions of the others. As all the algorithms use the *"room-representation"* as their starting point, it is the data-structure determining their actions. Ideally, the agents would always have the same *"room-representation"*, thus they would exactly predict each others actions without requiring to communicate.

In practice, the believes (ie. the *"room-representations"*) diverge. In order to keep them as synchronized as possible, the agents will therefore communicate. There can be problems with the followed approach (see sections 2.6 and 2.7 for more details).

2.4.1 Finite state machine

The *"agent-cameras"* are constantly running a loop that can be represented by the finite state machine of figure 2.4. The FSM contains five states:

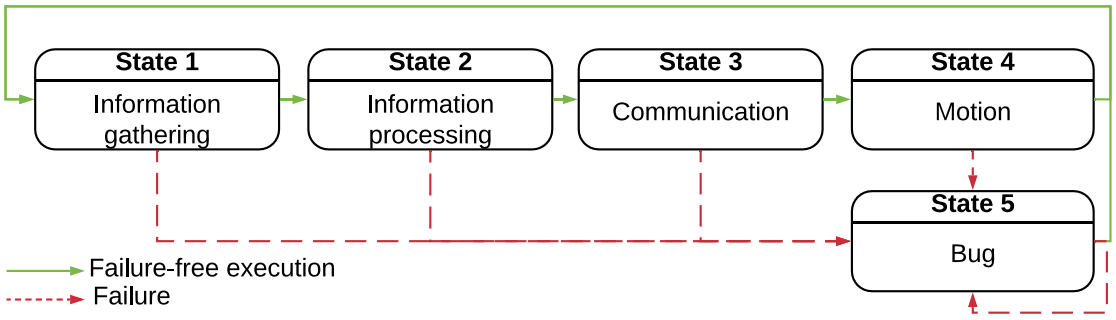


Figure 2.4: FSM-*"agent-camera"*

1. *"Data acquisition"* is the input state. Data is gathered and filtered using a Kalman filter (see section 3).

2. *"Process information in memory"* is a state in which various operations involving the data gathered in the previous state happen.

3. *"Communication"* is a state where the agent receives messages and responds appropriately (see section 2.5).

4. *"Motion"* is an optional state where the agent moves to a suitable position he calculated. (see chapter 4)

5. *"Bug"* is a state to simulate a failure with a given agent and to test how the others would react.

The following sections detail each of the states. Each state is illustrated by a small schematic composed by blocks. Each block has a number on it. When referring to a block numbered x, the notation: **(x)** is used.

2.4.1.1 State 1 - *"Information gathering"*

Each *"agent-camera"* starts by gathering information from its sensors. In the simulation, the exact positions are available **(2)**. Therefore, each agent artificially adds random Gaussian noise to each position **(3)**. However, this can easily be replaced by any other method of data acquisition. One could imagine, for example, analyzing a picture using an object detection algorithm, which outputs noisy positions, and using it as input in this phase of the agent's cycle **(1)**.

The positions for each seen target being now available to the agent, they are directly stored in memory and filtered using the noise removal strategy described in section 3 **(4)**. From here on, the agent will therefore only work with the filtered or predicted positions.

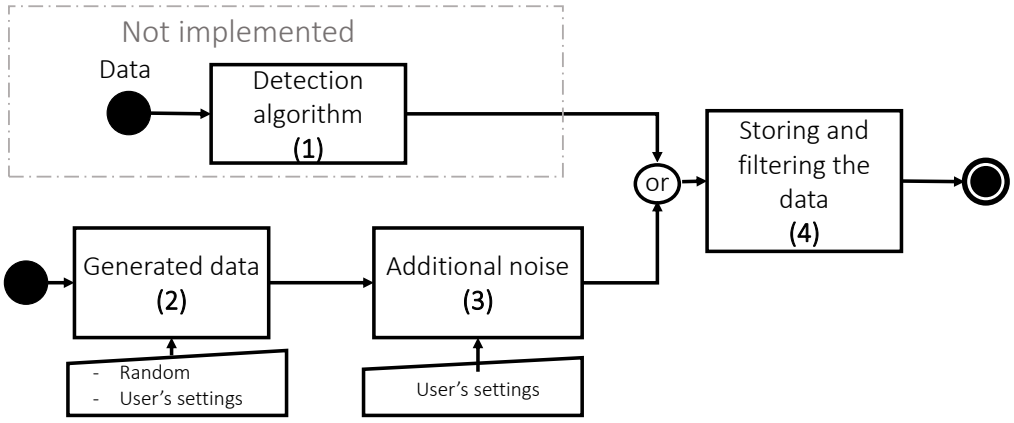


Figure 2.5: State 1 - *"Information gathering"*

2.4.1.2 State 2 - *"Process information in memory"*

In this state of the agents' cycle, several things happen. First, the *"room-representation"* of the agent is updated using the new information gathered and filtered **(1)**. This involves updating the information on the positions of known and/or new targets and agents. Then, based on this new information, each agent has to determine if it is in charge of sending the position of each target it sees to the user **(2)** (see 2.6). If so, it puts the information in a buffer that will be sent in the communication step **(6)**.

Next, several other pieces of information, needed to be transmitted to other agents, are stored in a buffer. These include:

- Current position of the agent. This way, all agents know the position of all others.
- Whether the agent started tracking or lost some target. This way, all agents know how many times each target is tracked.
- Information concerning the distributed Kalman filter (see section 3.2 and section B.4).

Furthermore, the agent updates the list of targets it needs to track **(4)**. Every time the confidence in the target's position (see section 4.2.1) goes below a certain threshold, the agent stops tracking the target. Inversely, if the confidence goes above the threshold, the agent starts tracking it.

When a new position of some target is gathered, the agent also runs an analysis to detect whether the target is moving **(3)**. This information will then be updated in the *"room-representation"*. The algorithm used is described in section F.2.

The final computation the agent does in this phase is to verify it is not too close to any other agent **(5)**. Indeed, if the positions and orientations of two agents are too similar (difference smaller that a predefined constant), the agents likely see the same part of the room with the same perspective. When that is the case, one of the agents initiates a default behaviour, moving around the room to search for an untracked target.

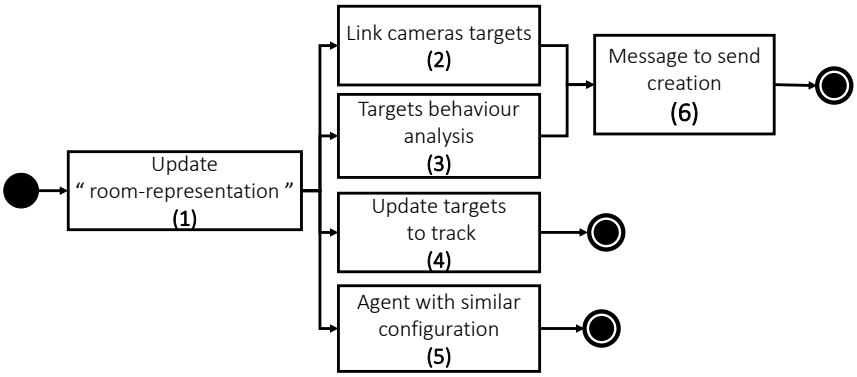


Figure 2.6: State 2 - *"Process information in memory"*

2.4.1.3 State 3 - *"Communication"*

In the communication step of the agent's cycle, each agent sends the messages it has stored in the message buffers, as described in the previous state **(5)**.When a message is sent, it is also removed from the buffer. In this stage, the agents also send and receive *"heartbeat"* messages, used to inform that they are not faulty (described in section B.1) **(2)**. Finally, each received message is treated, and responded to if needed **(3)** and **(4)**.

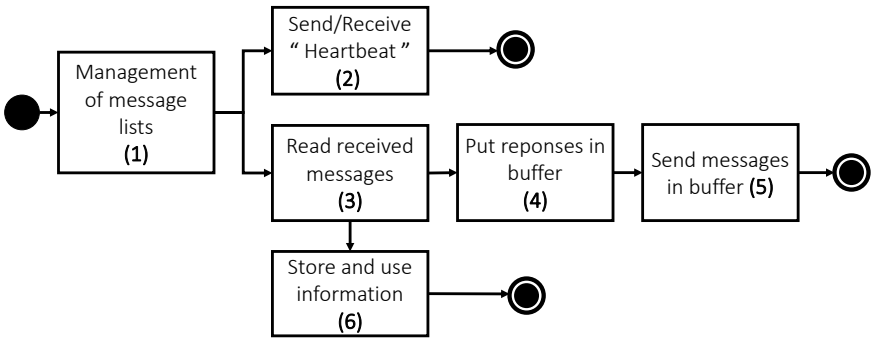


Figure 2.7: State 3 - *"Communication"*

2.4.1.4 State 4 - *"Motion"*

The final state of the agent is the one that contains all the logic determining the way the agent move. Given the position of agents and targets, the goal is to make each agent move such that each of the tracked targets is seen.

Each agent has a list of targets to track. At the beginning of this state, this list contains all targets seen by the agent. Then, the agent begins a process, inside which it loops until it finds a configuration (as defined in section **??**) allowing it to have a good view on all the targets in this list **(1)**. As long as it doesn't succeed to find such a configuration (either because the targets are too far apart, or because obstructions occur), the target with the lowest total priority (defined in section 4.5.1) is removed from the list and the agent tries again.

Using the configuration found, a controller then calculates the appropriate command to give to the agent **(2)** (see section 4.6). The agent then uses this command to actually move **(3)**.

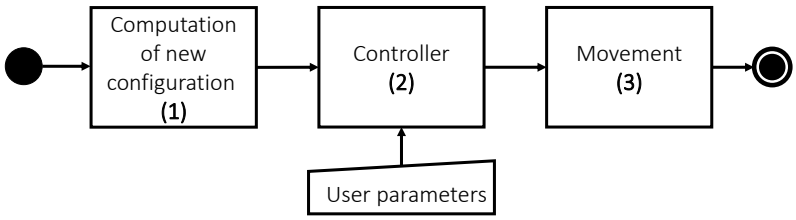


Figure 2.8: State 4 - *"Motion"*

2.5 Communication

Agents can communicate with each other by exchanging messages. A message contains a tag, a content, the sender identity and a list of receivers.

The tag represents the type of the message (detailed description in appendix B). Then, a message can be either broadcasted to multiple agents simultaneously or sent to a single receiver.

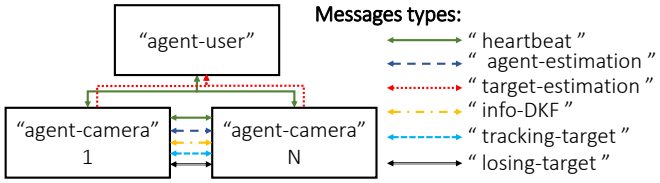


Figure 2.9: Agent communication overview description

Figure 2.9 illustrates the different messages exchanged by the agents. It can be noted that an *"agent-camera"* does not exchange the same messages with another *"agent-camera"* or with the *"agent-user"*.

2.5.1 Messages exchanged in introduction's use-case

Figures 2.10 and 2.11 summarize the exchanged messages for the *"agent-cameras"* (agent 0 and 1) and the *"agent-user"* (agent 100) respectively, during the execution of the introduction's use-case (section 1.2), in this case adding the distributed kalman filter (see section 3.2). Links to data and plots available in appendix H.

Figure 2.10 refers to the *"agent-camera"*. The *"agent-estimation"*, *"target-estimation"* and *"heartbeat"* messages are send at predefined rate. It can be noted that the *"Info-DKF"* messages represent an important part of the messages exchanged. An approach aiming at reducing them is proposed in section 3.6. Two *"tracking-target"* messages are sent at the beginning as the *"agent-camera 1"* detects the targets 0 and 2. Then, there is an obstruction at 5 s, therefore target 2 is lost. Consequently the agent also stops sending *"target-estimation"*. This could also have happened if the target moved in the region associated to the other camera (in this case the absence of *"target-estimations"* would not indicate that the target is lost). Finally at 6 s, the target is detected again thus a *"tracked-target"* is sent.

An important thing to notice is that *"agent-estimations"* are exchanged only between *"agent-cameras"* (there are none in figure 2.11). Indeed, the *"agent-user"* is not interested in the positions of the cameras. Furthermore, the *"agent-cameras"* use these messages to synchronize their *"room-representations"*. On the other hand, *"target-estimations"* are only exchanged between *"agent-camera"* and *"agent-user"*.

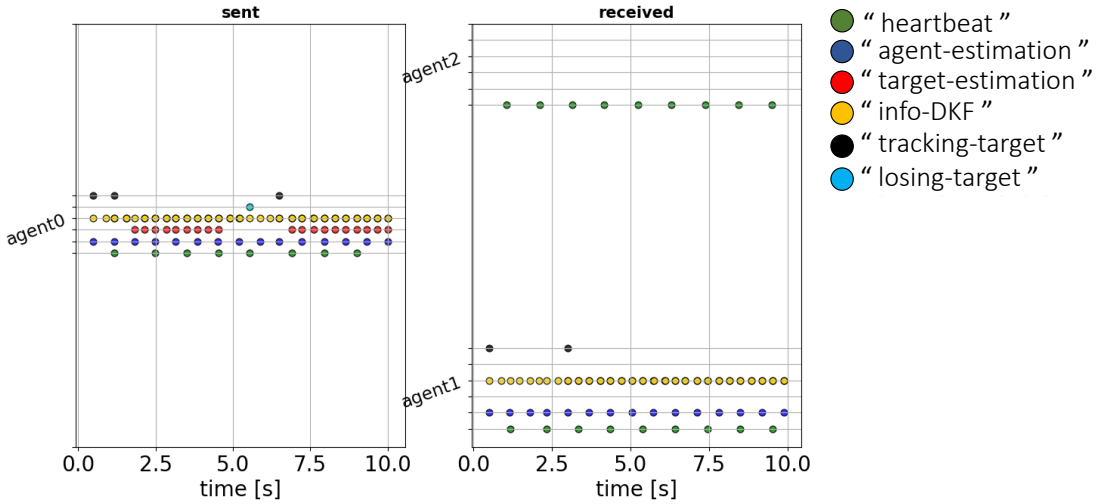


Figure 2.10: *“Agent-camera”* exchanges over 10 s

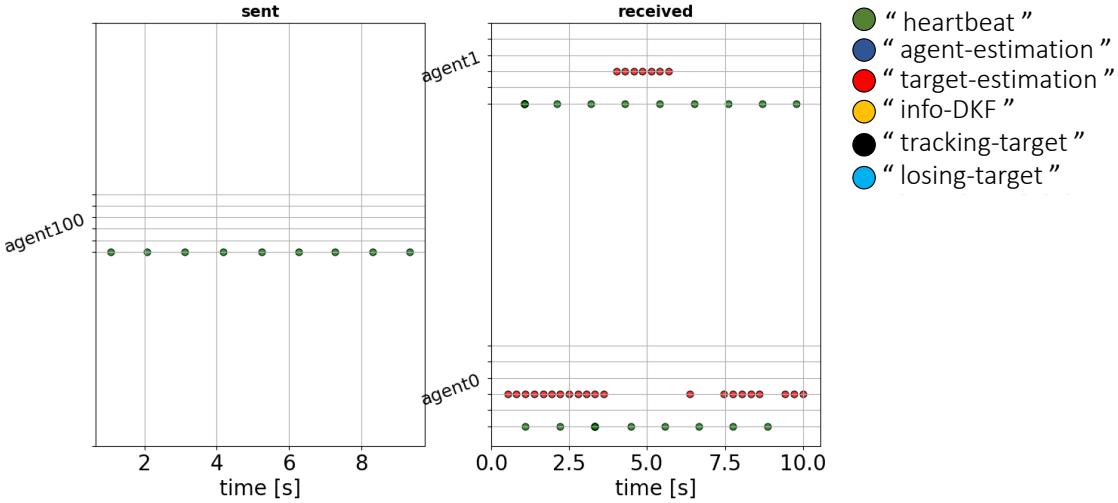


Figure 2.11: *“Agent-user”* exchanges over 10 s

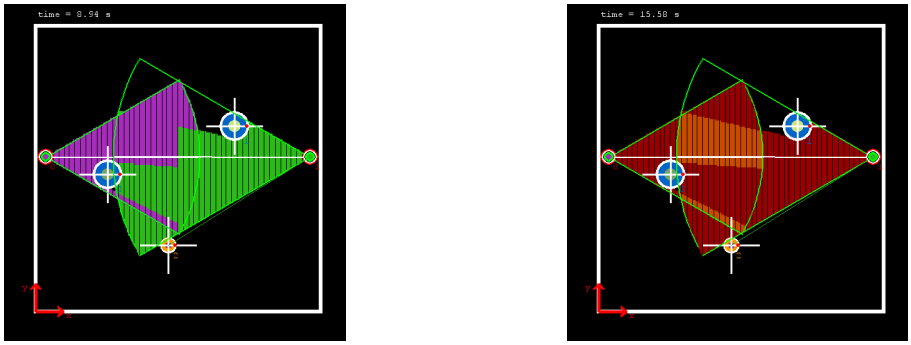
2.6 Coordination between agents

In every system composed of multiples sub-entities, coordination is always a concern. It defines a set of rules according to which sub-entities (the agents) interact optimally. In the implemented multi-agent system, coordination in the communication reduces the redundancy in the messages exchanged between the *"agent-cameras"* and the *"agent-user"*.

"Agent-cameras" should send the estimated positions to the user in a timely manner. After the distributed filtering, the agents that track the same targets have similar estimations on them. Therefore, in a matter of efficiency, all of them should not be sent to the *"agent-user"*.

Coordination is achieved through a criterion which acts as a trigger for sending the estimation to the *"agent-user"*. As the objective is to minimize the exchanged messages, it is important that this criterion only depends on information stored in the *"agent-camera's"* memory. In the implemented system, this criterion is the distance of the *"agent-camera"* to the target. The underlying assumption is that a camera will have a better view on a target if it is closer to it.

Figure 2.12a shows the distance based criterion for the room defined in the use-case of section 1.2. It illustrates which part of the room is associated to each *"agent-camera"*. To produce these figures, the algorithm of section F.1 is used. If the situation were static (ie. cameras and targets), a single computation would be sufficient to coordinate all *"agent-cameras"* at any time. To detect dynamic obstruction, the operation is repeated at each time step.

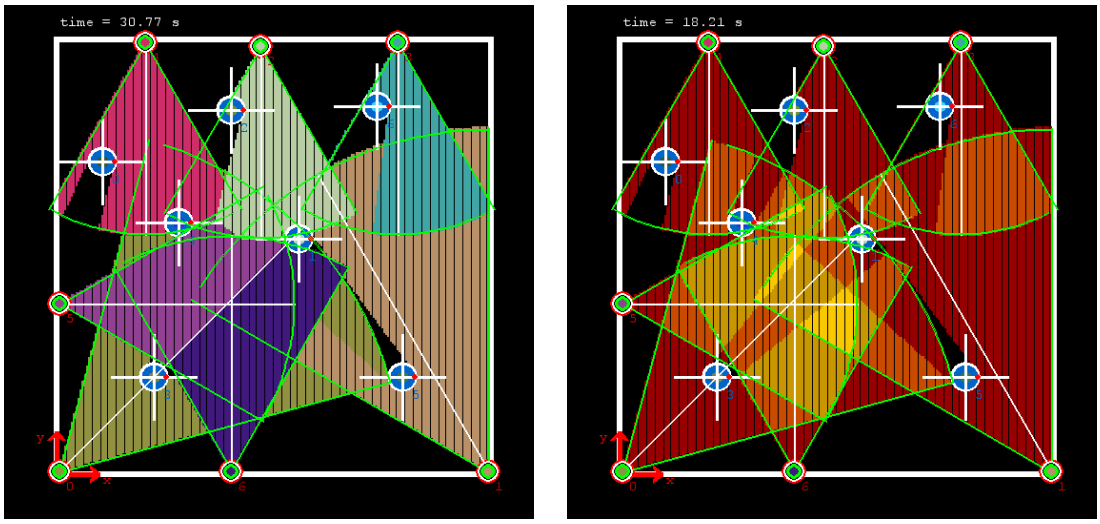


(a) Regions associated to cameras (b) Coverage

Figure 2.12: Obstruction algorithm application

Independently, the algorithm of section F.1 could also be used to position the camera appropriately in the room depending on the fix targets. This is depicted in the figure 2.12b, which shows the number of cameras that cover each part of the room. A black background means that no camera is covering the region. The red background indicates that one camera is covering the region. Then, the lighter the color, the more cameras are covering the region.

For illustration purposes, the same graphs for a more complex room are shown in figures 2.13a and 2.13b.



(a) Regions associated to cameras (b) Coverage in the room

Figure 2.13: Obstruction algorithm application

2.7 Discrepancies between *"room-representations"*

An *"agent-camera"* decides whether to send information to the *"agent-user"* according to a distance-based criterion (section 2.6): the agent closest to a target it sees is responsible for it.

To detect which agent is the closest to a target, the *"agent-camera"* making the decision needs estimates of the *"agent-camera's"* positions and the target's positions. When the agents are static, the *"room-representation"* will always be up to date with the agent positions. However, when agents move, *"agent-estimations"* messages are exchanged to keep the *"room-representation"* up to date.

In many cases, the estimates will be similar from one *"room-representation"* to the other. In these cases, actions undertaken by agents are consistent with one another. Figure 2.14 illustrates such a case. The *"room-representation"* *"A1-RR"* (*"room-representation"* of agent 1) and *"A2-RR"* (*"room-representation"* of agent 2) are similar and the estimates of target 1 and 2 are correctly transmitted to the *"agent-user"*. The *"USER-RR"* is thus correctly updated.

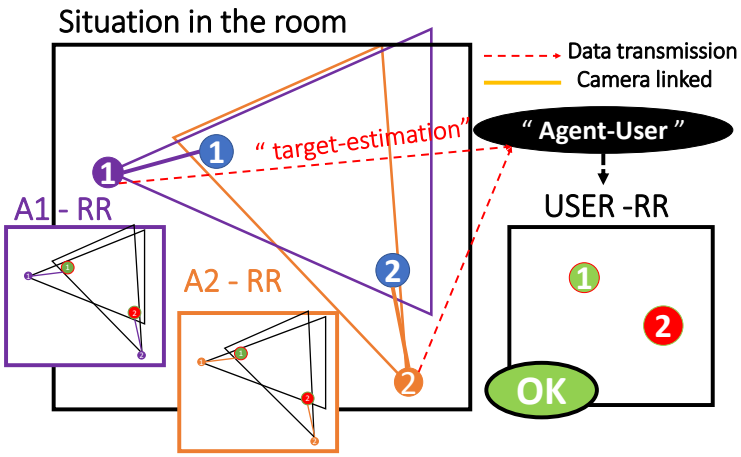


Figure 2.14: Synchronised *"room-representation"*

However, as *"target-estimation"* messages (2.5) are not exchanged between *"agent-cameras"*, the latter might have different believes concerning the position of the targets, which might cause the algorithm to fail.

This is illustrated in figure 2.15. Similarly to the case study previously discussed, two agents are tracking two targets, however the *"User RR"* is not correctly updated.

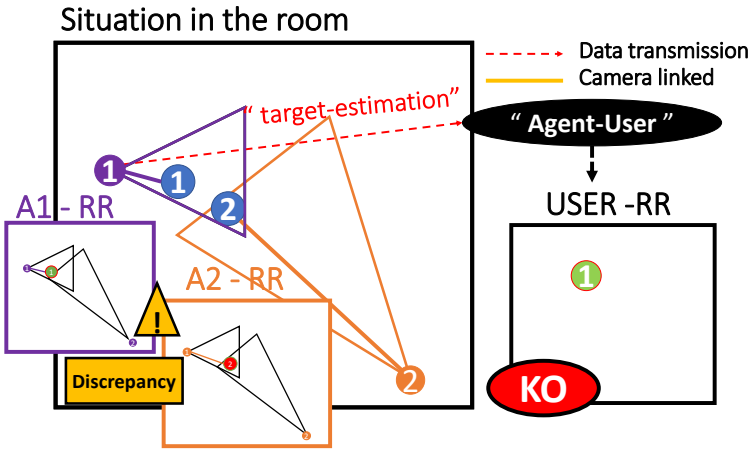


Figure 2.15: Discrepancy between *"room-representation"*

Figure 2.16 is a zoomed-in view of the *"room-representation"* in 2.15. The problem is described by analysing the scenario step by step.

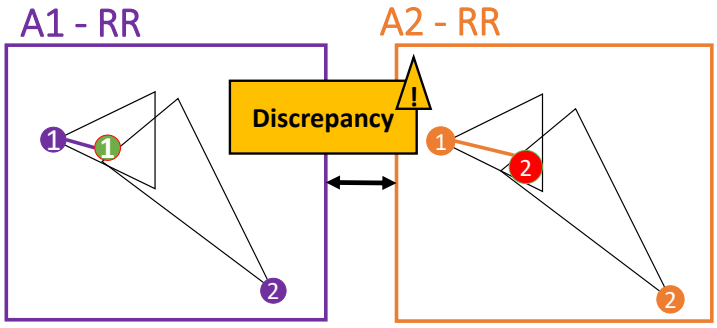


Figure 2.16: Zoom on *"room-representation"* in the figure 2.15

Initially, agent 1 sees target 1 (A1-RR). It calculates that agent 2 is too far to see it, therefore decides to send the information to the *"agent-user"*. Target 1 is obstructing target 2 for the agent 1, who therefore does not see it.

Then, agent 2 sees target 2 (A2-RR). It calculates that agent 1 is in a position to see it and is closer to it. Indeed, as the *"room-representation"* of agent 2 is not up to date regarding target 1, agent 2 assumes agent 1 will send the information regarding target 2 to the user and does not do so itself.

In this scenario, the *"agent-user"* will therefore not be informed at all about the position of target 2.

In order to prevent this from happening, the straight forward solution would be to exchange *"target-estimations"* at all moments. The *"room-representations"* of all agents would then be synchronized and this problem would not occur. However, the number of messages would then in average increase by $n * t_a$, where t_a is the average number of targets seen by an agent. As one of the main objectives is to have a scalable and light-weight system, exchanging this many messages is not acceptable.

The approach followed to solve the problem was different. Instead of making sure all agents are synchronized, mistakes can happen but multiple agents simultaneously send information to the user (lower than the number of agent in the room).

This way, the higher this number, the lower the probability that the *"agent-user"* does not receive any information about a target. In the example of figure 2.15, if two agents should send information about a target, the problem does not occur.

Another possible scenario could be the following. A target is located at the border of two regions associated to two *"agent-cameras"* (a region as shown in figure 2.12a). With the noise on the target and the delays for updating its position in the *"room-representations"*, it might happen that each *"agent-camera"* believes the target is within its associated region. Therefore, they both send the information to the *"agent-user"*.

This is a failure in the algorithm of section 2.6 because the same information is sent twice. However this is not a problem, as the main goal, tracking the target, is still fulfilled.

Such a case can be observed in the simple use-case of section 1.2. Figure 2.17 shows the *"room-representations"* of the two agents involved in the simple use-case, as well as the real position of the target. Both *"agent-cameras"* are not up-to-date and have different beliefs on the position of the target. In this scenario, both *"agent-cameras"* will send their estimations to the *"agent-user"*.

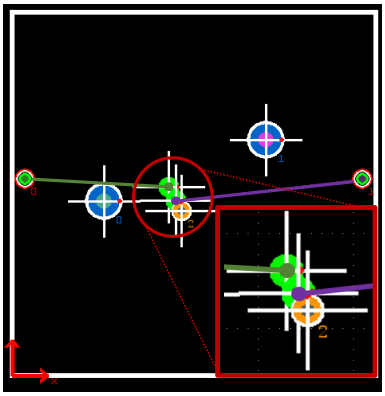


Figure 2.17: Discrepancy between *"room-representations"* in the simple use-case of section 1.2

2.8 Results and conclusion

In this chapter, *"agent-cameras"* and *"agent-users"* were introduced. They are autonomous, sociable, reactive and pro-active entities, that cooperate to form a so called multi-agent system. In the system, the information (estimates of target's position and speed) is first gathered by *"agent-cameras"* trough their associated camera and is then filtered (focus of chapter 3). Follows a communication step, between *"agent-cameras"* and the *"agent-user"*, leading to a summarized and global view of tracked target's trajectories (*"agent-user"* view). (Links to data, plots and a video showing the execution of the use case are available in appendix H)

Figure 2.18 shows a case study simulated in the software. There are three cameras with their field of view delimited by the green "triangles". Two fix targets (blue outer circle) and four moving targets (orange outer circle) are also displayed. The noiseless trajectories followed by the tracked targets over the execution of the simulation are shown (continuous dots). Furthermore, the pink dots correspond to the filtered measurements taken by the *"agent-cameras"*.

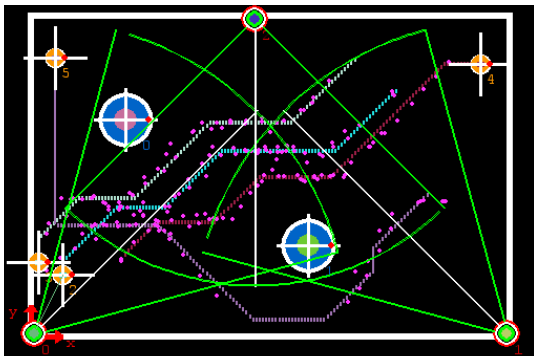


Figure 2.18: Use-case as represented in the *"simulation tab"*
link to videos available in appendix H

To coordinate the *"agent-cameras"*, a distance-based criterion is used. Figure 2.19 illustrates the regions inside which each of the *"agent-cameras"* has to communicate *"target-estimations"* to the *"agent-user"*.

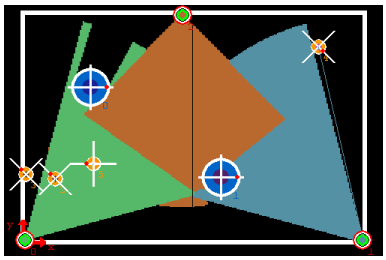


Figure 2.19: Region associated to cameras

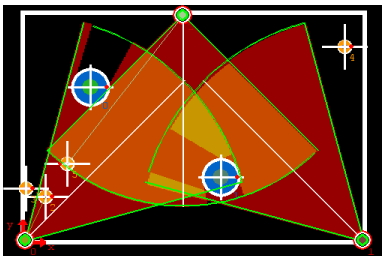


Figure 2.20: Room's coverage by the cameras

Figure 2.20 shows the coverage of the room. The black regions are not covered, in which case obstructions are unavoidable. As a reminder, red zones are under the surveillance of one camera, orange zones by two cameras and yellow zones by three cameras.

"Target-estimation" messages received by the *"agent-user"* are represented in the figure 2.21. The y axis groups the received messages by the *"agent-cameras"* that sent it.

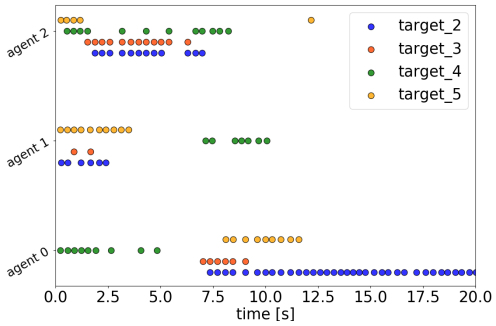


Figure 2.21: *"Target-estimation"* received by the *"agent-user"* from the *"agent-cameras"*

The *"agent-user"* continuously receives estimations for the blue target (number 2), thus the latter is tracked at all times. At the beginning, *"agent-camera"* 1 is in charge and sends 6 *"target-estimations"*, then *"agent-camera"* 2 takes charge and finally *"agent-camera"* 0 takes turn. Ideally, all targets would be tracked this way.

On the contrary, the yellow target was lost as no estimations were received between 2.5 s and 7.5 s. However, this is not the system's failure, as the obstruction is unavoidable. In fact, the target is moving in the bottom black region (in figure 2.20), therefore none of the agents are able to track it. After 13 s, the red, green and yellow targets are lost for the same reason.

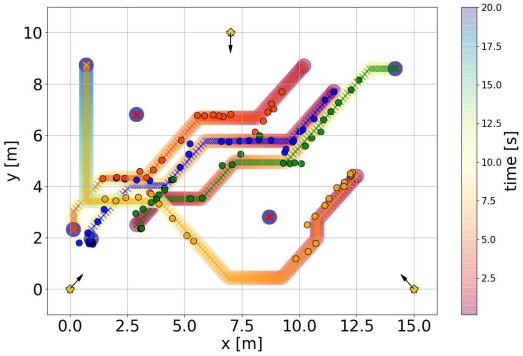


Figure 2.22: Tracked targets trajectories from the *"agent-user's"* perspective

Figure 2.23: legend for the figures 2.22 and 2.26

Finally figure 2.22 compares the artificially generated trajectories to the measurements received by the user. The number of received data depends on the rate at which *"target-estimations"* are sent, and can be adjusted (currently at 0.2 messages per second). In fact, the *"agent-cameras"* take more measurements (pink dots in the figure 2.18) than they send.

The system also needs to be robust against partial failure. An artificial failure is added to the case study presented so far to initiate a recovery process. The failure comes from *"agent-camera 2"* (on the top 2.24), that stops working at 2 s.

The failure can be observed by comparing the figures 2.21 and 2.25: in the second one, *"agent-camera 2"* is not sending *"target-estimations"* anymore after 2 s.

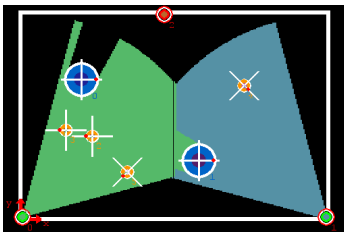


Figure 2.24: Region associated to cameras - *"agent-camera 2"* failing

When *"agent-camera 2"* fails, it is detected by the other *"agent-cameras"* as they do not receive *"heartbeat"* messages anymore. The remaining *"agent-cameras"* will immediately try to compensate by recomputing their associated region. The new regions therefore become larger, as shown in the figure 2.24 (to be compared with 2.19, computed before the failure happens).

Comparing figures 2.21 and 2.25, it appears clearly that in the scenario where *"agent-camera 2"* fails, the *"agent-cameras 0 and 1"* take over. This way, the targets keep being tracked, even with the failure of *"agent-camera 2"*. The result is that from the *"agent-user's"* perspective, the estimated trajectories contain the same number of estimations. This can be seen by comparing figures 2.22 and 2.26. Of course, compensation for a failed camera is not always possible, depending on the coverage of the room.

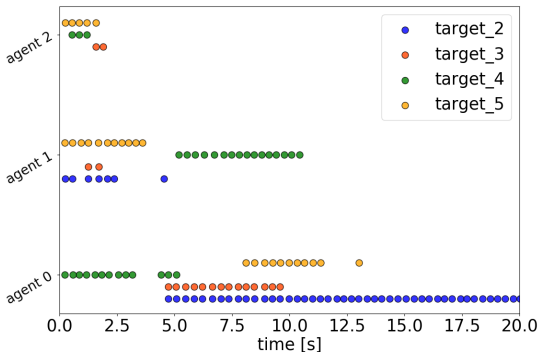


Figure 2.25: *"Target-estimation"* received by the *"agent-user"* from the *"agent-cameras"* - *"agent-camera 2"* failing

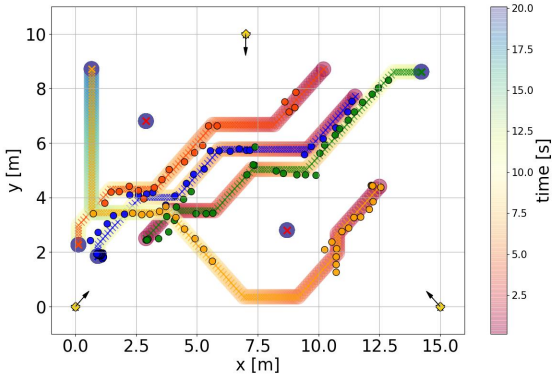


Figure 2.26: *"Target-estimation"* received by the *"agent-user"* from the *"agent-cameras"* - *"agent-camera 2"* failing

Chapter 3

State estimation from noisy measurements

Assuming the measurements collected from the sensors of the agents are noisy, a noise removal strategy has to be considered. The well established Kalman Filter is chosen for this project, amongst others because of its efficient online data filtering.

At first, a local Kalman filter is implemented: each agent estimates the position of the targets it is tracking using only the information it has collected. Since the agents constitute the nodes of an interconnected network, information is exchanged between them to improve the estimates. For that purpose, a Distributed Kalman Filter was implemented. Appendix C.1 provides a brief reminder of the traditional Kalman filter along with its equations. The notation used from here on is also introduced.

This chapter starts by describing the model defined for the local Kalman filter of each agent. Then, follows its distribution amongst the agents of the network. Finally, the chapter ends with an experimental comparison between the local and the global estimations.

3.1 Implementation of the Kalman filter

For the studied system, the state vector is defined as $\boldsymbol{x}(t) = [x\ y\ v_x\ v_y]^T$, where x, y are the coordinates of the target in the 2D plane and v_x, v_y are its speeds in each of the two axis of the plane. The targets are modeled as punctual objects. Moreover, they are assumed to move with constant speed and their acceleration is assumed to be negligible. This type of movement is described by a uniform rectilinear motion.

From its corresponding equations, the resulting transition matrix is:

$$\boldsymbol{F}(t) = \boldsymbol{F} = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.1}$$

where dt is the time between two observations.

For simplification purposes, the targets are moving in a uniform rectilinear motion in the simulation created, therefore the process noise can be omitted. Then, two different observation models are considered. The first assumes that the sensors on the agents can measure position as well as velocity, in which case, the observation matrix is defined as the identity matrix: 3.2 . The other case assumes that the sensors can only measure positions. In this case, the observation matrix is defined as: 3.3.

$$\boldsymbol{H_1}(t) = \boldsymbol{H_1} = \mathbb{I}[4,4] \tag{3.2}$$

$$\boldsymbol{H_2}(t) = \boldsymbol{H_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.3}$$

In all executions presented, $\boldsymbol{H_1}$ will be used. The filters' initial state vector is set to the first observation and the initial covariance is set to the covariances of the generated measurements.

Considering the sensors attached to the agents can produce outliers in the observed data, a validation step is included every time the agent receives a new observation. This step is described in section 3.2.1. This validation, however, also has another purpose. Even though the targets move in a uniform rectilinear motion, it is not assumed they don't turn. For the filters to produce relevant estimates with turning targets, they have to be reinitialised every time a target turns. In order to detect turns, when an agent receives an observation outside its validation region, the filter for the corresponding target is reinitialized.

A final point worth noting is that the agents do not necessarily take observations at constant time intervals (used in 3.1). A first solution would be to assume the differences in time intervals are negligible and use a constant average time interval. However, in order to improve the accuracy of the estimation, a different approach was taken. Each agent of the system stores the time at which the last observation was made. This way, when a new observation arrives, the agent can calculate the time interval between the observations. This solution works when trying to estimate the current position of a target, however it is not applicable when calculating the predicted positions. In this case, a constant average time has to be used.

3.2 Distributed Kalman Filer

Now that the local filter is implemented, different agent's observations on commonly tracked targets can be combined to improve the local estimates. A first idea would be that each agent sends the raw observations to a central unit, which would implement a Kalman filter regrouping all of them, with :

$$\boldsymbol{z}(k) \equiv [\boldsymbol{z_1}^T(k), ..., \boldsymbol{z_s}^T(k)]^T \tag{3.4}$$

$$\boldsymbol{H}(k) \equiv [\boldsymbol{H_1}^T(k), ..., \boldsymbol{H_s}^T(k)]^T \tag{3.5}$$

$$\boldsymbol{v}(k) \equiv [\boldsymbol{v_1}^T(k), .., \boldsymbol{v_S}^T(k)]^T \tag{3.6}$$

$$\boldsymbol{R}(k) = \mathbb{E}\{\boldsymbol{v}(k)\boldsymbol{v}^T(k)\} = \text{blockdiag}\{\boldsymbol{R_1}(k), ..., \boldsymbol{R_s}(k)\} \tag{3.7}$$

where $\boldsymbol{z(k)}$ is the stacked observation vector, $\boldsymbol{H(k)}$ is the observation matrix, $\boldsymbol{v}(k)$ is the process noise, $\mathbb{E}$ is the expectation symbol and blockdiag is an operator to represent the block diagonal matrix constructed using its arguments.

With this model, the central unit makes an estimate on the state variable and sends it back to each of the agents. Such a system is proposed by [2] in an organization they call hierarchical. This way, however, the system depends on the central unit: if it fails for some reason, the system cannot make estimations.

Another decentralized approach was proposed by [3]. In their architecture, which they call Distributed Kalman Filter (DKF), each node of the network sends information to the others. This information is then directly assimilated by each agent in order to locally reconstruct a global estimate of the state.

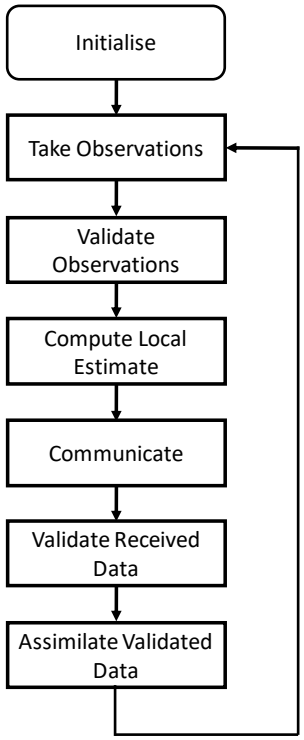


Figure 3.1: Synchronous DKF algorithm's flowchart

Figure 3.1 shows the main steps each agent follows. It starts by taking observations, then validates them (a step especially important in the case of multi-target tracking, allowing to differentiate targets in many cases). After that, a first local estimate of the state variable is computed, using the traditional Kalman filter.

Then comes the communication step, where relevant data is exchanged. Again, this data is validated by the receiver (not the sender). Finally, the data received is assimilated into the local estimation of the state, resulting in a locally computed global estimate of the state. A brief explanation of these steps follows. For more detail, see [3].

3.2.1 Nodal observations validation

In order to validate the observations gathered by some node, a "validation region" centered on the state prediction is defined and only observations that lay inside it are accepted. The validation region is constructed using a χ^2 test on the statistical significance that $\boldsymbol{z}(k) = \hat{\boldsymbol{z}}(k|k-1)$:

$$\tilde{\boldsymbol{z}}^T(k)\boldsymbol{S}^{-1}(k)\tilde{\boldsymbol{z}}(k) < \gamma, \tag{3.8}$$

where γ is found empirically, $\tilde{\boldsymbol{z}}(k)$ is the innovation as defined in equation (C.11) and $\boldsymbol{S}(k)$ is its covariance matrix

$$\boldsymbol{S}(k) = \boldsymbol{R}(k) + \boldsymbol{H}(k)\boldsymbol{P}(k|k-1)\boldsymbol{H}^T(k). \tag{3.9}$$

This step can be important for two reasons. First, the sensor can be malfunctioning and provide wrong observations. Second, in the case of multi-target tracking, the validation could be used for data identification.

In the implemented system, physical sensors are simulated by a non-faulty random number generator and target identification numbers are exchanged. For completeness, the nodal validation is implemented nonetheless.

3.2.2 Internodal validation

Observations from other nodes have to validated as well, for the same reasons are the nodal validation. As in the nodal validation, a "validation region" centered on the state prediction is defined. A χ^2 test on the statistical significance that $\boldsymbol{z}_j(k) = \hat{\boldsymbol{z}}_i(k|k-1)$ (i.e. the difference between the observation received from node j, and the value of the observation as predicted by the receiving node, i. The authors of [3] derive the following condition:

$$\eta_{ij}^T(t_j)\boldsymbol{N}_{ij}^{-1}(t_j)\eta_{ij}(t_j) \leq \gamma, \tag{3.10}$$

$$\eta_{ij}(t_j) = \boldsymbol{Y}_j^-(t_j)\boldsymbol{X}_j(t_j) - \boldsymbol{H}_j^T(t_j)\boldsymbol{H}_j(t_j)\hat{\boldsymbol{x}}_i(t_j|\tau_j), \tag{3.11}$$

$$\boldsymbol{N}_{ij}(t_j) = \boldsymbol{H}_j^T(t_j)[\{\boldsymbol{H}_j(t_j)\boldsymbol{Y}_j(t_j)\boldsymbol{H}_j^T(t_j)\}^{-1} + \boldsymbol{H}_j(t_j)\boldsymbol{P}_i(t_j|\tau_i)\boldsymbol{H}_j^T(t_j)]\boldsymbol{H}_j(t_j), \tag{3.12}$$

$$\boldsymbol{Y}_j^-(t_j) = \boldsymbol{H}_j^T(t_j)\{\boldsymbol{H}_j(t_j)\boldsymbol{Y}_j(t_j)\boldsymbol{H}_j^T(t_j)\}^{-1}\boldsymbol{H}_j(t_j). \tag{3.13}$$

While the nodal validation step is not required in the simulated system, this is not the case for the internodal validation. Especially when a simulation is executed with a large number of agents (5 or more), non-negligible delays start to appear. In this case, the exchanged measurements might take long to arrive to the destination node, resulting in out-of-date observation being received.

3.2.3 Computation of local estimate

The local estimate is computed using a traditional Kalman filter, where equations (C.5)-(C.11) are used.

3.2.4 Assimilation equations

The authors of [3] is based on the hierarchical formulation derived by [2]. In this formulation, the observation vector $\boldsymbol{z}(k)$ and the noise vector $\boldsymbol{v}(k)$ are unstacked into s subvectors corresponding to the measurements of each of the sensors and the observation matrix $\boldsymbol{H}(k)$ into corresponding submartices

$$\boldsymbol{z}(k) = [\boldsymbol{z}_1^T(k), ..., \boldsymbol{z}_s^T]^T \tag{3.14}$$

$$\boldsymbol{H}(k) = [\boldsymbol{H}_1^T(k), ..., \boldsymbol{H}_s^T]^T \tag{3.15}$$

$$\boldsymbol{v}(k) = [\boldsymbol{v}_1^T(k), ..., \boldsymbol{v}_s^T]^T. \tag{3.16}$$

It is assumed the partitions are uncorrelated

$$\mathbb{E}[\boldsymbol{v}(k)\boldsymbol{v}^T(k)] = \text{blockdiag}\{\boldsymbol{R}_1(k), ..., \boldsymbol{R}_s(k)\}, \tag{3.17}$$

From there, [3] derives the equations of assimilation of the variance and state, used by agent i to incorporate the estimation of the other agents j:
Assimilation of variance

$$\boldsymbol{P_i}^{-1}(k|k) = \boldsymbol{P_i}^{-1}(k|k-1) + \underbrace{\sum_{j=1}^{s} \boldsymbol{H}_j^T(k)\boldsymbol{R}_j^{-1}(k)\boldsymbol{H}_j(k)}_{\text{variance error info}}. \tag{3.18}$$

Assimilation of state

$$\hat{\boldsymbol{x}}_i(k|k) = \boldsymbol{P}_i(k|k)[\boldsymbol{P}_i^{-1}(k|k-1)\hat{\boldsymbol{x}}_i(k|k-1) + \underbrace{\sum_{j=1}^{s} \boldsymbol{H}_j^T(k)\boldsymbol{R}_j^{-1}(k)\boldsymbol{z}_j(k)}_{\text{state error info}}]. \tag{3.19}$$

The terms called `state error info` and `variance error info` in the above equations are therefore the terms to be communicated between agents in order to compute the global estimate.

The locally computed global estimate constructed by each node is - after the information from all other nodes has been assimilated - mathematically equivalent to a single Kalman filter constructed directly from all the stacked observations.

It is important to notice that the equations 3.18 and 3.19 apply to the synchronous case, where all agents communicate simultaneously. This being difficult to guarantee in a distributed system, [3] adapts the equations for the asynchronous case by replacing the fixed time steps k with continuous time t and an increment δt. The adapted equations for asynchronous global assimilation become:

Assimilation of variance

$$\boldsymbol{P_i}^{-1}(t_j|t_j) = \boldsymbol{P_i}^{-1}(t_j|\tau_i) + \underbrace{\boldsymbol{H}_j^T(t_j)\boldsymbol{R}_j^{-1}(t_j)\boldsymbol{H}_j(t_j)}_{\text{variance error info}}, \tag{3.20}$$

Assimilation of state

$$\hat{\boldsymbol{x}}_{\boldsymbol{i}}(t_j|t_j) = \boldsymbol{P_i}(t_j|t_j)[\boldsymbol{P_i}^{-1}(t_j|\tau_i)\hat{\boldsymbol{x}}_{\boldsymbol{i}}(t_j|\tau_i) + \underbrace{\boldsymbol{H}_j^T(t_j)\boldsymbol{R}_j^{-1}(t_j)\boldsymbol{z}_j(t_j)}_{\text{state error info}}, \tag{3.21}$$

where t_j is the time at which the observation at node j was made, τ_i is the difference between the t_j and the last observation of agent i.

In order to reach the asynchronous equations for assimilation, [3] makes the assumption that the communication delay is negligible. Considering such a network of agents would likely communicate via its own network, this assumption should be reasonable. However, the assumption can be relaxed by including timestamps when exchanging messages. On the other hand, the drawback of this approach is that the load of messages then increases.

3.2.5 Communication

As each node sends the calculated `state error info` and `variance error info` to every other node in the network, the communication cost is proportional to $n*(n-1)*\boldsymbol{p}$, where $\boldsymbol{p}$ is the communication cost per message. Furthermore, `state error info` $\in \mathbb{R}^n$, where n is the size of the state vector and `variance error info` $\in \mathbb{R}^{n\times n}$, meaning that the communication cost per message $\boldsymbol{p} \in \mathcal{O}(n+n^2)$.

3.3 Implementation of Distributed Kalman filter

In the simulation, the implementation of DKF follows the steps described in the previous section. The models are the same as the ones described in section 3.1. It is worth looking at the behaviour of the system when a target changes direction.

Figure 3.2 shows a scenario where a target rotates, from trajectory A to trajectory B, while agents 1 and 2 track it. In this scenario, if agent 1 detects the turn one frame later than it actually happened, it will make a wrong estimation and the error will propagate to agent 2, even if the latter had correctly detected the turn. However, when agent 2 receives agents' 1 data, it will first validate it before assimilating it. It will likely detect the data is outside its validation region and therefore not assimilate it. This is one of the reasons internodal validation is important.

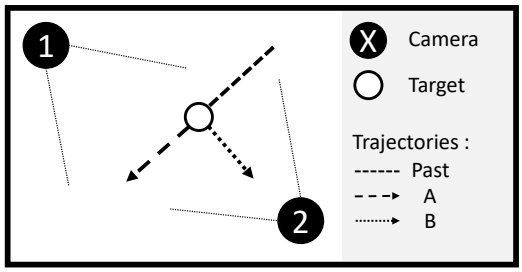


Figure 3.2: A target turning in the room

The observation matrices are constant in time and the same for all agents in the implemented system. However, if needed, each agent can have different observation matrices, as well as not constant in time.

3.4 Use of Kalman Filter for predictions of future positions

The Kalman filter offers the possibility to estimate the future positions of the targets. This is achieved, by plugging in the output of equations C.5 and C.7 into equations C.8 and C.9. These predictions are not made in a distributed fashion: every agent makes his own predictions. These predictions can then be used instead of the filtered measurements in order to update the *"room-representations"* of the *"agent-cameras"*.

3.5 DKF Performance Evaluation

In this section, the performance of the DKF is evaluated. Since the distributed filter requires a significant number of messages to be exchanged, its use should be justified by an improved noise reduction. The root mean squared error (RMSE) is used as a noise reduction metric. It is defined as:

$$RMSE = \sqrt{\sum_{k}(e_k^2)},$$

where e_k is the error of the kth observed or filtered data.

Ideally, the performance of the filter would be compared between executions of the same scenario, with an increasing number of *"agent-cameras"*. However, this is not a relevant comparison, as the software is implemented on a single machine. This means that all agents will not continuously run in parallel. Subsequently, when the number of agent increases, each of them is idle for a longer period of time, resulting in an increased RMSE.

Instead, during an execution, the local estimates computed by the *"agent-cameras"* before the internodal assimilation are stored. This way, the local estimates can be compared to the global estimates for each execution.

This was performed for the room of figure 3.3, with 2, 3, 4 and 5 agents. The positions of the agents are irrelevant, as each of them sees the whole room. The result is depicted in figure 3.4 (see appendix C.2 for more information on the experiment). For the scenario with two *"agent-cameras"*, the global estimate is less accurate than the local one. A possible explanation might be that the overhead due to the communication delays and scheduling effects is more important than the gain, and the global estimate has, in average, a larger RMSE than the local one. On the other hand, for the cases with 3, 4 and 5 *"agent-cameras"*, the estimate is slightly improved by using the DKF. Moreover, the estimate improvement seems to become larger with the number of *"agent-cameras"* (improvements of 3.2%, 5.4% and 7.8% for 3, 4 and 5 *"agent-cameras"* respectively).

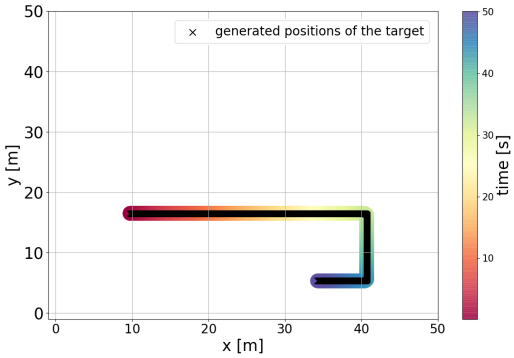


Figure 3.3: Map used for the performance evaluation of DKF

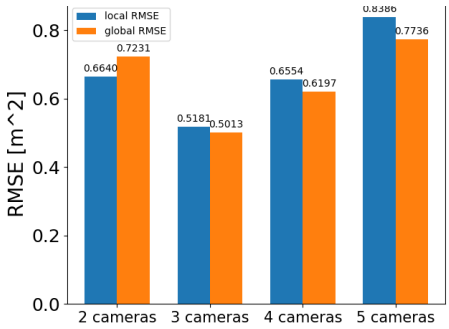


Figure 3.4: RMSE of local vs distributed estimates of state

3.6 Reduction of messages vs performance

As figure 2.10 illustrates, the number of messages exchanged for the DKF is by far larger than any other of the system's messages. A simple idea to reduce the number of messages was explored, where an agent sends its local estimations only if the innovation for the corresponding measurement is larger than a predefined lower bound. Intuitively, this condition ensures that a message is exchanged only if the amount of new information (represented by the prediction error, i.e. the innovation) is large enough.

The introduction of an innovation bound is a trade-off. On one hand, it will reduce the number of exchanged messages, while on the other hand information is lost. If the value of the bound is very large, the resulting global estimation will be the same as the local estimation of the state, as none of the messages will ever be sent. However, if the value of the bound is very low, the messages related to the DKF will not be reduced.

In order to evaluate the usefulness of the innovation lower bound, an experiment has been implemented. The simulation is executed 20 times in the room presented in figure 2.10. The difference between the mean local and global state estimates expressed in percentage of the local estimate (equation 3.22), as well as the mean percentage of DKF messages (out of the total number of sent messages) are plotted against a varying innovation lower bound. The results are illustrated in figure 3.5 (more information on the experiment can be found in appendix C.1).

$$\Delta_{rmse} = 100\left(\frac{local_{rmse} - global_{rmse}}{local_{rmse}}\right) \tag{3.22}$$

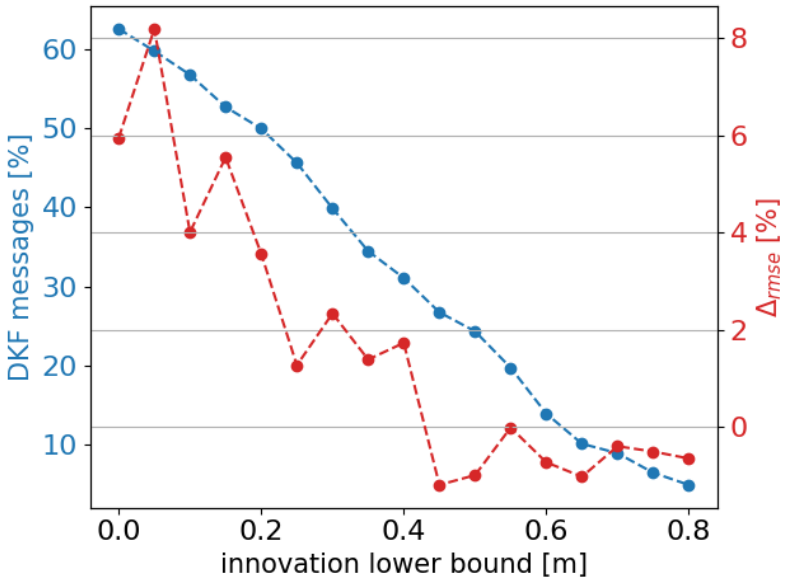


Figure 3.5: Percentage of DKF messages and Δ_{rmse} (equation 3.22)

Unsurprisingly, the percentage of DKF messages quickly decreases with the increasing innovation lower bound.

An observation that can be made concerns the evolution of the difference between the RMSE of local and global estimations. Until a lower bound of 0.4, the global estimations are more accurate than the local ones (positive increase in RMSE). After this point, however, the RMSE of the two estimations are nearly equal. This is due to the internodal validation step of the agents: when they receive an observation from another node, they verify if it falls inside a validation region. If only observations with a large innovation are exchanged, they are likely to all fall outside the validation region of the agent, therefore not being assimilated to the local state estimation.

3.7 Conclusion

A distributed noise reduction method was implemented, the DKF. In its synchronous form, this filter is mathematically equivalent to a central Kalman filter on the collected observations of all agents. In its asynchronous form, however, it is not mathematically equivalent. Furthermore, timestamps have to be exchanged for its performance not to drop when communication delays are not negligible.

While an increased noise reduction is achieved by the DKF, its difference in state estimation with the traditional local Kalman filter does not seem significant enough to justify the increased communication costs. However, the evaluation was performed on the simulation software, which is implemented on a single machine. Further evaluation on a truly distributed system is therefore necessary to draw decisive conclusions.

Finally, depending on the specific use case, minimizing the communications might be more important than having a precise state estimation. A method to reduce the amount of exchanged messages was derived, by only sending observations if their associated innovation is larger than a constant. Once again, this experiment should be conducted on a system that is physically distributed to draw conclusive results. If a significant message reduction can be achieved without impairing the performance of the filter too much, the method can then be used to look for a suitable middle ground, by defining the lower innovation bound to a specific value.

Chapter 4

Agent movement

A multi-agent system composed of fix cameras, as described in chapter 2, limits the possibility of action undertaken by the agents to prevent obstructions. Therefore, this chapter considers an improvement by enabling the agents to move in the room. This improvement is dedicated to the *"agent-cameras"* defined in chapter 2 and uses as input the filtered data as explain in chapter 3. The first section presents the flow chart of the agent's movement while the next ones dive into the algorithms involved in each of its steps.

4.1 Agent's movement flowchart

Figure 4.1 depicts the steps followed by an *"agent-camera"* to prepare and perform its next move.

The *"configuration computation"* block accepts a standardized input, the *"room-representation"* of an agent, which contains the positions of the targets. Based on this input, the targets to track are chosen, according to their confidence score (section 4.2.1). The *"configuration computation"* then outputs a configuration (section 4.2.3), which attempts to ensure that each tracked target is visible.

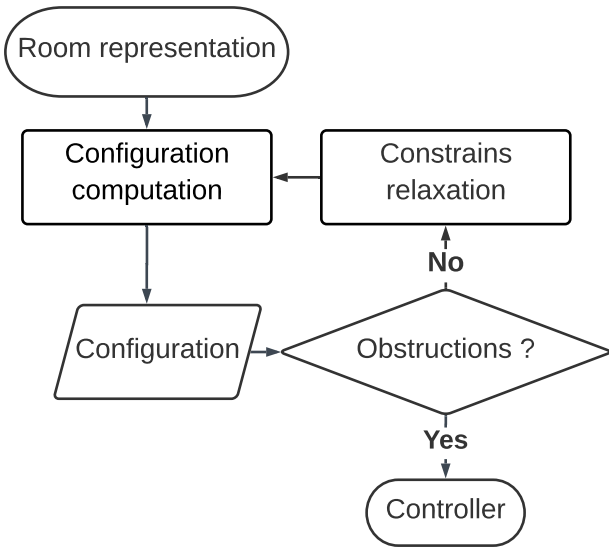


Figure 4.1: Steps preparing agent movement

Afterwards, the output configuration goes into a validation step: the configuration should not result in obstructed targets and its score (section 4.4.1) should be sufficiently high.

If an invalid configuration is found, a target is removed from targets to track. Thus constraints are relaxed as the motion algorithm is performed again with a reduced number of targets until a valid configuration is found.

Finally when a configuration meets the requirements, it is passed to a set of controllers that will then provide the appropriate movement commands to the agent.

This chapter starts with section 4.2, where some general hypothesis are explained. Then, sections 4.3, 4.4 and 4.5 dive into the other steps of the movement cycle. Finally, section 4.6 describes the controllers used to move the *"agent-cameras"* towards the configuration computed in the previous steps.

4.2 Modeling hypothesis

4.2.1 Target

A target is represented by a circular shape (see figure 4.2). It is described by a position x_t, y_t, a radius r and speeds v_{xt}, v_{yt}. It moves in a uniform linear motion.

There are also two main types of targets. *"Set fix"* and *"Unknown"*, corresponding to static and dynamic targets respectively.

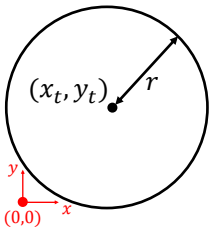


Figure 4.2: Target model

Associated with each target, is a confidence score, representing the accuracy of the position of target estimation with respect to time. When a target has just been updated (meaning the target was recently in the agent's field of view), this confidence score is set to a value proportional to the variance on the estimation produced by the Kalman filter (equation C.9). Several functions defining the decay are possible as shown in figure 4.3. Below the *"threshold"* (doted black line), the data is considered out of date.

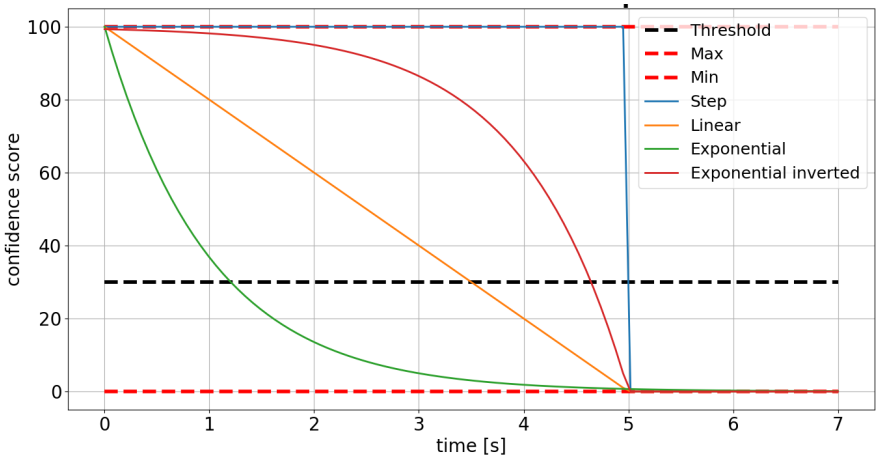


Figure 4.3: Possible decay of the score confidence in terms of time

4.2.2 Camera

A Camera is represented by a point x_c, y_c. Its orientation is defined by $\alpha \in [-\pi, \pi]$ and the field opening is defined by $\beta \in [0, \pi]$.It is also characterised by a limited field depth (see figure 4.4).

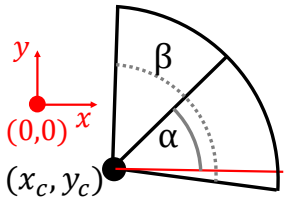


Figure 4.4: Camera model

A camera has up to four degrees of freedom: x_c, y_c, α, β since field depth is inversely proportional to β. The degrees of freedom can be modified using $v_{cx}, v_{cy}, \dot{\alpha}, \dot{\beta}$.

From these four degrees of freedom (DOF), we have defined 4 types of agents:

- **Fix** (1 DOF): The zoom, which corresponds to the angle β.
- **Rotative** (2 DOF): The zoom and orientation (α_c).
- **Rail** (3 DOF): Three degrees of freedom. The zoom, the orientation and the movement along only one axis. Such a movement could be considered in a camera fixed on a predefined track. It is represented with a parametric curve.

$$(x, y) = f(s), s \in [0, \text{rail length}] \tag{4.1}$$

- **Free** (4 DOF): The zoom, the orientation and movement along the two axis.

The model is only valid for an endocentric lens, which has the following effect: if several objects have the same size but are placed at different distances from the camera, they will appear with different sizes on the camera chip. The further an object is from the lens, the smaller it will appear. Other types of lenses such as telecentric or hypercentric are not modelled.

4.2.3 Configuration

A configuration is the output of the *"motion algorithms"*, which groups multiple elements and also corresponds to a standardized input for the controller. (see figure 4.1). It includes the current state of the camera $(x_c, y_c, \alpha, \beta)$, a list of tracked targets taken into account by the algorithm and a score (as described in section 4.4.1).

4.3 Configuration search

Based on a list of targets to track along with their positions, the goal is to find where the agent should position itself in order to view each target. Depending on the number of targets, the agent acts differently.

4.3.1 No tracked targets

It is possible that an agent does not have any targets in its field of view. In this case, it will initiate a default behaviour, attempting to cover different regions of the room, in search of targets. Agents that can rotate, do so until they find a target. When they detect a wall in their field of view, they switch directions of rotation, in order to avoid irrelevant configurations. Agents that can also move randomly explore the room. Two types of random movement have been defined. In the first one, the agent sets a random position in the room and move towards it for a predefined number of seconds. In the second type, the agents move towards the random position until they are close enough to this position. When it has been reached, a new random position is generated and the agent moves in this direction.

As each agent has the positions of the others (in its *"room-representation"*), it could use this information when there are no targets in its field of view. A possibility would be to define a behaviour where the agent moves towards parts of the room that no other agent is covering.

4.3.2 One tracked target

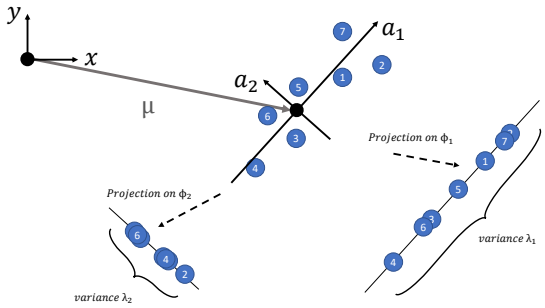
If only one target is in the field of view of the agent, the agent will try to position itself a predefined distance from the target. This distance, along with the position of the target, define a circle of acceptable positions on which the agent can position itself. The agent will then choose the point on this circle that is closest to its current position. Then, the agent will orient itself accordingly. β is calculated as: $\beta = M * arctan(r/d)$, where d is the distance between the target and the camera, r is the target radius and M is a security margin.

4.3.3 More than one tracked target

In case two or more targets are tracked, a method called Principal Component Analysis is suited. The Principal Component Analysis (PCA) or Karhunen Loeve theorem is often used for data compression or features recognition. The fundamental equations describing the method can be found in appendix E. This section proposes an adaption of the PCA method to find a suitable configuration for a camera based on the targets in its field of view.

4.3.3.1 PCA applied on the targets

Given an arrangement of n targets, PCA aims to maximize the number of visible targets. A PCA-analysis brings to light information hidden in the covariance matrix by decomposing it in the form of the equation 4.3.



$$\mathrm{x} = [(x_1^i, y_1^i), ..., (x_n^i, y_n^i)] \tag{4.2}$$

$$C_{\mathrm{XX}} = \begin{bmatrix} \boldsymbol{a_1} \\ \boldsymbol{a_2} \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} \end{bmatrix} \tag{4.3}$$

Figure 4.5: PCA - application use case

Illustrated in the figure 4.5, $\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ are the eigenvectors and λ_1 and λ_2 are the eigenvalues of the covariance matrix of the targets positions. Eigenvalues measure the spread of the data, while eigenvectors contain the direction of spread. From there, three different cases are analysed.

1. $\lambda_1 >>> \lambda_2$: Signal theory states that the energy contained in a signal is the sum of its covariance matrix eigenvalues (appendix: equationE.5). In this case, energy $= \lambda_1 + \lambda_2 \simeq \lambda_1$. Consequently, the signal is correctly approximated when only the position on the $\boldsymbol{a_1}$ axis is measured. The error is then given by λ_2 (appendix: equation E.6).

This can be applied to the cameras. The bisector of their field of vision has to be perpendicular to the $\boldsymbol{a_1}$ vector, in order to maximize the probability to detect the targets (as they are spread along this axis). Moreover, the bisector should either intersect the mean or the median of the target's positions, depending on the application. In the ideal case where all the targets are aligned, there are no possible obstructions. Furthermore, as the error is given by λ_2, a threshold can be defined to accept or reject this approximation.

2. $\lambda_1 \simeq \lambda_2$: This situation is much different, as there is no axis to privilege over the other. Dropping either axis would mean an equal loss of information.
3. λ_1 or $\lambda_2 > 2 \cdot \text{field depth} \cdot tan(\frac{\beta}{2})$ is a problematic case illustrated by the figure 4.6. The field opening is smaller than the spread, thus the camera is physically not able to track all the targets.

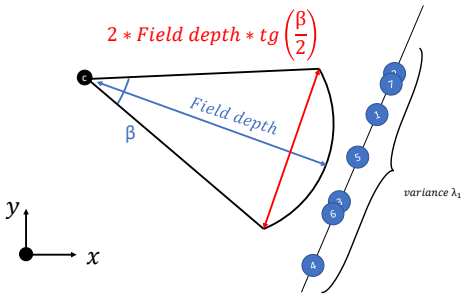


Figure 4.6: PCA - particular case

In cases 2 and 3 there is not a clear best solution. Therefore, another criterion is required to determine which targets should be tracked (such as a priority level as explain in section 4.5.1). Another possibility in a system where cameras can move is to initiate communication to receive support.

4.3.4 Application to simulated cases and limitations

PCA analysis requires at least two targets in the field of vision. Otherwise it is not possible to use this method. However there is no theoretical upper bound on the number of targets it can handle. Results are obtained with the python library scikit-sklearn [4].

For a given set of positions, the PCA method will output two directions, allowing to determine the orientation of the camera. Then by defining a distance from the origin along the axis with the smaller variance, the candidate positions are restricted to two points. Subsequently, a last criterion is needed to chose among them. In the implemented system, the closest point to the current position of the camera is chosen.

4.3.4.1 Two targets

For two targets, $\lambda_1 >>> \lambda_2 \simeq 0$ always holds. The result is graphically represented in the graph 4.7, where vectors show the principal directions ($\boldsymbol{a_1}$ and $\boldsymbol{a_2}$ in figure 4.5) and their length is proportional to the variance along the corresponding axis (λ_1 and λ_2).

Obstructions occur when the camera and the two targets are aligned. Therefore placing the camera perpendicular to the direction of alignment of the two targets centers ensures an obstruction free configuration.

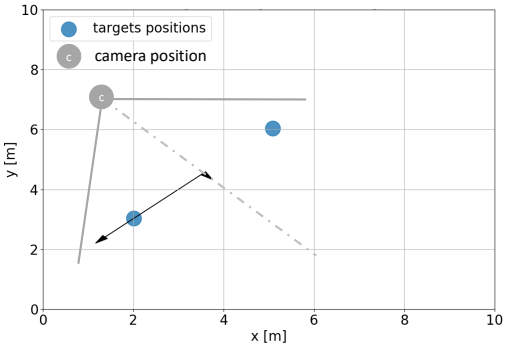


Figure 4.7: PCA - applied on two targets

However a failure might occur in the particular situation illustrated in the figures 4.6 and 4.8 (explained in section 4.3.3.1). A solution in this case is to place the camera nearly parallel to the target alignment in order to benefit from the field depth. This is useful only if the field depth is longer than the distance between the two targets. This is represented in the figure 4.8. The drawback of this method is that the target closest to the camera will obstruct a large portion of its field of view, as it is closer to it than usual.

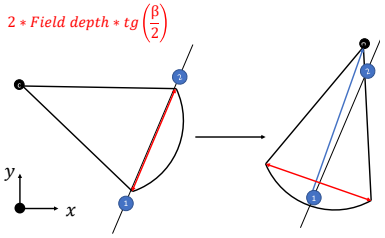


Figure 4.8: Two widely spread targets

4.3.4.2 Three targets

With three targets, the probability that an obstruction occurs increases. However, it is still possible to find an obstruction-free configuration for every arrangement of targets in the room.

To guarantee a view on each of the targets, the final condition to chose between the two points resulting from the PCA method is redefined. The two targets furthest apart are selected and the plane is divided in two from the line connecting them. Placing the *"agent-camera"* on the plane that doesn't contain the third target will then be chosen.

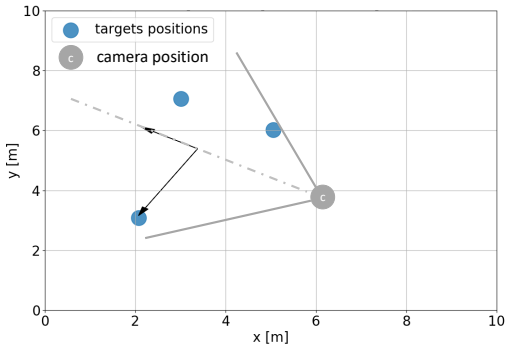


Figure 4.9: PCA applied on three targets

By placing the third target further away from the camera, it will appear smaller. Thus the field left to observe the room is higher. It is possible to track it on a larger area and it appear to moves slower.

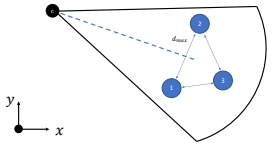


Figure 4.10: Schematic camera positioning

4.3.4.3 N targets

Finding a suitable configuration becomes more complicated with an increasing number of targets. However PCA analysis still works and returns two axis along with their variance. It is therefore still used to maximize number of targets seen. Furthermore depending on the values of the variances, the camera might not be able to track all targets on its own. A reasonable limit with this method is three targets as shown in the section above.

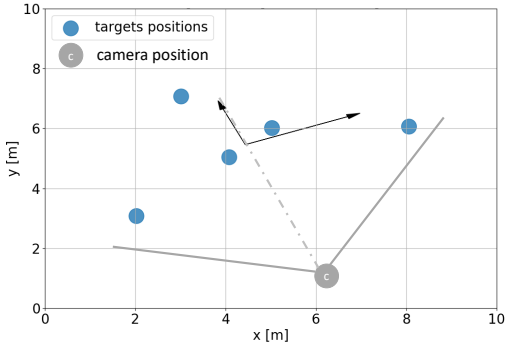


Figure 4.11: PCA - applied on five targets

4.3.5 Avoiding similar views

When two agents are too close, there is a risk that both of them have the same view of the room. In this case it means that an obstruction could not be compensated for, as the view are not complementary. In order to avoid that, the agents check whether they are too close to each other with the same orientations. If they are, then one of the agents will initiate a default behaviour, as defined in section 4.3.1.

4.4 Configuration validation

Once a configuration has been found, it needs to be validated before the agents move towards it. This validation step is important as the configuration returned using the methods described in the previous section does not come with the guarantee that the targets are all unobstructed. Furthermore, it might be preferable to track less targets, with a "clearer" view on them, rather than poorly tracking more targets. For these reasons, two validation criteria have been defined.

The first step of the validation is to check whether all targets are visible. Thus, the configuration is considered invalid if at least one target is obstructed (in the sense of algorithm in appendix F.1).

The next validation step concerns the configuration score - as defined in section 4.4.1. It is a possible quantitative evaluation of how "clearly" the configuration allows the targets to be seen. A comparison to a threshold score determines whether the configuration is valid.

4.4.1 Configuration score

The potential field method (appendix D) can be adapted to compare configurations. A score is computed using a *"heat map"*, which attributes a value to every possible position in the room. This way, it is possible to associate a score to each measured (or filtered) target's position. The *"configuration score"* is the sum of the separate score attributed to each target.

4.4.1.1 *"Heat map"* Definition

The *"heat maps"* are mathematically defined as:

$$U^n_{\text{heat map}}(\boldsymbol{x}) = \begin{cases} U^1_{\text{basis},i}(\boldsymbol{x}) & \text{if } n = 1 \\ \max(U^{n-1}_{\text{heat map}_i}(\boldsymbol{x}), U^1_{\text{basis}}(\boldsymbol{x})) & \text{if } n > 1 \\ \max(U^{n-m}_{\text{heat map}_i}(\boldsymbol{x}), U^m_{\text{heat map}_i}(\boldsymbol{x})) & \text{if } n > m \end{cases} \tag{4.4}$$

where $n > m > 0$ is the order of the heat map and represents the number of basis functions composing it. $U^1_{\text{basis}}(\boldsymbol{x})$ represents a basis function of order 1 that covers a single region of interest. It is worth mentioning that $U^n_{heat-map}(\boldsymbol{x}) \in [0, 1]$ to get a normalized value. Furthermore, it can be noticed that a heat map is created from the composition of lower order functions.

Potential basis functions examples

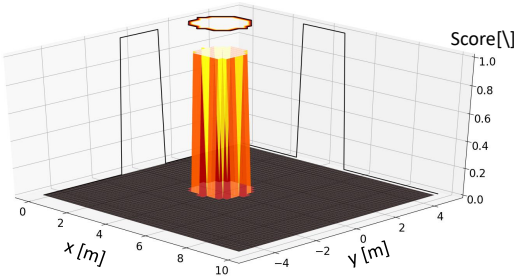


Figure 4.12: $U^1_{\text{basic step}}$

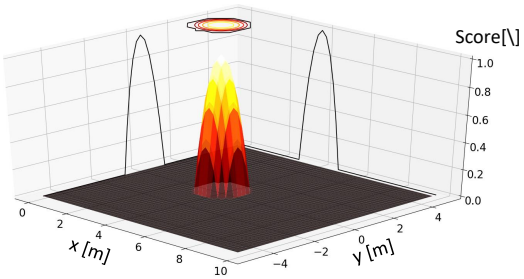


Figure 4.13: $U^1_{\text{basic quadratic}}$

$$U^1_{\text{basic step}}(\boldsymbol{x}) = \begin{cases} \xi & \text{if } \rho(\boldsymbol{x}) < \rho_0 \\ 0 & \text{elsewhere} \end{cases} \tag{4.5}$$

$$U^1_{\text{basic quadratic}}(\boldsymbol{x}) = \begin{cases} \frac{\xi}{\rho_0^2}(\rho_0^2 - \rho(\boldsymbol{x})^2) & \text{if } \rho(\boldsymbol{x}) < \rho_0 \\ 0 & \text{elsewhere} \end{cases} \tag{4.6}$$

ξ is constant to modify the potential amplitude $\in [0, \xi]$ and ρ_0 is the distance of influence

The position of a target is compared against a predefined position, which is the center of the basis function. As shown on the figure 4.14, the score associated to a target's position is bounded between [0,1] (for a factor $\xi = 1$). A score of one means that the estimate matches perfectly with the awaited position. The score then decreases as the difference in positions increase. Therefore, a *"heat map"* can be defined in order to evaluate the positioning of a camera for a given list of target. The maximisation of this score leading to the optimal configuration.

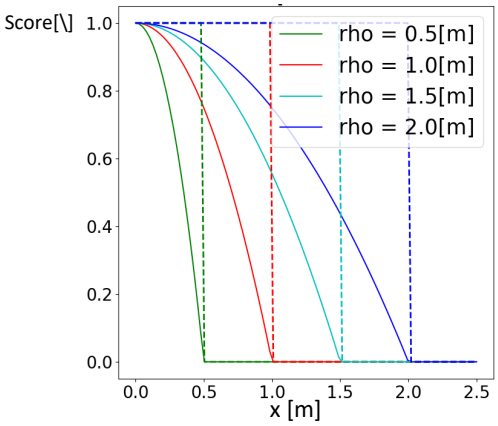


Figure 4.14: Basis function in terms ρ_0

The tolerance on the position can be tweaked with ρ_0 as shown in the figure 4.14. For a given position, a larger ρ_0 will correspond to a higher score. In the figure 4.14, the plain lines correspond to equation (4.5) and the doted ones to equation (4.6)

Note : Basis functions are in this example centered around $(0, 0)$. A coordinate change on $\rho(\boldsymbol{x})$ (domain of application) is required to move and orient it anywhere in the 2-D plane.

4.4.1.2 Computation of configuration score

The *"heat maps"* can be used for the computation of a configuration score, given a number of target positions. The *"agent-camera"* seeing the targets is assumed to be in the origin of the *"heat map"*, with its bisector parallel to the x-axis. The *"heat-map"* should then be defined as non-zero only in the regions inside the field of view of the camera.

The computation of a configuration score is now mathematically formulated. Given a list of targets positions:

$$\boldsymbol{X} = [(x_1^i, y_1^i), (x_2^i, y_2^i), ..., (x_n^i - 1, y_n^i - 1), (x_n^i, y_n^i)] \tag{4.7}$$

The *"configuration score"* is computed with the following formula:

$$\text{Score}_i = \frac{1}{n} \sum_{j=1}^{n} U^n_{\text{heat map}}(\boldsymbol{x}^i) \tag{4.8}$$

The problem with the first evaluation method is that multiple targets can be associated to the same basis function. This could also mean that some basis functions are not associated to any target. In that case, the computed score will be higher than the threshold, even tough the pattern doesn't correspond to the *"heat map"*. One solution is to remove the basis function each time a target is associated to it. The evaluation method then becomes:

$$\text{Score}_i = \frac{1}{n} \sum_{p=0}^{n-1} \sum_{j=1}^{n-p} U^{n-p}_{\text{heat map}}(\boldsymbol{x}^i) \tag{4.9}$$

$$U^{n-p}_{\text{heat map}}(\boldsymbol{x}^i) = \begin{cases} U^{n-p}_{\text{heat map}}(\boldsymbol{x}^i) & \text{if } U^{n-p}_{\text{heat map}}(\boldsymbol{x}^i) = \max(U^{n-p}_{\text{heat map}}(\boldsymbol{X})) \\ 0 & \text{else} \end{cases} \tag{4.10}$$

4.4.1.3 *"Heat map"* examples

"Unitary field" In the figure 4.15, the region $\rho(\boldsymbol{x})$ is the camera field itself. A target brings a score of 1 if it is inside and 0 otherwise. This is a simple map and it detect whether a target is inside the field of view. Thus it could replace the second point of the algorithm presented in the section F.1.

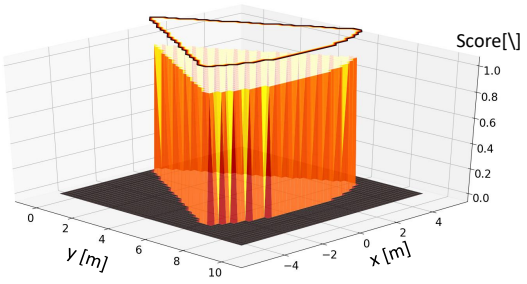


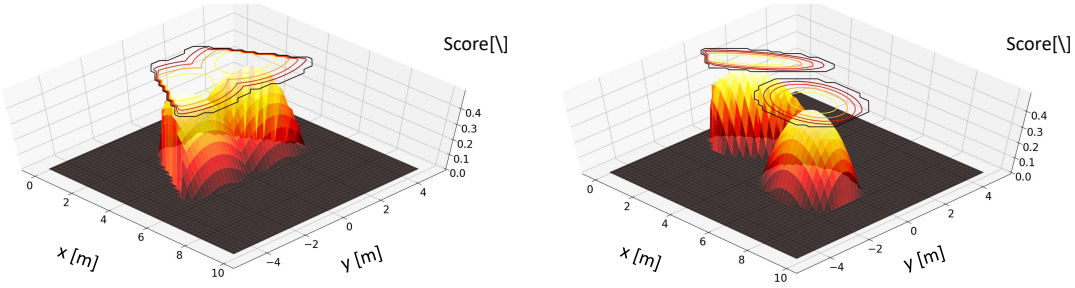
Figure 4.15: *"Unitary heat map"*

$$U^1_{\text{unitary field}}(\boldsymbol{x}) = \begin{cases} 1 & \text{if inside the field of vision} \\ 0 & \text{elsewhere} \end{cases} \tag{4.11}$$

Using this *"heat map"*, the best camera configuration is the one that follows the biggest number of targets without taking into account the way they are placed.

"One target in the middle" The case where there is only one target to track is now examined. A possible approach is to position the camera at a given distance from the target. A score of 1 means that the camera is perfectly placed, a score of 0.95 means that the camera should move slightly to get the wanted configuration. A score of 0 means that the configuration is not good at all. Corresponding *"Heat maps"* are shown in the figures 4.12 and 4.13.

"Two targets in the middle"* or *"Two targets side-way" Two scenarios with two targets are illustrated with the figures 4.16a and 4.16b. The first will result in a higher configuration score when the two targets are close in the middle of the field of view. On the second map, the score will be higher when the targets are a bit further apart. These maps offers the possibility to easily identify certain situations, allowing the *"agent-camera"* to react appropriately.



(a) Two targets centered (b) Two targets side-ways

Figure 4.16

4.4.1.4 Additional notes

The score can be use to compare different scenarios and pick the more appropriate one, similar to a pattern-matching.

A big advantage of the *"heat maps"* is the modularity and the scalability they provide. A *"heat map"* can be created for an unlimited number of targets and every possible arrangement. However, dedicated *"heat maps"* need to be created for each possible scenario which is time consuming. Also, the tolerance factor (ρ_0) is difficult to define, as it is a compromise between the probability to observe a particular arrangement and the confidence in the observation. A smaller variance increases how well a scenario can be identified, however the probability of observing this particular situation decreases.

Applying the *"heat map's"* definition, figure 4.17 shows a function that evaluates the position of three targets. It is built by composing the *"heat map"* from figure 4.13 with the one from figure 4.16b.

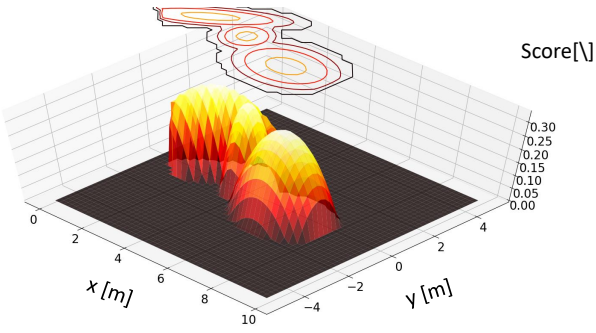


Figure 4.17: *"Heat map"* composition: Two target side-ways and $U^1_{\text{basic quadratic}}$

4.5 Constraints relaxation

Each of the targets in the field of view of the agent constitutes a constraint on the possible configurations the agent can adopt. When a valid configuration for the current list of targets to track cannot be found (as explained in section 4.4), one of the targets is dropped from this list, and the agent calculates a configuration from this new list of targets to track. This means that the dropped target is not followed by the agent anymore.

The agent could at this point inform the others of the loss and a recovery strategy could be initiated. A strategy allowing to chose which of the targets of the set should be dropped is therefore necessary. Especially in a multi-agent network of cameras, where the goal is to keep track of as many targets as possible, multiple aspects can be considered. The target priority is now described, followed by the implemented strategy adopted when a target is dropped.

4.5.1 Target priority

Different types of priorities are defined. The final target priority is then a weighted sum of these priority types.

A first criterion defined is a *"user priority"*. It set on a case-by-case basis by the user of the software, according to the importance given to a specific type of target. For example, if the system is used in an airport, the user priority of a person holding a firearm would be set to a very high value in order not to be lost, whereas an old lady would likely have a lower user priority. This type of priority is denoted by the symbol **P_user**.

A second criterion defined is an *"internal priority"* attributed to each target by the *"agent-camera"*. This binary internal priority is set to *"Low"* when a target enters the field of view of the agent. It is set to *"High"* when a target was already in the field of view in the previous cycle. This type of priority prioritizes targets already tracked by the agent, over targets that just came into its field of view. This priority is denoted by the symbol **P_internal**.

A third criterion is defined in order to reflect the quality of the current view of each target. If a target is seen "clearly", then it might be desirable to keep this target in the field of view. A metric that quantifies the view on a target is defined in section 4.4.1: the configuration score. It was used in the validation process of

a configuration, given a set of targets. In this case, however, the score is used to evaluate the position of a target, given the current configuration of the agent. This way, a target resulting in a high configuration score for the current configuration is considered as seen "clearly". This priority is denoted by the symbol **P_score**.

Taking these criteria into account, the final priority of a target is a weighted sum of the values of the three criteria:

$$\mathbf{P\_target} = U * \mathbf{P\_user} + I * \mathbf{P\_internal} + S * \mathbf{P\_score},$$

where **P_target** is the resulting priority for a target and U, I and S, are variables that can be tuned according to the use case.

4.5.2 Behaviour when target dropped

Initially, when an agent was about to drop a target, it communicated with the agents seeing it as well. Then, the sender would wait for answers, not being able to make progress. Either another agent ensured the tracking, or the target would be lost. In the second case, the agent would attempt to drop a different target, starting the same communications.

However, waiting for the responses is time consuming and creates delays. By the time of an exchange, the target could have disappeared potentially causing the failure of the system. Moreover, the agents who receive the message could communicate between them, in order to avoid all of them tracking the same target. In this case, communication costs become increasingly heavy, and the idea was abandoned.

In the implemented system, in order to remove delays, the *"agent-cameras"* always track the targets entering their field of view and set the target's internal priority to *"Low"*. Moreover, when a target starts being tracked, the *"agent-cameras"* broadcast a *"tracking-target"* message (see appendix B.1). Similarly, when a target is lost, the agent broadcasts the loss the others, by sending a *"losing-target"* message. This way, all *"agent-cameras"* know at each moment how many times each target is tracked (including the targets not tracked by anyone). Using this information, many different algorithms could be defined without needing to exchange messages when the agent decides which target to drop.

4.6 Movement towards a configuration

The controller regulates the degree of freedom of an *"agent-camera"*. It takes as input a configuration, which sets the desirable position, orientation and zoom to reach (DOF described in 4.2). The controllers ensures a smooth movement targeting always a better tracking. The first version was a proportional controller (often called P-controller) and is not explained in detail in this report. A second controller was developed as neither the configuration computation nor the P-controllers take into account obstructions occurring over the displacement. The new controller uses the potential fields method, explained in appendix D.

Figure 4.18 illustrates a situation with three targets (red dots), an ideal configuration (green dot) and two cameras (blue dots). The configuration was accepted by the validation process (see section 4.4) and the controller moves the cameras towards their optimal configuration. However an obstruction occurs when the camera (at the bottom left) crosses the line joining the two targets (red lines in the figure 4.18). An extension of the P-controller was developed to prevent such situations from happening, forcing the agents to stay away from obstructed regions. Obstructed regions are modelled as lines connecting two targets and the controller ensures that the agent is repelled when it comes too close. In figure 4.18, the colored vectors show the direction of the repulsion vector resulting from the new controller.

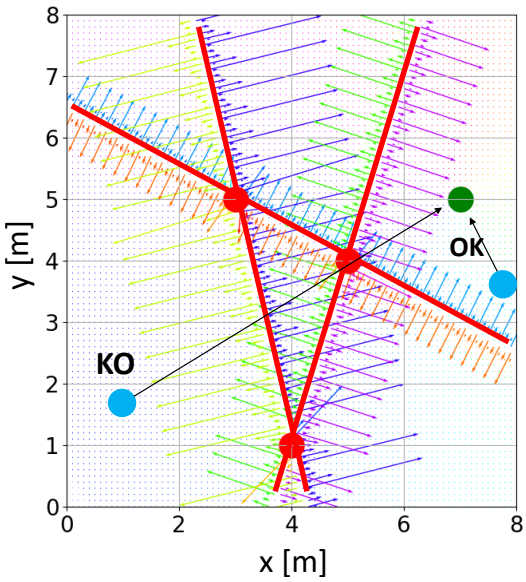


Figure 4.18: Obstruction simplified model

4.6.1 Obstruction avoidance with potential field method

An obstruction occurs when a target partially or totally hides the field of view of the camera. In figure 4.19, the field (green zone) is obstructed by the target_1, thus the red zone is not under surveillance of the camera (red dot). Target_2 is not tracked in this configuration. The goal is to find a better configuration, allowing both targets to be seen. To find such a configuration, the repulsive potential should push the camera away from the configuration (x_{c-KO}, x_{c-KO}) (red dot), up to some point (x_{c-OK}, y_{c-OK}) (green dot), where both targets are

tracked. To achieve that, a region inside of which the camera cannot penetrate is defined, using an appropriately placed repulsive potential. The boundary of the repulsive potential forcing the camera to stay outside the region that contains all the undesired configurations is highlighted by the yellow doted line in the figure 4.19.

The first step to find an appropriate potential is to define a point representing the center of the potential function. The yellow point in figure 4.19 is a relevant choice: it is the mean of the target positions. Therefore, it will always take into account both targets. Using a circular shape, as proposed in the literature, is not optimal as it reduces the configuration space too much. For instance, in figure 4.19, there exist a set of configurations where both targets could be seen, but are forbidden, considering the camera cannot enter inside the yellow circle.

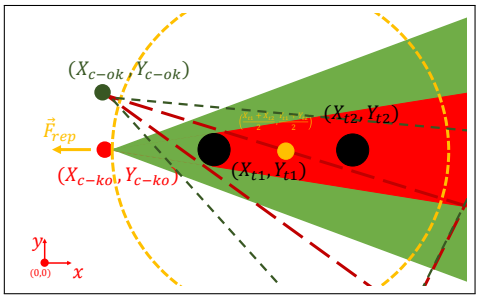


Figure 4.19: Circular potential

Figure 4.20 shows the repulsive forces resulting from the potential field in circular mode. One potential source is added in the positions of each target. The potential source from figure 4.19 is also present. In the right plot, black arrows outline the direction of the resulting forces, also drawn in color with more detail. Figures 4.22 and 4.24 are constructed in a similar fashion.

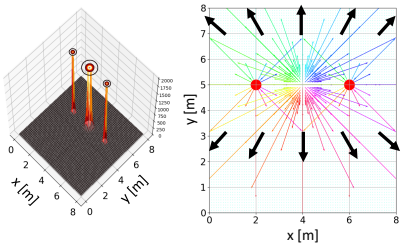


Figure 4.20: *"Circular mode"*

For the purposes of the system, is it more appropriate to deform the circle into an elliptical shape, whose longest axis is parallel to the target alignment. This way, the repulsion is focused on a smaller region, corresponding only to the unwanted configurations.

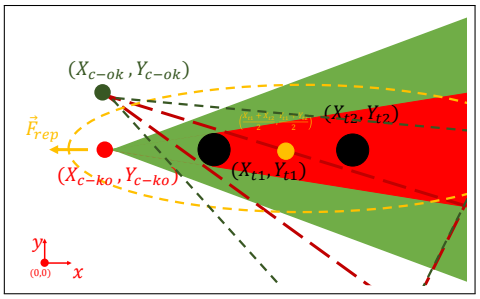


Figure 4.21: Ellipsoid potential

In this case the distance to the center of the potential is computed as follows:

$$\rho(\boldsymbol{x}) = \rho_{\text{ellipsoid}}(\boldsymbol{x}) = \sqrt{\alpha_x \; x^2 + \alpha_y \; y^2}, \tag{4.12}$$

where α_x and α_y are factors needed to stretch the circular potential along the x and y axis.

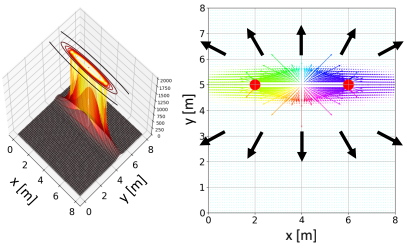


Figure 4.22: *"Ellipsoid mode"*

To get the quickest misalignment of the camera with respect to the targets, the applied force should be perpendicular (as shown in the figure 4.24) to the line connecting the targets. The force direction being defined by $\nabla\rho(\boldsymbol{x})$ and assuming the x-axis is parallel to the targets alignment, the following formula is appropriate:

$$\rho(\boldsymbol{x}) = \rho_{linear}(\boldsymbol{x}) = y \tag{4.13}$$

The potential creates a barrier along the line connecting the targets, with a repulsive force perpendicular to it (see $\overrightarrow{F}_{rep}$ in figure 4.24).

However there is one problem left: the field does not result in a misalignment between the camera and the targets, as the repulsive potential field is purely radial with its center between the two targets. Figure 4.21 illustrates this problem (see direction of $\overrightarrow{F}_{rep}$ being the repulsive force acting on the camera).

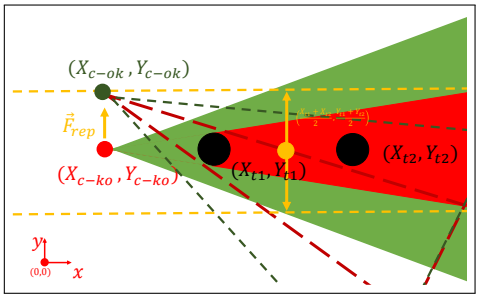


Figure 4.23: Linear potential

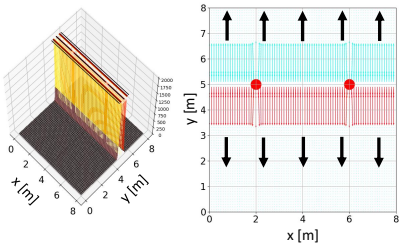


Figure 4.24: *"Linear mode"*

One problem with the linear mode is that it creates an infinitely long barrier, preventing the camera from ever crossing it. It would be desirable, however, that the camera crosses this barrier, if for any reason, it is necessary. To achieve this, the elliptical and linear modes are combined using the superposition principle. This is done by adjusting the parameter $\alpha_{\text{combine}} \in [0, 1]$. In this case, the resulting field is given by:

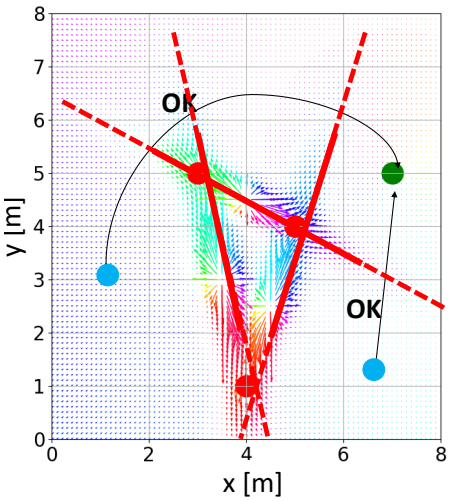


Figure 4.25: Obstruction simplified model

$$U_{\text{combine}}(\boldsymbol{x}) = \gamma(1 - \alpha_{\text{combine}}(t))U_{\text{elliptical}}(\boldsymbol{x}) + \alpha_{\text{combine}}(t)U_{\text{linear}}(\boldsymbol{x}) \tag{4.14}$$

An additional effect of this approach is to slightly modify the direction of the repulsive force such that it is not completely perpendicular to the line connecting the targets. This way, the camera can be pushed away or attracted towards the middle of the two targets. The repulsive effect is obtain using $\gamma = 1$ and the attractive effect by setting $\gamma = -1$

Figure 4.26 shows the resulting repulsive field. It tends to repel the camera away from the point centered between the two targets. Black arrows in the right plot qualitatively show the direction of the forces (color is associated to it).

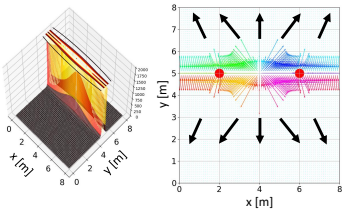


Figure 4.26: *"combined repulsive mode"*

Figure 4.27 presents the resulting attractive field. In this case the linear field has to be set high enough ($\alpha_{combine} \leq 0.4$) to prevent the camera from ending up on the point centered between the two targets (the elliptical potential act as attractive potential due to $\gamma = -1$).

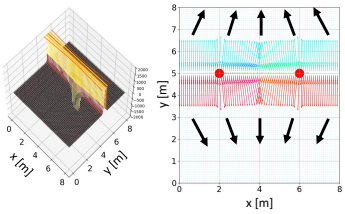


Figure 4.27: *"combined attractive mode"*

4.6.2 Results

This section evaluates several controller modes (shape of the potential field) described in the previous section. As depicted in figure 4.28, the case-study is the following: one target is fix in the middle of the room, while another one moves around it (Links to data an plots available in appendix H).

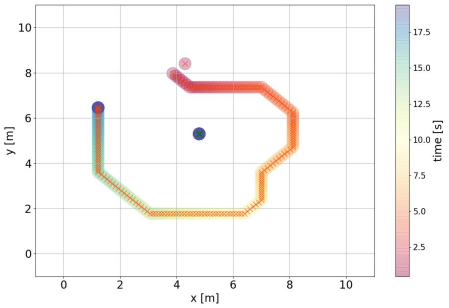


Figure 4.28: target's trajectories

The first possibility consists in moving towards a configuration as computed in section section 4.3. The resulting trajectory of the *"agent-camera"* is illustrated in the figures 4.30 and 4.31. The computed configuration is represented by a star, while the measured configuration is denoted by a diamond.

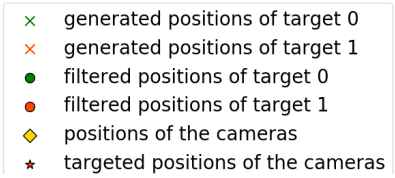


Figure 4.29: legend for the figures 4.30, 4.33 and 4.34

Figure 4.30 shows that the *"agent-camera"* follows a circular trajectory, as it turns around to keep tracking both targets. The variation of each degree of freedom (except β) of the *"agent-camera"* is illustrated in figure 4.31. One can see that each DOF is closely following the computed configuration, showing its efficient control.

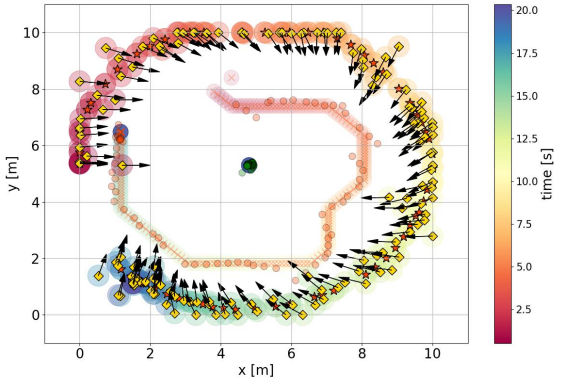


Figure 4.30: *"agent-camera"* following the computed configuration

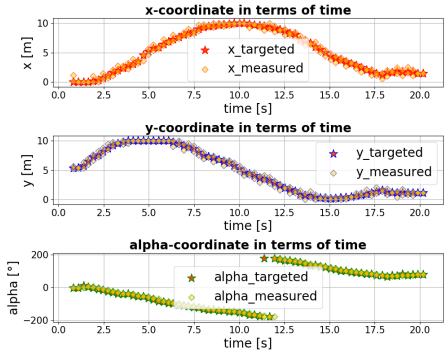


Figure 4.31: Comparaison between the targeted and measured *"agent-camera's"* DOF in terms of times

Finally, figure 4.32 shows the command associated to each DOF. They are included in $[-100\%, 100\%]$, 100% being the maximum speed achievable. Therefore, the figure shows that the gains are chosen to avoid saturation as the commands mostly stay in a range of $[-50\%, 50\%]$.

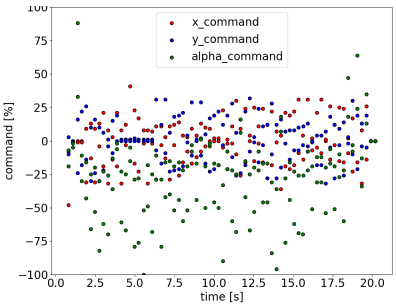


Figure 4.32: controller's command for each DOF in terms of time

In the next modes, ξ is set to 0. This means that the computed configuration does not act as an attractive potential, therefore not influencing the controller's output. On the contrary, the *"agent-camera"* only reacts to repulsive potentials (ie target's positions) .

In the figure 4.33, the *"agent-camera"* is repelled when it is aligned with the two targets. This is the case because the controller is set in the *"linear mode"*. The resulting force is then perpendicular to the alignment of the targets as shown in the figure 4.23.

In the figure 4.34, the *"agent-camera"* is partially attracted towards a point located between the two targets. For comparison purposes, the positions resulting from the PCA method are also shown on the figure (stars). While the distance to the targets is bigger when using PCA, the general shape of the trajectory is similar. However, this distance can be tweaked using the parameter η.

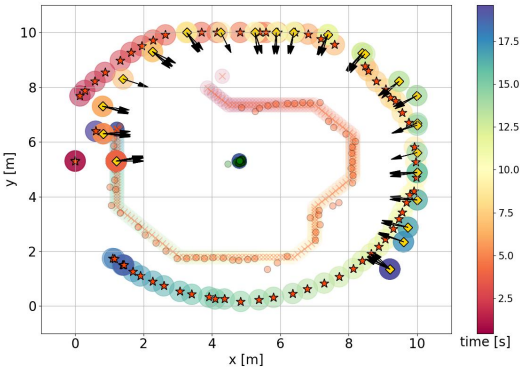


Figure 4.33: $\xi = 0$ and *"linear mode"*

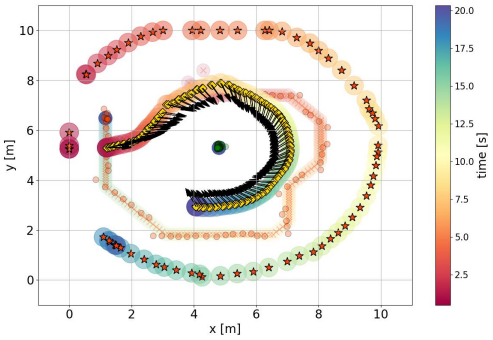


Figure 4.34: $\xi = 0$ and *"combined attractive mode"*

To sum up, this controller allows to move the *"agent-cameras"* when an obstruction occurs, without needing to compute a suitable configuration. Furthermore, the potential field method can also be used to reach a computed configuration.

4.7 Results and conclusion

In this chapter, allowing the *"agent-cameras"* to move was considered, aiming to improve the tracking quality. The movement strategy of an *"agent-camera"*, given a set of targets to be tracked, can be decomposed in the following steps. First, a goal configuration is computed, attempting to provide a "clear" view on every target. The goal configuration then undergoes a validation step. It is validated if non of the targets are obstructed and the *"configuration-score"* is higher than a threshold. If the configuration is not validated, the procedure starts over, after removing a target from the targets to be tracked by the *"agent-camera"*. Otherwise, the goal configuration is given to the controllers, resulting in the movement of the *"agent-camera"* towards the goal configuration. (Links to data, plots and video of the execution of the use-case are available in appendix H)

A comparison with the case-study presented in section 2.8 is made, to evaluate the impact of the *"agent-amera"* movement on the tracking quality.

The scenario represented in figure 4.35 is the same as the one of section 2.8 (same target trajectory). However, the *"agent-cameras"* are, in this case, of different types: the two bottom ones are *"rotative"*, while the the top one is *"free"*.

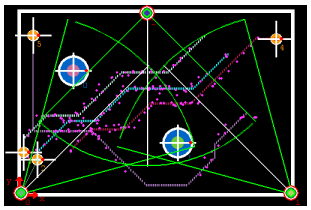


Figure 4.35: Use-case as represented in the *"simulation tab"*
link to video available in appendix H

Before comparing the results, figures 4.36 and 4.37 show the coverage of the room at the beginning and the end of the execution. It is clear in the second figure that the agents moved to keep tracking the targets.

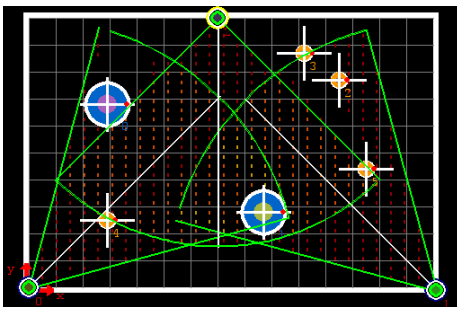


Figure 4.36: Coverage of the room at the beginning

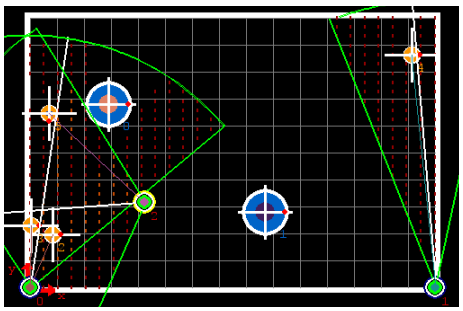


Figure 4.37: Coverage of the room at the end

Additionally, figure 4.38 shows the *"agent-camera's"* trajectories at the end of the simulation. Yellow diamonds are the successive positions reached, while the black arrows is the associated orientation.

The *"free"* camera starts at the top and moves all the way down to the bottom while the *"rotative"* cameras rotate around themselves. The plotted arrows show the different orientations adopted by the *"agent-camera"*.

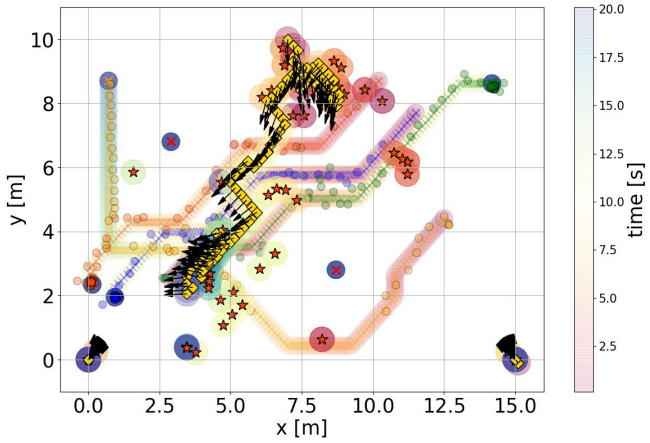


Figure 4.38: *"Agent-camera"* 's trajectories in terms of time

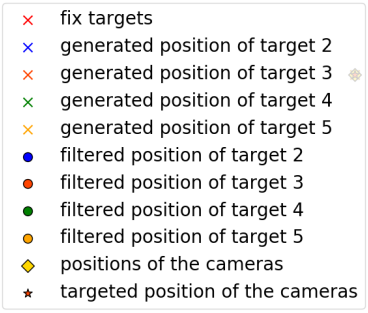


Figure 4.39: legend of figure 4.38, 4.42, 4.43

A first comparison is made between the *"target-estimations"* received by the *"agent-user"* in the two scenarios (figures 4.41 and 4.40). The figures show that *"agent-cameras"* are able to track the targets for longer periods of time. Indeed, none of the targets are lost in the case of moving target.

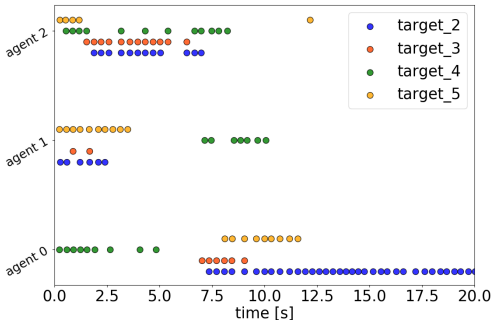


Figure 4.40: *"Target-estimation"* received by the *"agent-user"* from the fix *"agent-cameras"* (section 2.8)

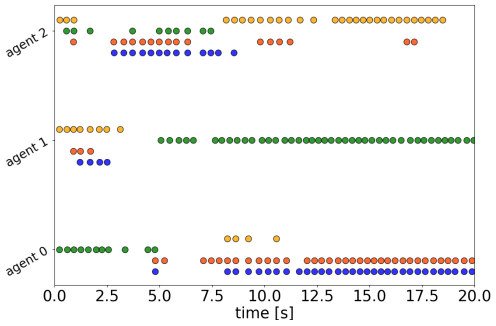


Figure 4.41: *"Target-estimation"* received by the *"agent-user"* from the moving *"agent-cameras"*

A second comparison is made between the reconstructed trajectories in the two scenarios.

A first observation concerns the trajectories as reconstructed by the *"agent-user"* (figures 4.42 and 4.43). For this use-case, it appears clearly that the quality of the tracking is improved when the *"agent-cameras"* are moving: all targets are tracked for at least as long as in the fix *"agent-cameras"* case.

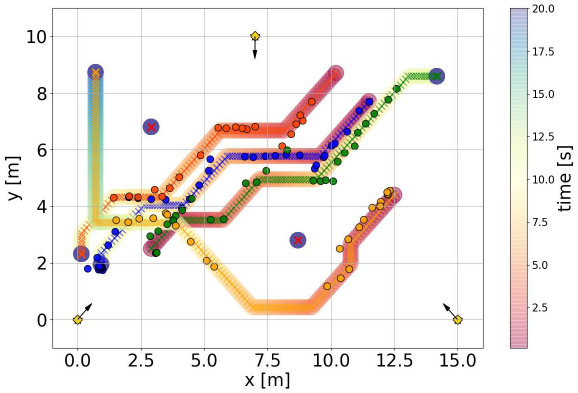


Figure 4.42: Tracked targets trajectories from the *"agent-user"* 's perspective (results from the section 2.8)

A second observation that can be made is the presence of the obstruction between 2.5 s and 7.5 s for the yellow target. This obstruction is unavoidable in both cases, as the *"agent-camera"* at the bottom right has to choose between tracking the yellow or the green target, which are moving in opposite directions. At this point, the user-defined priorities will determine which of the targets is tracked.

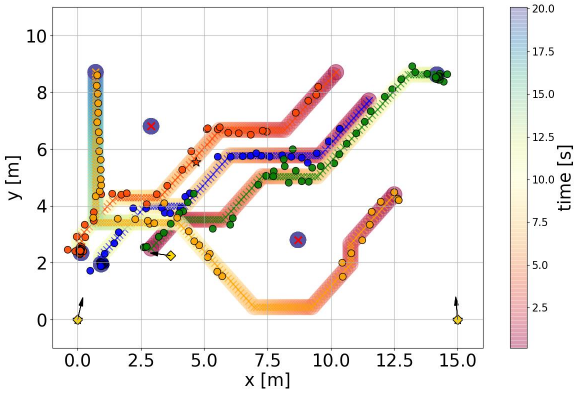


Figure 4.43: Tracked targets trajectories from the *"agent-user"* 's perspective

In the end, the movement of the agents increases the performance of the tracking for the use-case examined in this section. Of course, the movement algorithm should be further analyzed in different use-cases to draw decisive conclusions about its usefulness. Moreover, allowing the *"agent-cameras"* to move adds an important extra layer of complexity, both in the software and hardware layers. Therefore, the performance increase for the scenario in which it will be used needs to be considerable, to justify the additional effort required.

Chapter 5

Simulation

The simulation software is a crucial step in the research process. The modeling hypothesis for the cameras and targets are described in section 4.2. The software is implemented in python3 and its class diagram can be found in appendix G. The source code of the software can be found here: `https://github.com/xenakal/Simulation_Interactions`.

5.1 General architecture

Trying to make the simulation representative of the real system, a video stream processed by a detection algorithm would be the closest input. However, as mentioned in the introduction, this was not the focus of the research. To model this input, a *"room"* has been defined, which contains information on all simulated elements. These include the dimensions of the physical space as well as information on agents and targets (position, orientation and others). Agents can be activated/deactivated and targets can appear/disappear at any time.

Each agent (*"agent-cameras"* and *"agent-user"*) is running on its own separate process to model the parallel nature of the network in a real situation. An additional thread moves the targets in the *"room"* at regular time intervals to simulate the changes in the environment (independently from the agents).

"Agent-cameras" and the *"agent-user"* include the following main objects:

- **Mailbox and messages list**: a Mailbox ([5]) is a concept implemented in python 3 ensuring a safe communication between threads. It is the way agents communicate.
- **Memory**: Data structure used to hold information gathered and filtered by the agents.
- **room-representation**: The *"room-representation"* is a data structure representing the agent's beliefs on the simulated reality (the *"room"*). In other words, the *"room-representation"* contains the information about the environment gathered and filtered by the agents. Compared to the Memory, it does not have a notion of time: it represents the last memories gathered for each target.

Additional objects belong only to *"agent-cameras"*.

- **Camera** models a physical camera, able to take pictures of its virtual environment using algorithm in section F.1.
- **LinkTargetAgent**: responsible for the coordination between agents (see section 2.6).
- **Controller**: Along with a function called *"move_based_on_config"* (in the class *"agent_camera"*), it regulates the several degrees of freedom.

Moreover all simulated parameters correspond to real physical quantities (speeds, positions...).

The software also includes a graphical interface used for visualisation. However, it can also be run without it, for example when gathering results. Furthermore, after an execution of some scenario, a number of plots and log files are automatically generated.

5.2 Graphical User Interface

The software includes a GUI for visualizing the evolution of the elements. Its use is optional, as the logic of the software is defined separately. In all figures presented, the first row of buttons on top corresponds to different tabs of the software, whereas the second row corresponds to functionalities within the selected tab.

5.2.1 *"Simulation tab"*

The *"simulation tab"* (shown in the figure 5.1) provides a graphical visualization of the simulated situation

The white square in the middle represents the room's boundaries and the red arrows at the bottom left represent 1 m in the x and y directions. Elements (ie *"agent-cameras"* or targets) evolve within those limits. In the figure, the fix targets are represented with a blue outer circle, whereas the moving ones have an orange outer circle. Active *"agent-cameras"* are green, whereas inactive ones are red.

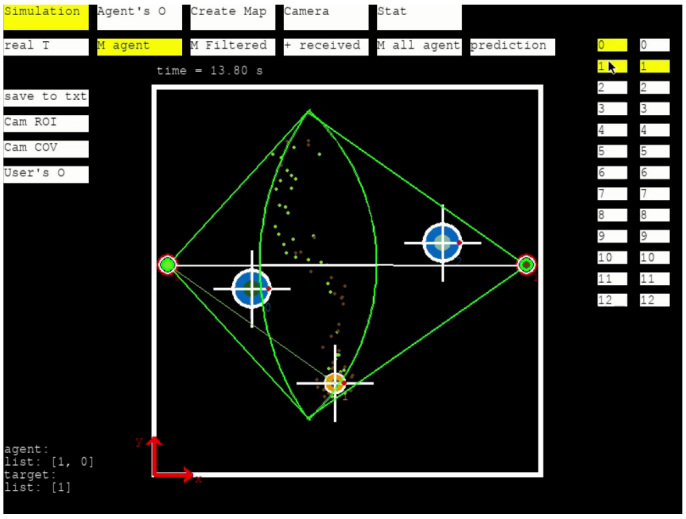


Figure 5.1: Simulation tab

There are several options available in this tab, which can be enabled using the top buttons (second row). Each of these buttons displays the trajectory of the selected targets in a different form. Agents and target can be selected using the right columns of numbered buttons (right column to select a target and left one to select an agent).

- `real T` shows the noiseless trajectory (simulated values).
- `M agent` displays the measurements taken by the selected agents.

- `M Filtered` displays the filtered positions. The FilterPy ([6]) library was used to create the filter.
- `prediction` shows the future positions as predicted by the selected agent.

In this tab, there are three additional options (the buttons to the left of the room):

- `Cam ROI` shows the regions associated to each agent (as explain in the section 2.6, the result is shown in the figure 2.12a).
- `Cam COV` shows the room coverage (as shown in figure 2.12b).
- `User's O` shows the system's output. It is the filtered positions of the targets being tracked, send by the *"agent-cameras"* to the *"agent-user"*.

5.2.2 *"Agent's output tab"*

The *"Agent's output tab"* (figure 5.2) displays the *"room-representation"* of the selected agents (selected by clicking a button on the right).

One graphical notation is different from the *"simulation tab"*: the *"confidence"* on a target's position. It is represented by the color of the outer circle of the target and it ranges from green to red. A green color corresponds to a high confidence whereas a red color to a low confidence (the confidence as defined in section 4.2).

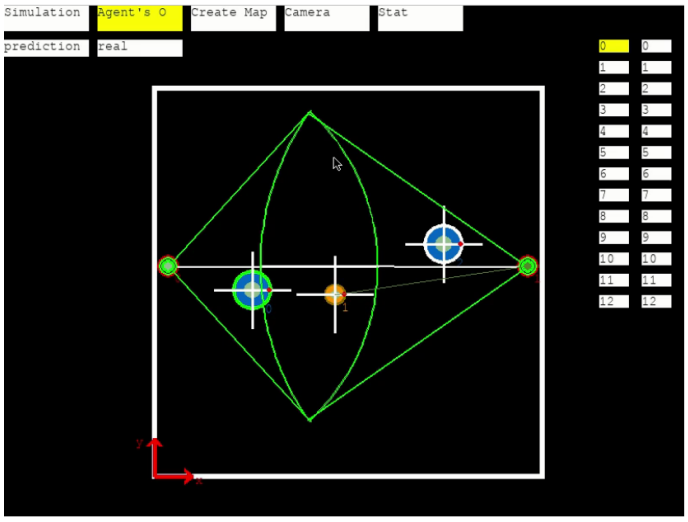


Figure 5.2: Agent's output tab

In this tab, an agent's *"room-representation"* can be compared to the *"room-representation"* of another agent or with the real positions of the targets by selecting the option `real`. This tab is very useful to understand the behaviour of an agent.

5.2.3 *"Map creation tab"*

The *"Map creation tab"* is an environment to create case-studies. A map can be created by placing elements in the room. When a certain type of element is selected, a set of options for the element appear in the left of the tab. Each of the options corresponds to an attribute of the element, that can be tweaked to preference. Finally, trajectories for targets or cameras can also be assigned in this tab.

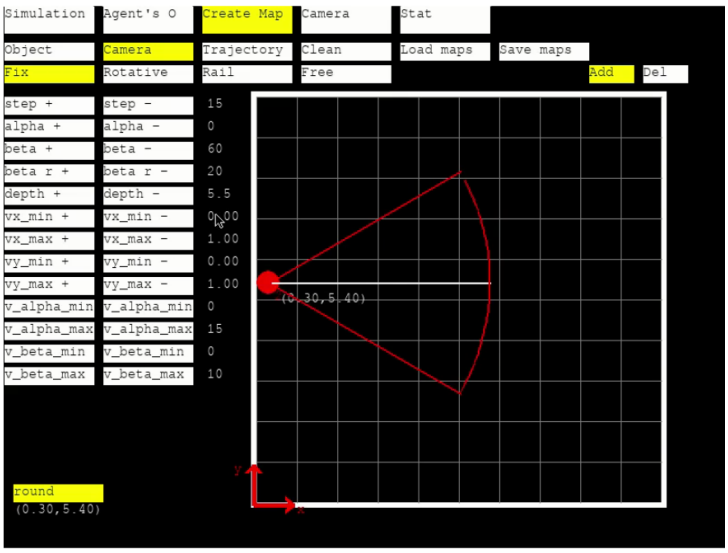


Figure 5.3: Map creation tab

5.3 Limitation of simulation

On one hand the modeling helps to describe reality in a simplified way because it forces to express problems with relevant hypothesis. On the other hand reality is more complex and cannot be fully modeled. Subsequently, the simulation is limited in several ways.

One limitation is that the multi-threaded architecture is limited by the machine on which the software is run, as all threads are executed on it. Even on a multicore machine, each agent is not continuously active. This will depend on the thread scheduling done by the operating system.

Another limitation becomes apparent when multiple agents (more than 4) are in the room. In this case, the scheduling effects start affecting the results. This happens because the time during which the threads are idle (ie. another thread is running) increases. In turn, this leads to the agents being out of date for a longer period of time, which impacts their synchronization. In order to evaluate a system at a larger scale, a true distributed system should be implemented, where the threads would run on separate machines (or cores).

Chapter 6

Conclusion

The fundamental purpose of this thesis was to develop a multi-agent system for robust distributed tracking in the presence of obstructions. A secondary topic of this thesis was the consideration of allowing the agents to move, thereby increasing their autonomy. A simulation was developed to evaluate the proposed architecture, focusing on the behaviour and cooperation of the agents. Additional sub-problems have been studied, represented in green in figure 6.1. Among those is the noise removal strategy, using a Distributed Kalman Filter and the obstruction avoidance algorithm. Each of these subjects has been addressed by scientific papers and has room for improvement.

To begin with, the DKF used dates back to 1993. Since then, many improved distributed filtering strategies have been developed. For example, [7] proposes a distributed Kalman filter where each node only communicates with its neighbors, thus reducing communications. Another example is [8], using a sensor-fusion algorithm based on the least-squares estimate of a vector (function of the measurements of all the agents) and the covariance of this fused least-squares estimate. This approach is robust to changes in the network topology, well suited for a scalable system.

Furthermore, the simulation offers the possibility to evaluate methods for calculating the optimal position of an agent, based on the positions of the targets in its field of view (and optionally the positions of the other agents). [9] is interested in the assignment of targets to sensors in order to minimize the expected error. [10] proposes a metric of a camera configuration (placement of cameras relative to the targets) taking resolution and occlusion characteristics into account. This approach, should then be used along with an optimization engine to search the state space of possible configurations. Furthermore, it could also be used as a substitute for the *"score"* as defined in section 4.4.1. Finally, for problems like these, machine learning is often effective and worth investigating. Such a solution is proposed by [11].

As stated in the introduction, the creation of a real multi-agent tracking system involves several steps that are not covered in this thesis. This section describes possible solutions for each of these steps.

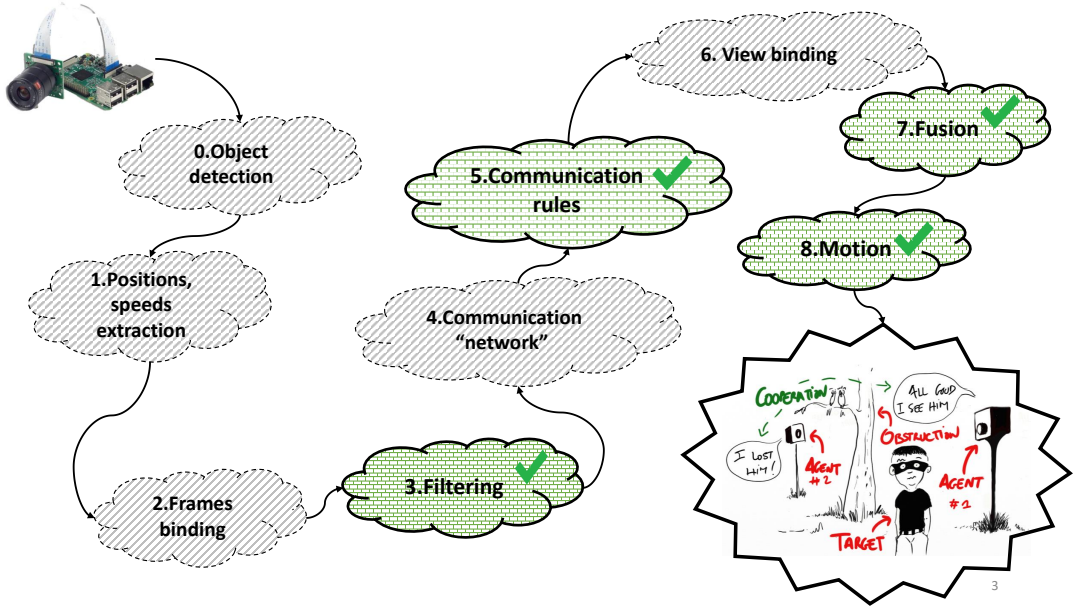


Figure 6.1: Steps before reaching the final goal

First, the object detection algorithm has to be implemented on the single board computer (**(0)** in figure 6.1). The challenge lies in that at least 10 frames per second should be reached as output of the object detection algorithm. Below this limit, real-time tracking is not an option. Moreover the frame rate determines the time between detectable obstructions. With this in mind, the single board computer and the choice of the object detection algorithm have to be considered together: depending on the hardware specifications of the single board computer (for example the brand of the GPU), an implementation of the detection algorithm gives varying results (see appendix A).

Then the input data being provided by images, the depth information is lost. A transformation from the image plane to the 3D coordinates corresponding to the position of the object in the room has to be performed (**(1)** in figure 6.1). This can be done, for example, as proposed by [12] or [13].

The two further remaining steps involve data association (**(2)** and **(6)** in figure 6.1). This is a classification problem, where the objective is to associate a new estimate of a target's position with the previous ones. If the agents need to exchange information about specific targets (for example to keep track of the targets not seen by any agent), data identification become necessary. Solutions to this problem have been proposed in the literature, amongst others by [14], who uses a set of key feature points to link a given object from frame to frame. One might consider using such a technique for data association between agents as well.

Finally, in a real system, it will be important to consider how the single board computers communicate with each other (**(5)** in figure 6.1). There are several ways to do so, such as communication via ssh over a local network or a direct cable connection. However, The method will depend on the single board computers used as well as the use case of the system.

Besides the remaining problems, this work focused on the basic conceptualization of a multi-agent network. Research on agent behaviour allowing to keep tracking targets in presence of obstructions was conducted trough a simulation. The latter being limited in some points, the need for a real implementation on single board computers has become apparent and constitutes the next step of the project.

Appendix A

Preliminary work

Before working on the simulation, some research was conducted to run the YOLO object detection algorithm on the raspberry PI.

The following data was obtained by running YOLO on the raspberry PI for a single picture :

- The complete version of YOLO V3 [15] took approximately 6 min, giving a confidence of 99% for the detection.
- The tiny version of YOLO V3 only took 30 s, however the confidence dropped to 50% and sometimes the detection failed.
- A nnpack version accelerator for YOLO V3 [16] allows to reach a time of 1.5 s, which starts getting close to an acceptable time for real-time tracking.

A solution to get a faster execution is to use GPU computation. YOLO V3 provides this possibility, using the CUDA parallel computing platform. However, this solution cannot be used with the Raspberry Pi, as CUDA is only compatible with an NVIDIA GPU (Raspberry Pi have a Broadcom GPU). Alternative implementations of YOLO exist that use OpenCL (for non NVIDIA GPUs) and are compatible with the raspberry Pi GPU with the tiny YOLO V3, such as [17]. Otherwise, the standard version of YOLO v3 with GPU computation can be used on other single board computers, such as the Jetson Nano. It can be noted that the latter provides better performances than the raspberry PI, lowering the time to 0.5 s per frame with the full version [18]. An implementation on a single board computer is therefore possible, but several different parameters have to be taken into account to achieve acceptable performance for online object detection.

Appendix B

Messages types

This appendix defines the different types of messages.

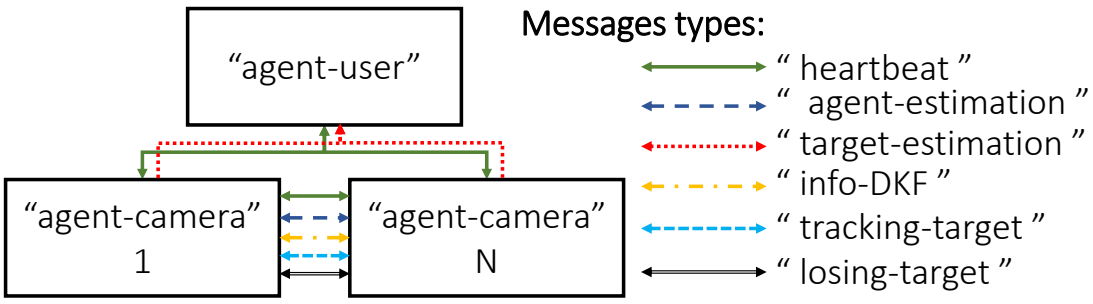


Figure B.1: Communication between agents

B.1 *"Heartbeats"*

This type of messages is the simplest yet essential. In the schematic B.1 (also in the B.2) they are represented by the green arrows. *"Heartbeats"* do not have a payload (ie. the message tag is sufficient information). An agent will send a *"Heartbeat"* to inform the others that it is alive. Therefore, they allow the dynamic adaptation of the agent topology, making the system more modular. This type of message should be sent at regular intervals, t, to all other agents. If the last received *"Heartbeat"* dates back to more than $C * t$, where C is an arbitrary positive integer, the sender will be considered as inactive by the other agents.

Figure B.2 summarizes a classical scenario. States are referenced by the symbol **(x)**, where x is the number of the state also on the schematics. At first, each agent knows about all other agents in the room, but it considers all of them as inactive **(2)**. Then, Agent1 sends a *"Heartbeat"* to agent2, who updates the status of Agent 1 to active **(3)** $\rightarrow$ **(4)**. Suddenly, Agent 1 stops working and therefore stop sending messages **(2)**. Agent 2 does not receive a heartbeat from Agent 1 for a duration of at least $3 * t$ and therefore updates the status of Agent 1 to inactive again **(4)** $\rightarrow$ **(3)**. Nevertheless, as soon as Agent 1 recovers and sends *"Heatbeats"* again, Agent 2 will update its status back to active and the situation is equivalent to one before Agent 1 failed.

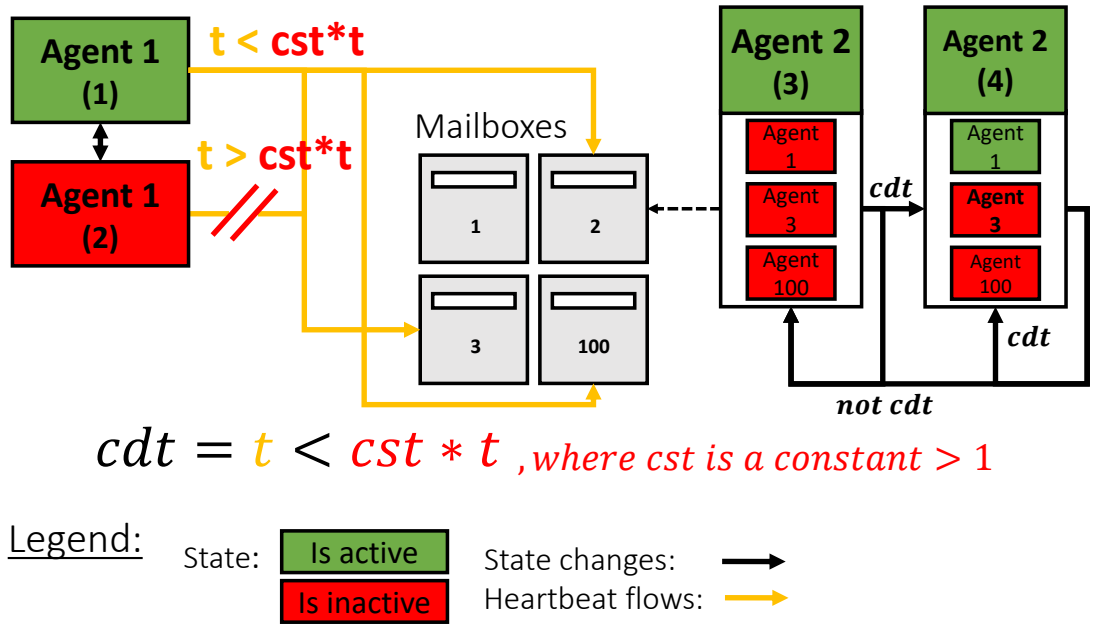


Figure B.2: *"Heartbeats"* principle

B.2 *"Target estimation"*

Agents exchange information about their *"room-representation"*. When *"agent-cameras"* need to inform the *"agent-user"* of any changes they notice in the room, *"target-estimation"* messages are sent. These types of messages contain information such as positions, speeds, sizes, confidence score. *"Agent-cameras"* usually do not exchange this type of messages between them, as they exchange dedicated filter messages (see section B.4).

B.3 *"Agent estimation"*

When agents move in space, their *"room-representation"* also needs to be updated to keep track of the last positions. The analysis run in *"Link Camera Target"* (see sections 2.6 and 2.7) uses the current position of every agent to draw conclusions. Therefore, when an agent moves, it needs to keep the others informed to ensure they are still synchronised on the same representation of the room. This task is performed by sending *"Agent estimations"* messages.

An *"agent-camera"* does not send this information to the *"agent-user"* as the objective is to track the target independently of the way the camera moves. Avoiding those exchanges prevents from useless communication and does not affect in any way the main goal.

B.4 *"Ditributed Kalman filter (DKF)"*

Agents cooperation aims at reducing the noise ratio of the data measured by the camera. For that purpose, a distributed Kalman filter was implemented, requiring the agents to exchange information based on their measurements. The filter and the required communication are detailed in section 3.2.

B.5 *"Tracking/Losing target"*

When a target comes or leaves the field of view of an agent, the agent broadcasts a tracking/losing target message to inform the others of it. This way, all agents know at any point in time how many agents in total are tracking each target. This can be useful in different situations. For example, if some agent doesn't see any target, it could start to search for a target that is not seen by anyone.

Appendix C

Kalman Filter

C.1 Theoretical reminder

This appendix constitutes a reminder of the Kalman Filter. It is assumed that the reader is already familiar with it.

The system is represented by a state vector $\boldsymbol{x}(k)$ at time k. It's dynamics are represented by the transition equation

$$\boldsymbol{x}(k) = \boldsymbol{F}(k)\boldsymbol{x}(k-1) + \boldsymbol{G}(k)\boldsymbol{w}(k), \tag{C.1}$$

where $\boldsymbol{F}(k)$ is the transition matrix of the model, $\boldsymbol{G}(k)$ is the noise model and $\boldsymbol{w}(k)$ is the process noise. The initial state $\boldsymbol{x}(O)$ is assumed to be Gaussian with known mean $\boldsymbol{x_0}$ and covariance $\boldsymbol{P_0}$. Furthermore, observations $\boldsymbol{z}(k)$ are acquired according to the equation

$$\boldsymbol{z}(k) = \boldsymbol{H}(k)\boldsymbol{x}(k) + \boldsymbol{v}(k), \tag{C.2}$$

where $\boldsymbol{H}(k)$ is the observation model and $\boldsymbol{v}(k)$ is the observation noise. It is assumed that $\mathbb{E}[\boldsymbol{w}(k)] = \mathbb{E}[\boldsymbol{v}(k)] = 0$, $\mathbb{E}[\boldsymbol{w}(k)\boldsymbol{w}^T(j)] = \boldsymbol{Q}(k)\delta_{k,j}$, $\mathbb{E}[\boldsymbol{v}(k)\boldsymbol{v}^T(j)] = \boldsymbol{R}(k)\delta_{k,j}$ and $\mathbb{E}[\boldsymbol{w}(k)\boldsymbol{v}^T(j)] = 0$, where $\mathbb{E}$ is the expectation symbol. $\boldsymbol{w}(k)$ and $\boldsymbol{v}(k)$ are assumed to be Gaussian.

Then, the estimate of the state is defined as

$$\hat{\boldsymbol{x}}(k|j) \equiv \mathbb{E}[\boldsymbol{x}(k)|\boldsymbol{z}(1), ..., \boldsymbol{z}(j)], \tag{C.3}$$

i.e. the expected value of state $\boldsymbol{x}(k)$ at time step k based on observations up to time step j. It is assumed that $j \leq k$. Finally, the covariance of the state is defined as

$$\boldsymbol{P}(k|j) \equiv \mathbb{E}[(\boldsymbol{x}(k) - \hat{\boldsymbol{x}}(k|j))(\boldsymbol{x}(k) - \hat{\boldsymbol{x}}(k|j))^T|\boldsymbol{z}(1), ..., \boldsymbol{z}(j)], \tag{C.4}$$

For such a system, the Kalman filter provides a recursive strategy for calculating $\hat{\boldsymbol{x}}(k|k)$ based on $\hat{\boldsymbol{x}}(k-1|k-1)$ and the new observation $\boldsymbol{z}(k)$, as well as $\boldsymbol{P}(k|k)$ based on and $\boldsymbol{P}(k-1|k-1)$.

The filter starts with a prediction step from time k to time $k+1$.

Prediction:

$$\hat{\boldsymbol{x}}(k|k-1) = \boldsymbol{F}(k)\hat{\boldsymbol{x}}(k-1|k-1) \tag{C.5}$$

$$\hat{\boldsymbol{z}}(k|k-1) = \boldsymbol{H}(k)\hat{\boldsymbol{x}}(k|k-1) \tag{C.6}$$

$$\boldsymbol{P}(k|k-1) = \boldsymbol{F}(k)\boldsymbol{P}(k-1|k-1)\boldsymbol{F}^T(k) + \boldsymbol{G}(k)\boldsymbol{Q}(k)\boldsymbol{G}^T(k) \tag{C.7}$$

Then, observations are registered and an update step takes place.

Update:

$$\hat{\boldsymbol{x}}(k|k) = \hat{\boldsymbol{x}}(k|k-1) + \boldsymbol{W}(k)\tilde{\boldsymbol{z}}(k) \tag{C.8}$$

$$\boldsymbol{P}^{-1}(k|k) = \boldsymbol{P}^{-1}(k|k-1) + \boldsymbol{H}^T(k)\boldsymbol{R}^{-1}(k)\boldsymbol{H}(k) \tag{C.9}$$

where

$$\boldsymbol{W}(k) = \boldsymbol{P}(k|k)\boldsymbol{H}^T(k)\boldsymbol{R}^{-1}(k) \tag{C.10}$$

is the Kalman gain and

$$\tilde{\boldsymbol{z}}(k) = \boldsymbol{z}(k) - \hat{\boldsymbol{z}}(k|k-1) \tag{C.11}$$

is the innovation factor. Furthermore, the equations (C.5), (C.6) and (C.7) are initialized with $\boldsymbol{P}(-1|-1) = \boldsymbol{P_0}$ and $\hat{\boldsymbol{x}}(0|-1) = \boldsymbol{x_0}$. It is worth noting that the information form of the covariance is used here.

C.2 DKF simulation experiments

The scripts producing the performance evaluation presented in section 3.5 are available in appendix H. Each scenario is executed 20 times. For each, the RMSE (root mean square error) is calculated, finally keeping the average over the 20 executions. The nodal and internodal validation bounds are found empirically. For this evaluation, the standard deviation of the noise on the position and the velocities of the observations is of 0.5 m. The corresponding nodal and internodal validation bounds are, [8, 7, 7, 7] and [6, 9, 12, 13] respectively (for 2, 3, 4 and 5 *"agent-cameras"*).

Appendix D

Potential field method

The potential field method was developed by roboticists and is mainly used for path planning. Forces deriving from a potential modulated by $N_{\text{objectives}}$ attractive points and $N_{\text{repulsive}}$ repulsive points acting on the robot to make it move. This section summarizes the expressions found in the literature ([19]) .

The potential $U_{res}(\boldsymbol{x})$ and the corresponding force acting on the agent $\overrightarrow{F}_{res}(\boldsymbol{x})$ are computed as:

$$\boldsymbol{x} = (x_c, y_c) \tag{D.1}$$

$$U_{res}(\boldsymbol{x}) = \sum_{n=1}^{N_{\text{objectives}}} U_{att,n}(\boldsymbol{x}) + \sum_{n=1}^{N_{\text{obstacles}}} U_{rep,n}(\boldsymbol{x}) \tag{D.2}$$

$$\overrightarrow{F}_{res}(\boldsymbol{x}) = -\nabla U_{res}(\boldsymbol{x}) = \sum_{n=1}^{N_{\text{objectives}}} \overrightarrow{F}_{att,n}(\boldsymbol{x}) + \sum_{n=1}^{N_{\text{obstacles}}} \overrightarrow{F}_{rep,n}(\boldsymbol{x}), \tag{D.3}$$

where (x_c, y_c) is the center of the camera. U_{att}, U_{rep}, F_{att}, F_{rep} are defined in the next section. This algorithm has the advantage to be relatively fast, as the force (equation D.3) can be computed locally only on the point of interest and without needing to calculate the potential (equation D.2) as well. With this method, every point (attractive and repulsive) has an impact on the final force.

D.1 Potential and forces found in literature

The domain of the potential field is the set of points satisfying $\rho(\boldsymbol{x}) < \rho_0$, where ρ_0 is a predefined distance. Equation D.4 corresponds to the definition of $\rho(\boldsymbol{x})$ as found in the literature.

$$\rho(\boldsymbol{x}) = \rho_{\text{circular}}(\boldsymbol{x}) = \sqrt{x^2 + y^2}. \tag{D.4}$$

This equation is transformed in section 4.6.1 to get the equivalent for an ellipse and a line.

Note : $\rho(\boldsymbol{x})$ is in this example centered around $(0, 0)$. A coordinate change on $\rho(\boldsymbol{x})$ (domain of application) is required to move and orient it anywhere in the 2-D plane.

D.1.1 Attractive potential and forces

$$U_{att}(\boldsymbol{x}) = \frac{\xi}{2}\rho(\boldsymbol{x})^2 \tag{D.5}$$

$$\overrightarrow{F}_{att}(\boldsymbol{x}) = -\nabla U_{att}(\boldsymbol{x}) = \begin{cases} -\xi\rho(\boldsymbol{x})\nabla\rho(\boldsymbol{x}) & \text{if } \rho(\boldsymbol{x}) < \rho_0 \\ \text{constant} & \text{elsewhere} \end{cases} \tag{D.6}$$

ξ is a constant value to modify the potential amplitude and ρ_0 is a distance.

D.1.2 Repulsive potential and forces

$$U_{rep}(\boldsymbol{x}) = \begin{cases} \frac{\eta}{2}\left(\frac{1}{\rho(\boldsymbol{x})} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho(\boldsymbol{x}) < \rho_0 \\ 0 & \text{elsewhere} \end{cases} \tag{D.7}$$

$$\overrightarrow{F}_{rep}(\boldsymbol{x}) = -\nabla U_{rep}(\boldsymbol{x}) = \begin{cases} \frac{\eta}{2}\left(\frac{1}{\rho((\boldsymbol{x}))} - \frac{1}{\rho_0}\right)\frac{1}{\sqrt{\rho_0}}\nabla\rho(\mathbf{x}) & \text{if } \rho(\boldsymbol{x}) < \rho_0 \\ 0 & \text{elsewhere} \end{cases} \tag{D.8}$$

η is a constant value to modify the potential amplitude and ρ_0 is a distance.

Appendix E

Principal component analysis

The PCA approach aims at approximating a stochastic process $\boldsymbol{X} = [\boldsymbol{x_1}, ..., \boldsymbol{x_n}]$, $\boldsymbol{\mu} = \mathbb{E}(\boldsymbol{X})$ based on its eigenvectors and its covariance matrix. The goal is to find an orthonormal basis $\boldsymbol{\Phi} = [\boldsymbol{\phi_1}, ..., \boldsymbol{\phi_n}] \Rightarrow \langle \boldsymbol{\phi_i}, \boldsymbol{\phi_j} \rangle = \delta_{ij}$; such that:

$$\boldsymbol{X} - \boldsymbol{\mu} = \Phi \cdot \boldsymbol{b} \tag{E.1}$$

As the basis is orthonormal, the coefficients b_i are computed by projecting the signal on the associated orthogonal basis vector:

$$b_i = \langle \boldsymbol{X} - \boldsymbol{\mu}, \boldsymbol{\phi_i} \rangle = (\boldsymbol{X} - \boldsymbol{\mu})^T \boldsymbol{\phi_i}^* \tag{E.2}$$

The goal is to find uncorrelated coefficient b. Since the covariance matrix is a diagonal matrix the idea is to use it in order to identify the coefficients, as illustrated in the equations:

$$\mathbb{E}\{b_i, b_j^*\} = \mathbb{E}\{\boldsymbol{\phi_i^{*T}}(\boldsymbol{X} - \boldsymbol{\mu})(\boldsymbol{X} - \boldsymbol{\mu})^{*T} \boldsymbol{\phi_j}\} = \boldsymbol{\phi_i^{*T}} C_{xx} \boldsymbol{\phi_j} \tag{E.3}$$

$$\boldsymbol{\phi_i^{*T}} C_{xx} \boldsymbol{\phi_j} = \lambda_j \delta_{ij} \Rightarrow C_{xx} \boldsymbol{\phi_j} = \lambda_j \boldsymbol{\phi_j} \tag{E.4}$$

Therefore the search basis is formed from the eigenvectors of the covariance matrix. λ represent its eigenvalues. An interesting feature is that the signal energy is equal to the sum of all eigenvalues:

$$\mathbb{E}\{\|\boldsymbol{x} - \boldsymbol{\mu}\|^2\} = \mathbb{E}\{\|\boldsymbol{\phi_i} \cdot \boldsymbol{b}\|^2\}$$
$$= \mathbb{E}\left\{ \left(\sum_i b_i \boldsymbol{\phi_i} \right)^T \left(\sum_j b_j \boldsymbol{\phi_j} \right) \right\} = \sum_i \sum_j \mathbb{E}\{b_i b_j\} \underbrace{\boldsymbol{\phi_i^T} \boldsymbol{\phi_j}}_{\delta_{ij}} = \sum_i \lambda_i \tag{E.5}$$

The approximation of the signal $\mathbf{x} - \mu$ by $m < N$ basis functions produces an error equal to:

$$\mathbb{E}\{\|\mathbf{x} - \hat{\mathbf{x}}\|^2\} = \mathbb{E}\left\{\left\| \sum_{i=m+1}^{N} b_i \phi_i \right\|^2\right\} = \sum_{i=m+1}^{N} \lambda_i \tag{E.6}$$

This theoretical reminder is based on the section *"Eigenwert-Verfahren"* of [20].

Appendix F

Algorithms

F.1 Obstruction detection algorithm

Given the camera configuration - x_c, y_c, α, β - and the targets configurations - x_t, y_t, r - it is possible to determine analytically which targets are obstructed. Targets and cameras are represented under the assumptions of section 4.2.

Procedure:

1. Applying a frame coordinate (in frame x^*, y^*) change to set the x axis as the bisector of the camera field. Every coordinate change needs to be computed for each target.

   Camera :

   $$x_c^* = 0, y_c^* = 0, \alpha^* = 0 \qquad \text{(F.1)}$$

   Targets :

   $$x_t^*, y_t^* \qquad \text{(F.2)}$$

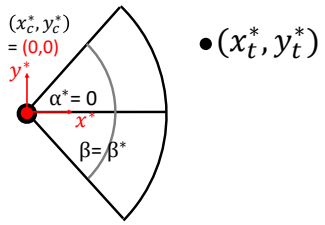


Figure F.1: Camera and target in the new frame

2. Removing each target that does not respect the condition:

   $$\frac{-\beta}{2} < \text{angle}_t < \frac{\beta}{2} \quad \text{and} \quad \text{norm}_t < \text{field depth} \qquad \text{(F.3)}$$

   where,

   $$\text{norm}_t = \sqrt{x_t^{*2} + y_t^{*2}} \quad \text{and} \quad \text{angle}_t = atan2(y_t^*, x_t^*) \qquad \text{(F.4)}$$

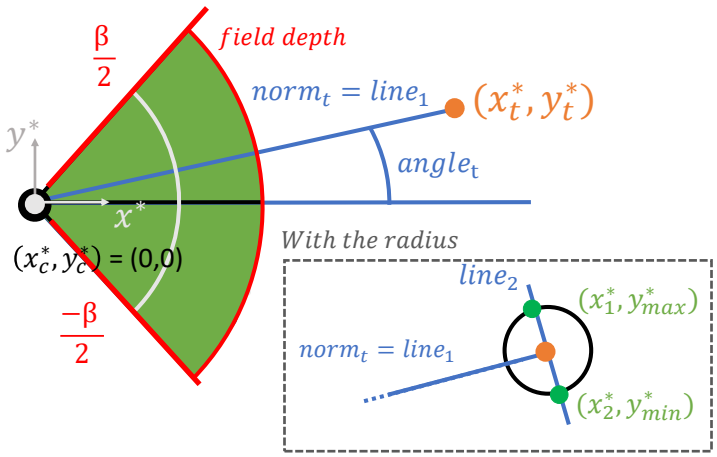


Figure F.2: Condition *"in field"* graphically represented

This condition (F.3) states that the center of the target is inside the field of view. However it does not take the radius into account. To do so, multiple steps are required (steps are illustrated in the figure F.2) :

(a) Computation of line 1: link the target center (x_t^*, y_t^*) and the camera center (x_c^*, y_c^*).

(b) Computation of line 2: perpendicular to line 1 passing trough the point (x_t^*, y_t^*).

(c) Computation of target extreme points (x_1^*, y_{max}^*) and (x_2^*, y_{min}^*) : intersection of line 2 with a circle or radius r center on (x_t^*, y_t^*)

(d) Computation of $\text{angle}_{\text{max}}\text{angle}_{\text{min}}$:

$$\text{angle}_{\text{t,max}} = atan2(y_{max}^*, x_1^*) \quad \text{and} \quad \text{angle}_{\text{t,min}} = atan2(y_{min}^*, x_2^*) \quad \text{(F.5)}$$

(e) Modification from the condition:
If only seeing one of the extremities is sufficient to recognize the target, the condition becomes:

$$\text{angle}_{t,max} > -\frac{\beta}{2} \quad \text{or} \quad \text{angle}_{t,min} < \frac{\beta}{2} \quad \text{and} \quad \text{norm}_t < \text{field depth} \quad \text{(F.6)}$$

If seeing both extremities is needed, the condition becomes:

$$\text{angle}_{t,max} < \frac{\beta}{2} \quad \text{and} \quad \text{angle}_{t,min} > -\frac{\beta}{2} \quad \text{and} \quad \text{norm}_t < \text{field depth} \quad \text{(F.7)}$$

It should be noted that $norm_t$ is defined using the center of the target. Checking the two extremities of the target instead of only the center was also an option. However, if the radius of the targets is assumed to be small, this approximation is reasonable and allows for a simpler algorithm.

3. By this point, all the targets within the field of view of the agent are known. One should now verify whether any of the targets within this set obstructs the view to any other.

   To that end, all pairs of targets are checked to verify whether one obstructs the other. To verify if a target is obstructed, a set of points are defined on it, that are called *"verification points"*. If any of these points is not visible, then the target is considered obstructed. The procedure to check if a point is visible is as follow:

   Let (x_o^*, y_o^*) be the point being tested. The region obstructed by the target needs to be defined. This region depends on the camera position and target position (x_t^*, y_t^*) as represented in the figure F.3. However, by applying a coordinate transformation from the room coordinates to the camera coordinates, the obstructed region only depends on the target position, as the camera will be located at the origin.

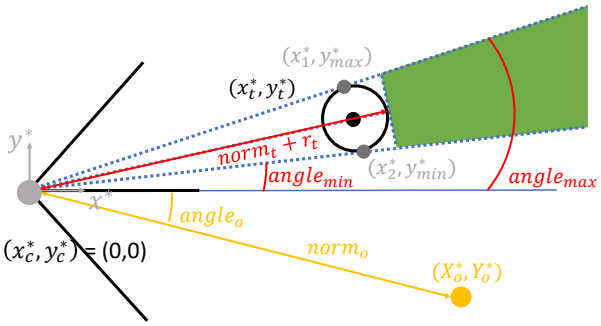


Figure F.3: Condition *"hidden"* graphically represented

$$\text{norm}_\text{o} = \sqrt{x_o^{*2} + y_o^{*2}} \quad \text{and} \quad \text{angle}_\text{o} = atan2(y_o^*, x_o^*) \tag{F.8}$$

To take into account the perspective of the camera, two lines are drawn: one from $(0,0)$ to the (x_1^*, y_{max}^*) and one from $(0,0)$ to (x_2^*, y_{min}^*). Adding the fact that the field is hidden behind the target, the condition becomes (green area in the figure F.3):

$$\text{angle}_{t,min} < \text{angle}_o < \text{angle}_{t,max} \quad \text{and} \quad \text{norm}_\text{t} + r_t < \text{norm}_\text{o} \tag{F.9}$$

An efficient computation can be achieved by saving the norms and extremities computed in point 2. An iterative procedure can then be used, starting with the closet target (from the point (x_c^*, y_c^*)). When a target is found to be in the hidden zone it is removed and not taken into account anymore.

For instance one could chose to set the *"verification points"* as follows:

- If only considering the center, the point (x_o^*, y_o^*) becomes the target center.
- If also considering the radius, apply an iterative process
  (a) (x_o^*, y_o^*) corresponds to the target center.
  (b) then, (x_o^*, y_o^*) corresponds to one of the extremities.
  (c) finally (x_o^*, y_o^*) corresponds to the other extremity.
- This approach can be generalized to n points and for any shape.

F.2 Motion and orientation change detection

The goal is to detect whether a target is fix or moving. If it is moving, the direction changes should also be detected. It is assumed that the positions and speeds are measured.

$$(\boldsymbol{x_n}, \boldsymbol{y_n}) \text{ being the n last positons values} \tag{F.10}$$

$$(\boldsymbol{v_{xn}}, \boldsymbol{v_{yn}}) \text{ being the n last speed values} \tag{F.11}$$

- *"Position algorithm"*

$$d_i = \sqrt{x_i^2 + y_i^2} \tag{F.12}$$

The target is fix if : $\text{std}(\boldsymbol{d_n}) < \text{security margin} * \sqrt{2} * \text{std error on position}$

- *"Speed algorithm"*

$$d_{vi} = \sqrt{v_x i^2 + v_y i^2} \tag{F.13}$$

The target is fix if : $\text{mean}(\boldsymbol{d_{vn}}) < \sqrt{2} * \text{mean error on speed}$

Detecting orientation changes is achieved only using speeds. Replacing simply the positions by speeds in what we called *"position algorithm"*.

Appendix G

Class Diagram of Simulation

This section contains the class diagram. Each region denoted by a letter is zoomed in the following pages.

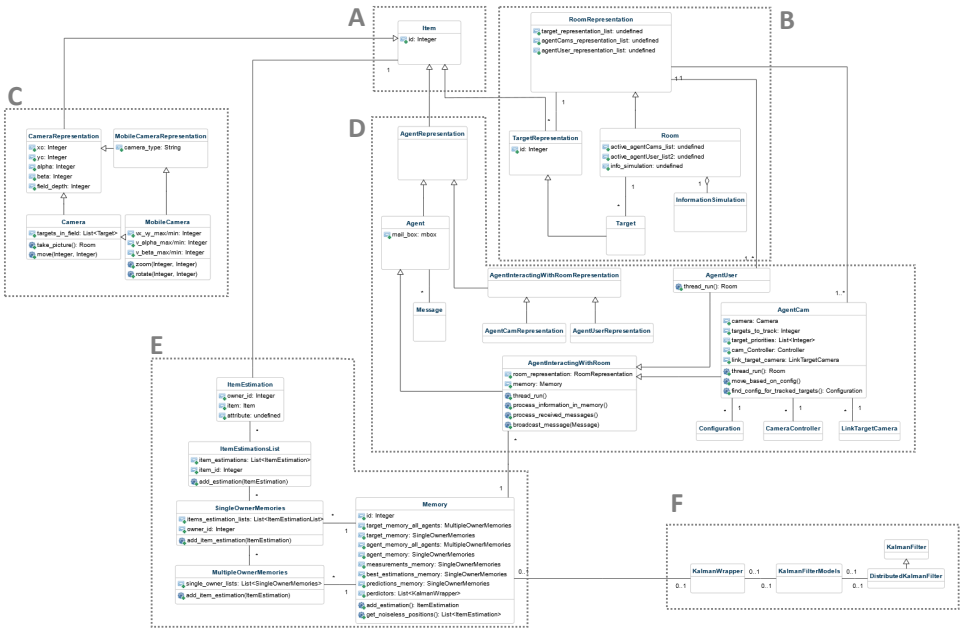


Figure G.1: Class diagram of simulation

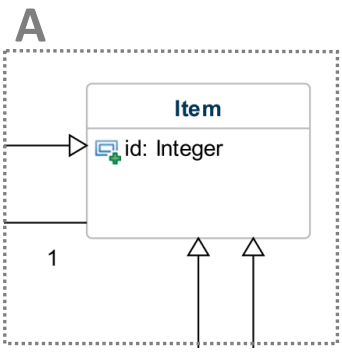


Figure G.2: A - Item Class

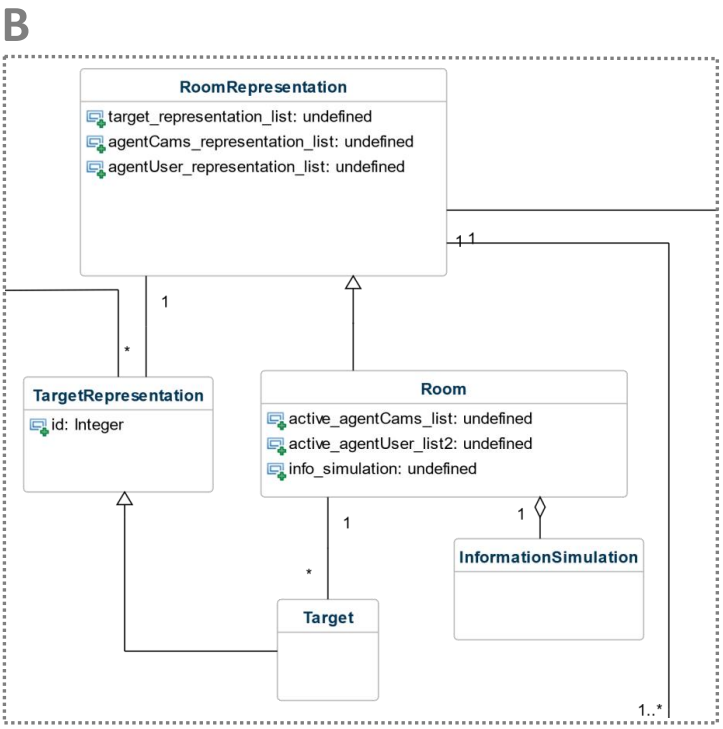


Figure G.3: B - Room and Target

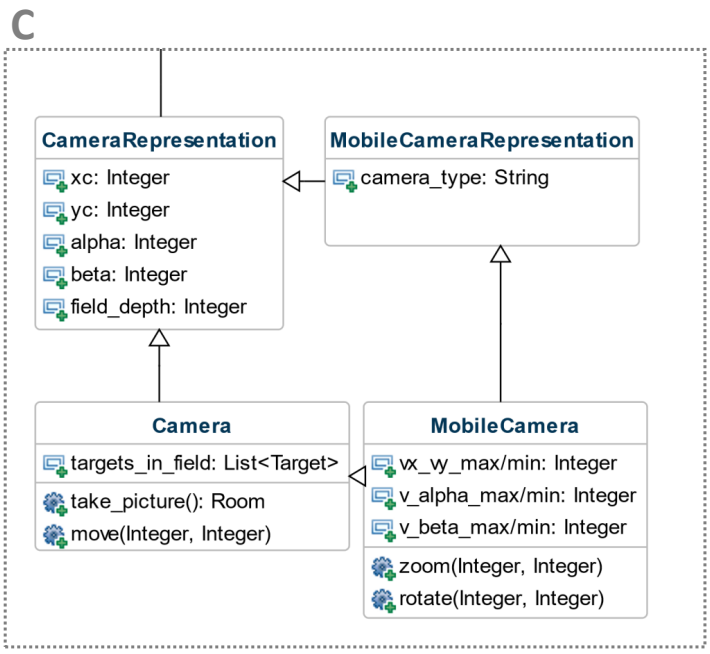


Figure G.4: C - Cameras

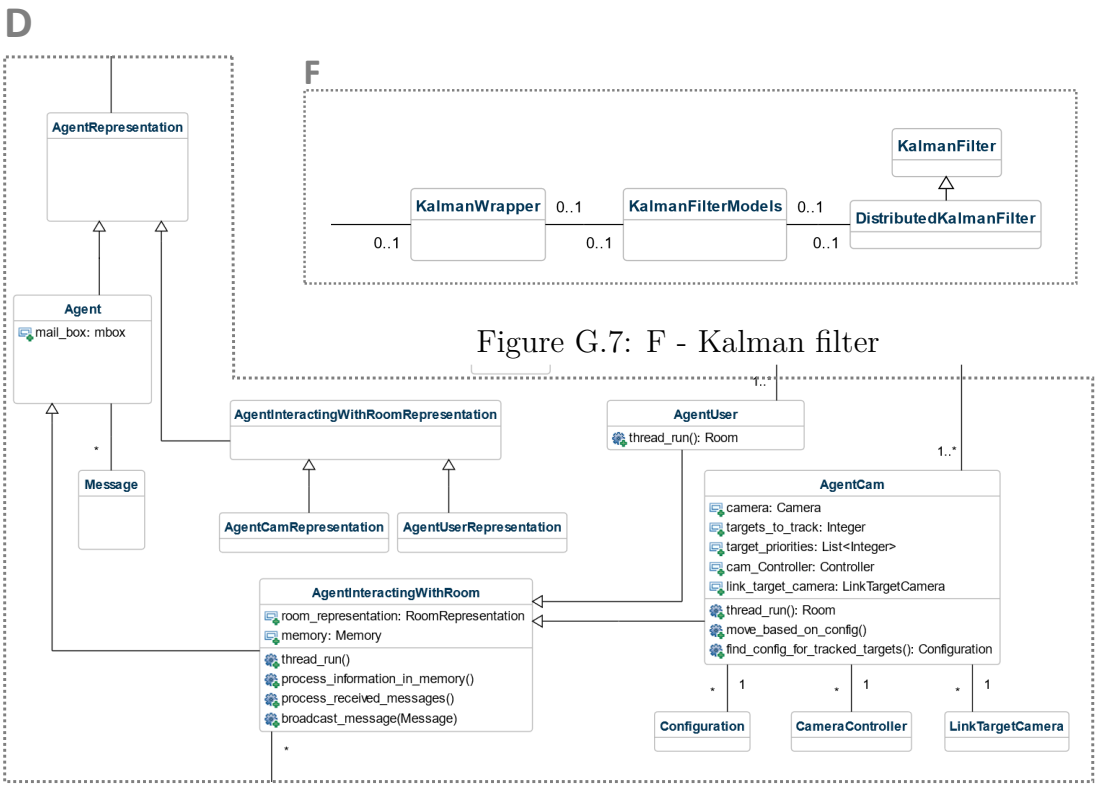


Figure G.7: F - Kalman filter

Figure G.5: D - Agents

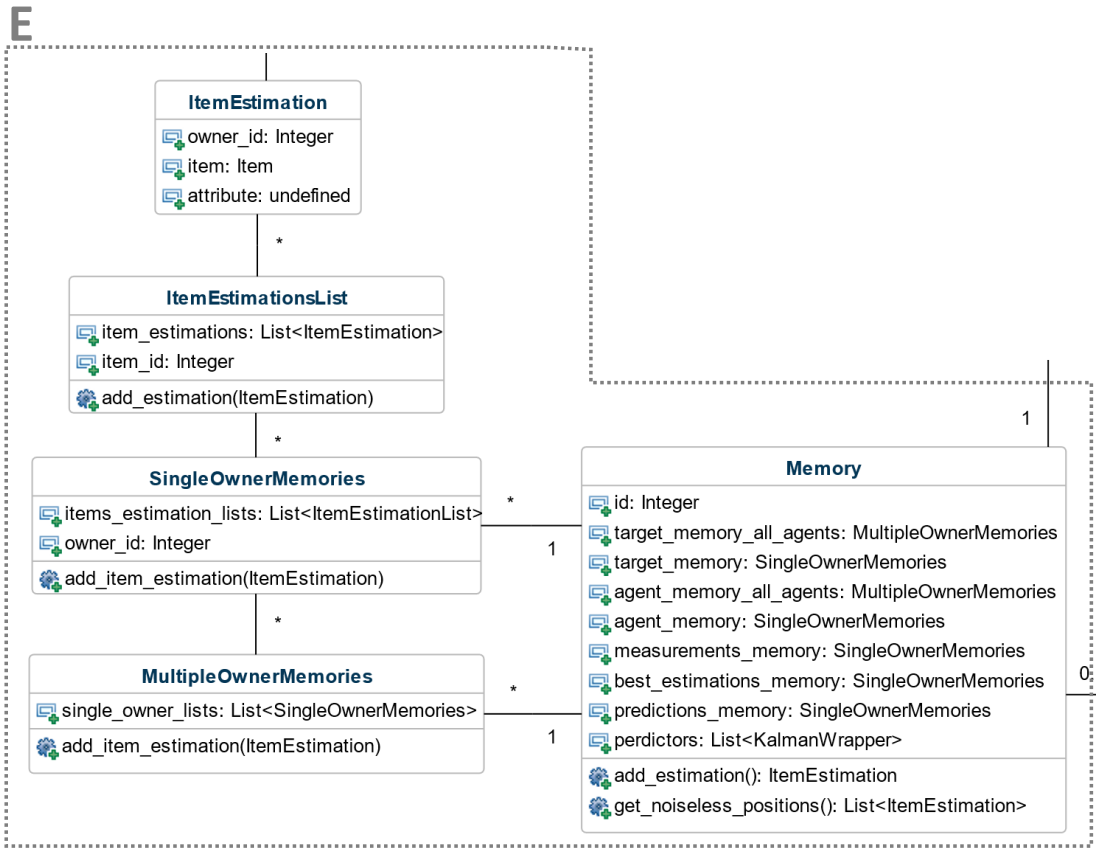


Figure G.6: E - Memory

Appendix H

Link to resources

This section groups several link, which contains the data used to obtain some of the results presented in this report. Settings for the parameter are available in the file *constants-"simulation-name".py*, however even with identical parameters, the multi-thread architecture gives slightly varying results.

- **General link to the simulator:**
  `https://github.com/xenakal/Simulation_Interactions`
- **Illustrative videos of the case studies:**
  `https://github.com/xenakal/Simulation_Interactions/tree/master/videos_use_cases_report`
- **Chapter 1: Case study *"introduction"* (section 1.2):**
  `https://github.com/xenakal/Simulation_Interactions/tree/master/to_share/Data_saved-chapitre1-use_case/plot`
- **Chapter 2: Messages exchanged (section 2.5):**
  `https://github.com/xenakal/Simulation_Interactions/tree/master/to_share/Data_saved-chapitre1-use-case-DKF/plot/messages`
- **Chapter 2: Case study *"conclusion"* (section 2.8) link:**
  - **Without failure:**
    `https://github.com/xenakal/Simulation_Interactions/tree/experiments_ch_2_4/to_share/Data_saved-chapitre2-use_case`
  - **With failure:**
    `https://github.com/xenakal/Simulation_Interactions/tree/experiments_ch_2_4/to_share/Data_saved-chapitre2-use_case-failure`

- **Chapter 3: Kalman (sections 3.5 & 3.6):**
  `https://github.com/xenakal/Simulation_Interactions/tree/master/dkf_evaluation`
- **Chapter 4: Case study *"controller results"* (section4.6.2)**
  - $\xi \neq 0$, ***"linear-mode"*:**
    `https://github.com/xenakal/Simulation_Interactions/tree/experiments_ch_2_4r/to_share/data_saved-controller-xi`
  - $\xi = 0$, ***"linear-mode"*:**
    `https://github.com/xenakal/Simulation_Interactions/tree/experiments_ch_2_4/to_share/data_saved-controller-repulsive-mode`
  - $\xi = 0$, ***"attractive-mode"*:**
    `https://github.com/xenakal/Simulation_Interactions/tree/experiments_ch_2_4/to_share/data_saved-controller-attractive-mode`
- **Chapter 4: Case study *"conclusion"* (section 4.7)link:**
  `https://github.com/xenakal/Simulation_Interactions/tree/experiments_ch_2_4/to_share/Data_saved-chapitre4-use_case`

Bibliography

[1] N. V. Belov, B. Y. Buyanov, and V. A. Verba, "Real-time object classification on aircraft board using raspberry pi 3,"

[2] H. R. Hashemipour, S. Roy, and A. J. Laub, "Decentralized structures for parallel kalman filtering," *IEEE Transactions on Automatic Control*, vol. 33, no. 1, pp. 88–94, 1988.

[3] B. Rao, H. Durrant-Whyte, and J. Sheen, "A fully decentralized multi-sensor system for tracking and surveillance," *The International Journal of Robotics Research*, vol. 12, no. 1, pp. 20–44, 1993.

[4] M. B. D. A. E. M. E. G. P. Alexandre Gramfort, Olivier Grisel, "Principal component analysis." `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html?highlight=pca#sklearn.decomposition.PCA`, 2007–2019.

[5] P. S. Foundation, "18.4. mailbox — manipulate mailboxes in various formats." `https://docs.python.org/2/library/mailbox.html`, 1990–2020.

[6] R. R. L. Jr, "Filterpy - kalman filters and other optimal and non-optimal estimation filters in python." `https://github.com/rlabbe/filterpy`, 2015.

[7] P. Alriksson and A. Rantzer, "Distributed kalman filtering using weighted averaging," in *Proceedings of the 17th International Symposium on Mathematical Theory of Networks and Systems*, pp. 2445–2450, Kyoto Kyoto, Japan, 2006.

[8] D. P. Spanos, R. Olfati-Saber, and R. M. Murray, "Distributed sensor fusion using dynamic consensus," in *IFAC World Congress*, Citeseer, 2005.

[9] V. Isler, S. Khanna, J. Spletzer, and C. J. Taylor, "Target tracking with distributed sensors: The focus of attention problem," *Computer Vision and Image Understanding*, vol. 100, no. 1-2, pp. 225–247, 2005.

[10] X. Chen and J. Davis, "Camera placement considering occlusion for robust motion capture," *Computer Graphics Laboratory, Stanford University, Tech. Rep*, vol. 2, no. 2.2, p. 2, 2000.

[11] R. Semaan, "Optimal sensor placement using machine learning," 2016.

[12] J. Siswantoro, A. S. Prabuwono, and A. Abdullah, "Real world coordinate from image coordinate using single calibrated camera based on analytic geometry," in *International Multi-Conference on Artificial Intelligence Technology*, pp. 1–11, Springer, 2013.

[13] H. M. Lee and W. C. Choi, "Algorithm of 3d spatial coordinates measurement using a camera image," *J Image Graphics*, vol. 3, no. 1, 2015.

[14] P. Tissainayagam and D. Suter, "Object tracking in image sequences using point features," *Pattern Recognition*, vol. 38, no. 1, pp. 105–113, 2005.

[15] AlexeyAB, "Yolov4 - neural networks for object detection." `https://github.com/AlexeyAB/darknet`, 2014–2020.

[16] shizukachan, "Darknet with nnpack." `https://github.com/shizukachan/darknet-nnpack`, 2014–2018.

[17] azuredsky, "Yolo opencl inference engine (linux & windows)." `https://github.com/azuredsky/YoloOCLInference`, 2017.

[18] S. Canu, "Yolo v3 - install and run on nvidia jetson nano (with gpu)." `https://pysource.com/2019/08/29/yolo-v3-install-and-run-yolo-on-nvidia-jetson-nano-with-gpu/`, 2019.

[19] N. Amato, "Potential field methods." 2004.

[20] D. W. Prof.Dr. Uwe Kiencke, Dipl.-Ing.Michael Schwarz. Oldenbourg Wissenschaftsverlag GmbH, 2008.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**