SAPIENZA
UNIVERSITÀ DI ROMA

Simulating and tracking multiple users in smart spaces

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Candidate

Dario Litardi
ID number 1489178

Thesis Advisor

Prof. Massimo Mecella

Co-Advisor

Dr. Francesco Leotta

Academic Year 2018/2019

Thesis defended on 31 October 2019
in front of a Board of Examiners composed by:

Prof. Rosati (chairman)
Prof. Grisetti
Prof. Querzoni
Prof. Mecella
Prof. Scardapane

Simulating and tracking multiple users in smart spaces
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: October 23, 2019

Author's email: litardi.1489178@studenti.uniroma1.it

Abstract

The goals of this thesis are the simulation of a smart environment and the tracking of the trajectories of the users that inhabit it. The first issue that has been addressed is the emulation of the users and their movements inside a smart space. The smart spaces studied are exclusively indoor environments. This problem has been solved by designing and developing a new software that acts as a simulator. It generates the trajectories of each person that moves inside the smart environment and it lets to graph in real time these paths. Two types of trajectories have been taken into consideration: continuous and discrete. A continuous trajectory can be define as a set of points with a fine granularity. A discrete trajectory is defined as a sequence of positions in the space calculated at the triggering of the PIR sensors installed inside the smart space. The discrete trajectory is called as a trajectory with coarse granularity or rarefied trajectory. The main difference between the continuous and discrete trajectories is in the level of granularity of the data: coarse in the discrete case depending from the the triggering of the sensors and fine in the continuous. A continuous trajectory is also called dense trajectory. The second phase of the thesis work has been dedicated to the analysis and implementation of algorithms and their validation, for tracking and classifying both the continuous and the discrete trajectories of each different user.

Contents

Chapter 1

Introduction

Smart environments have the potential to allow users to engage and interact seamlessly with their immediate surroundings. This has been made possible by the introduction of intelligent technologies, coupled with software-based services. It is evident that technological advances have provided a new era for both sensing technology and computational processing to facilitate the vision of smart environments. Although a number of challenges exist in their deployment, a number of large-scale programs are endeavoring to progress their uptake further. Smart spaces are not some sort of virtual reality, but rather physical environments decked out with technology. They are implemented with monitors and sensors that enable humans and integrated technological systems to interact. These environments could improve personal and professional productivity, increase energy efficiency, simplify complex processes, and potentially make daily life easier and less stressful. Smart homes, sometimes known as connected homes, are an example of smart spaces, with technology that improves safety, convenience, entertainment, and productivity. These smart homes have become commonplace. Other smart spaces run the gamut of different types of environments: offices and communal workspaces, apartment communities, hotels, and other types of lodging properties, malls, grocery stores, shopping centers, hospitals and other healthcare providers, public places such as libraries and schools. Transportation portals such as airports and train stations could all be designed as varied types of smart spaces. As technology becomes more advanced and more widely used, technologies will allow these environments to connect in unique and evolving ways. This seamless integration could promote a safer, cleaner, healthier environment on a much larger scale.

A big challenge in this field is to classify the habits of the several users that live in a smart space. Each person move inside a smart environment every day, e.g. a smart house or a smart office, tracking some habitual paths, from a usual object or target to another, e.g. a sofa, a desk or a kitchen table. The smart spaces that we have included are only indoor environment. The data of the trajectories of the habitual paths of each user can be collected from the position sensors, PIR, inside the smart space; these data are stored in a unique log called sensor log. The goal is to classify the trajectories of the users and go back to the original habitual path of each person.

The thesis work can be divided in two main part. The first part regards the

development of a simulator for emulating a smart space and generating the users trajectories of the habitual paths. The second phase concerns in the analysis and the implementation of algorithms for reconstructing and classifying both continuous and discrete trajectories. The tasks of the simulator are the generation of the users trajectories and their graphic representation on a topological map of the smart space. In the second part of the work, the goal is to extract the trajectories of each user from the unique sensor log and to reconstruct their paths. This issue has been solved using different methods, depending on the type of the trajectories: Kalman filters, Supported Vector Machines.

Chapter 2

Introducing smart spaces

In this section will be described the main technologies used and briefly the current state of art in this research field.

2.1 Smart spaces

Smart spaces represent an emerging class of IoT-based applications. Smart homes and offices are representative examples where pervasive computing could take advantage of ambient intelligence (AmI) more easily than in other scenarios where artificial intelligence. The universAAL specification defines a smart space as "an environment centered on its human users in which a set of embedded networked artefacts, both hardware and software, collectively realize the paradigm of ambient intelligence" (AmI)[7]. AmI systems are characterized by the following features: sensitivity, responsiveness, adaptivity, ubiquity and transparency. The term sensitivity refers to the ability of an AmI system to sense the environment and, more generally, to understand the context it is interacting with. Strictly related to sensitivity are responsiveness and adaptivity, which denote the capability to timely act, in a reactive or proactive manner, in response to changes in the context according to user preferences (personalization). Sensitivity, responsiveness and adaptivity all contribute to the concept of context awareness. Finally, the terms ubiquity and transparency directly refer to concept of pervasive computing. AmI is obtained by merging techniques from different research areas including artificial intelligence (AI) and human-computer interaction (HCI).

The research community have come up with several concepts that address different aspects of smart spaces. The vision of a smart spaces system incorporate ideas found in Nomadic Computing, Ubiquitous Computing, Wearable Computing, Intelligent Environments and Co-operative Buildings. The fundamental ideas behind Nomadic Computing are derived from the observation that, today, computer users can be considered to be nomads, in that they carry portable computers and communication devices on their travels between office, home, hotel, car and so on. These portable devices range from laptops and Personal Digital Assistants (PDA) to smart cards and wristwatch computers. The nomad is expecting his devices to work adequately wherever he may travel, even though environment and communication capabilities may change dramatically as he moves. Access to computing and communications

is necessary, not only from “home base” of the user but also during travels and when the nomad reaches his final destination. This new user behaviour implies fundamental changes in the requirements on our computing and communication systems. The key functions in Nomadic Computing is independence, referring to perception of computing environment and ability to automatically adjust to momentary capabilities of communication and access in a transparent and integrated fashion. The concept of Nomad Computing is rather wide in scope and it partly comprises many of today's computer and telecom systems such as GSM and Wireless LAN. Nevertheless, the introduction of Nomadic Computing stipulates the major shift of paradigm in how we use computers in our daily life. It also points out many important issues and research areas that are central for establishing smart space systems. Ubiquitous Computing extends the ideas of Nomadic Computing. In any Ubiquitous Computing system many computers are located in the environment surrounding the user. The key ideas are that they should be abundant, invisible to the user and networked together. The user, with help from a set of small devices, interacts with hundreds of computers located in the physical world around him or her. However, these personal devices are supposed to be small and inexpensive, so that every one can afford them. In order to create a working structure for Ubiquitous computing, computers of all sizes and shapes are needed, not only those computers equipped with a mouse, keyboard and display. Finally, Ubiquitous computing is about facilitating system interaction. Interaction is not necessarily limited to typing at a keyboard or clicking in a display. A Ubiquitous computing system accepts user voice input. Wearable computer systems oppose to the Ubiquitous Computing approach by having computers everywhere. A user in a wearable computer system carries all necessary data and equipment with him or her, worn like clothing. Data is kept in the worn device and is not shared with the environment. If two different wearable want to share the same resource, they need the capability to communicate with each other. In a wearable computer system it is up to the wearable device to solve the contention for accessing a common resource. The resource itself is highly specialised and it has very little extra computational power. This produces high demands on flexibility of the software of the wearable device. An Intelligent Environment resembles a Ubiquitous Computing system. However, in the user needs no device to enable interaction with the Intelligent Environment system. The system accepts only camera and microphone input. Intelligent environments, as defined by Michael H. Cohen, are “spaces in which computation is seamlessly used to enhance ordinary activity”. By embedded Cohen means that the main input for theses systems are cameras, microphones and speakers coupled with computer vision, speech recognition and speech synthesis algorithms. These functions allows the user to perform interaction as he or she naturally should feel comfortable with, by performing a dialogue with the system as if the system was another person. The aim is to make the user forget that he or she is interacting with a software system. Finally, A Co-operative Building as defined by Norbert Streitz et al. is ”flexible and dynamic environment that provides co-operative workspaces supporting and augmenting human communication and collaboration”. The Co-operative Building is a kind of Ubiquitous Computing system where the emphasis is in enabling computer-aided co-operative work. However, the co-operative building should not only augment co-operation among the users but also work in a co-operative manner

with them. Furthermore, a Co-operative Building should not be limited by the physical dimensions of it, but also provide seamless access and interaction with the virtual space. The parts of a system that constitute a Co-operative Building may not be physically located at the same place. The system on a whole brings together local environments to a seamless entity. Within this entity users can interact as easily as being in the same location.Smart environments are broadly classified to have the following features:

- Remote control of devices, like power line communication systems to control devices.
- Device Communication, using middleware, and Wireless communication to form a picture of connected environments.
- Information Acquisition/Dissemination from sensor networks.
- Enhanced Services by Intelligent Devices.
- Predictive and Decision-Making capabilities.

Figure 2.1. Example of the huge application fields of the smart spaces.

2.1.1 Background

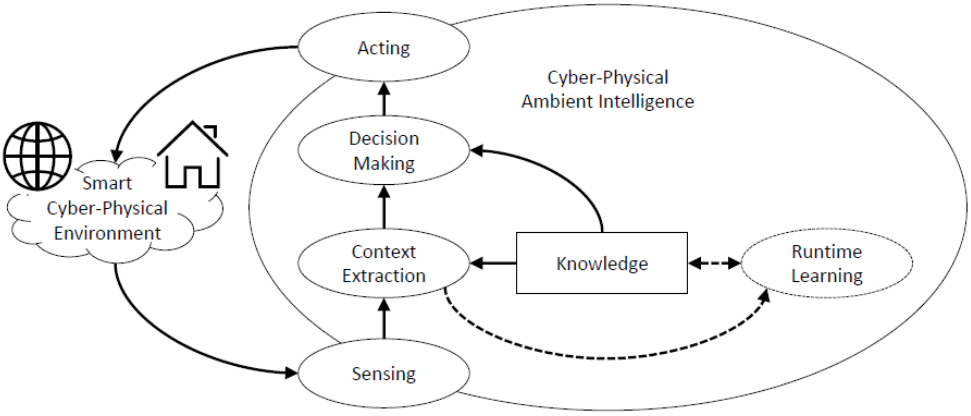


Figure 2.2. The ambient intelligence closed loop. Arrows denote information flow. Dashed lines are used to denote optional information flows.

Figure 2.2 depicts the closed loops that characterize a running smart space. The main closed loop, depicted using solid arrows and shapes, shows how the knowledge of environment dynamics and of users behaviours and preferences is employed to interpret sensors output in order to perform appropriate actions on the environment. Sensor data is first analysed to extract the current context, which is an internal abstraction of the state of the environment from the point of view of the AmI system. The extracted context is then employed to make decisions on the actions to perform on the controlled space. Actions related to these decisions modify the environment (both physical and digital) by means of actuators of different forms. Sensors can be roughly divided into physical ones, which provide direct measurements about the environment (e.g., humidity, brightness, temperature), the devices and the users, and cyber ones, which provide digital information, not directly related to physical phenomena, such as user calendars. A cyber sensor often provides information related to the presence of the user in the cyberspace (e.g., the calendar of an user, a tweet posted by him/her, etc.). The term sensor data encompasses raw (or minimally processed) data retrieved from both physical sensors and cyber sensors. We can imagine a smart space producing, at runtime, a sensor log containing raw measurements from available sensors[6].

Definition 1 *(Sensor Log). Given a set S of sensors, a sensor log is a sequence of measurements of the kind <ts, s, v> vi where ts is the timestamp of the measurement, $s \in S$ is the source sensor and v the measured value,which can be either nominal (categorical) or numeric (quantitative)[8].*

Measurements can be produced by a sensor on a periodic base (e.g., temperature measurements) or whenever a particular event happens (e.g., door openings)[8]. As many of the algorithms proposed in the literature borrow the terminology of data mining, the sensor log could be conceived as a sequence of events instead of a sequence of measurements.

»

Definition 2 *(Event Log). Given a set $E = \{e_1,...,e_{nE}\}$ of event types, an event sequence is a sequence of pairs <e, t>, where $e \in E$ and t is an integer, the occurrence time of the event type e.*

Definition 2 is more restrictive than Definition 1 . Translating a sensor log into an event log could cause a loss of information especially if discretization of periodic sensor measurements is required. Authors in the field of smart spaces uses, sometimes as synonyms, a variety of terms to refer to the state of the environment and the tasks humans perform in it. For the rest of the article, we will use the following terminology:

- **Context:** The state of the environment including the human inhabitants. This includes the output of sensors and actuators, but also the state of human inhabitants including the action/activities/habits he/she is performing. In this very comprehensive meaning, the term situation is sometimes used.
- **Action:** Atomic interaction with the environment or a part of it (e.g., a device). Recognizing actions can be easy or difficult depending on the sensors installed. Certain methods only focuses on actions and they will not be part of our survey. In some cases methods to recognize activities and habits completely skip the action recognition phase, only relying on the raw measurements in the sensor log.
- **Activity:** A sequence of actions (one in the extreme case) or sensor measurements/events with a final goal. In some cases an action can be an activity itself (e.g., ironing). Activities can be collaborative, including actions by multiple users and can interleave one each other. The granularity (i.e., the temporal extension and complexity) of considered activities cannot be precisely specified. According to the approach, tidying up a room can be an activity whereas others approaches may generically consider tidying up the entire house as an activity. In any case, some approaches may hierarchically define activities, where an activity is a combination of sub-activities.
- **Habit:** A sequence or interleaving of activities that happen in specific contextual conditions (e.g., what the user does every morning between 08:00 and 10:00).

Knowledge plays a central role in AmI systems. As it intervenes both for context extraction and decision-making, it takes the form of a set of models describing (i) users behaviour, (ii) environment/device dynamics, and (iii) user preferences. Anyway, knowledge should not be considered as a static resource as both users behaviour and preferences change over time. Vast majority of works in the area of ambient intelligence suppose the knowledge to be obtained off-line, independently from the system runtime. A second optional loop in Figure 2.2, depicted using dashed arrows, shows that the current context could be employed to update the knowledge by applying learning techniques at runtime. Noteworthy, AmI is not intended to be provided by a centralized entity, on the contrary, its nature is distributed with embedded devices and software modules, possibly unaware one of each other, contributing to its features.

The Information Society Technologies advisory group (ISTAG) suggests that the following characteristics will permit the societal acceptance of ambient intelligence. AmI should:

- facilitate human contact.
- be oriented towards community and cultural enhancement.
- help to build knowledge and skills for work, better quality of work, citizenship and consumer choice.
- inspire trust and confidence.
- be consistent with long term sustainability—personal, societal and environmental—and with lifelong learning.
- be made easy to live with and controllable by ordinary people.

A variety of technologies can be used to enable Ambient intelligence environments such as Bluetooth Low, Energy, RFID, Microchip implant, Sensors like Ambient light sensor (photodetector), thermometers, proximity sensors and motion detectors,Software agents, Affective computing, Nanotechnology, Biometrics.

2.1.2 Supported sensors

Sensing technologies have made significant progress on designing sensors with smaller size, lighter weight, lower cost, and longer battery life. Sensors can, thus, be embedded in an environment and integrated into everyday objects and onto human bodies without affecting users' comfort. Nowadays, sensors do not only include those traditionally employed for home and building automation (e.g., presence detectors, smoke detectors, contact switches for doors and windows, network-attached and close circuit cameras) but also more modern units (e.g., IMU—Inertial Measurements Units such as accelerometer and gyroscopes, WSN nodes), which are growingly available as part of off-the-shelf products. Measured values are usually affected by a certain degree of uncertainty. Sensors have indeed their own technical limitations as they are prone to breakdowns, disconnections from the system and environmental noise (e.g., electromagnetic noise). As a consequence, measured values can be out of date, incomplete, imprecise, and contradictory with each other. Techniques for cleaning sensor data do exist, but uncertainty of sensor data may still lead to wrong conclusions about the current context, which in turn potentially lead to incorrect behaviours of the system[7].

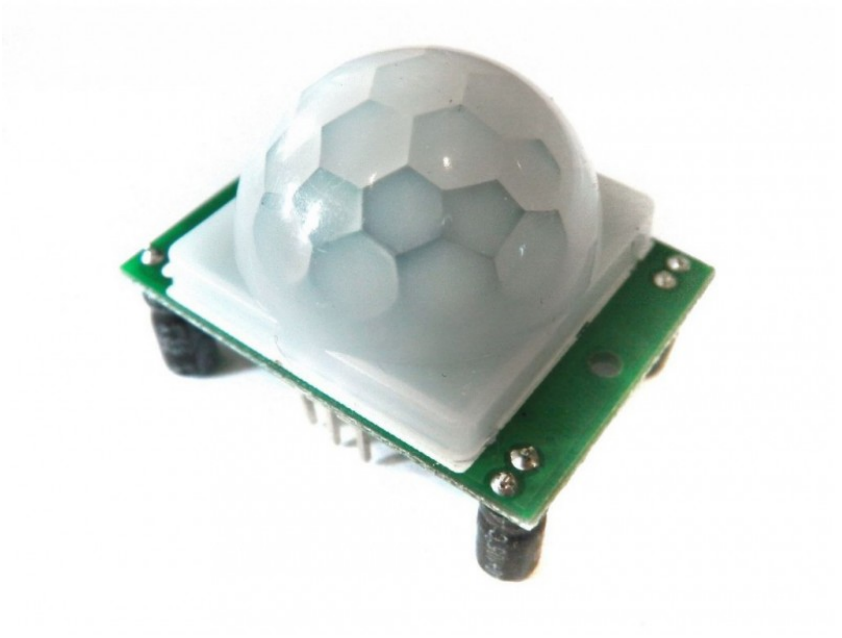


Figure 2.3. A PIR sensor (Passive InfraRed sensor).

Formalisms employed for representing knowledge in AmI systems often need environmental variables to be binary or categorical. A wide category of sensors (e.g., temperature sensors) produce instead numerical values, making it necessary

to discretize sensor data before they can be used for reasoning[7]. Discretization methods in machine learning and data mining are usually classified according to the following dimensions:

- *Supervised vs. Unsupervised.* Unsupervised methods do not make use of class information in order to select cut-points. Classic unsupervised methods are equal-width and equal-frequency binning, and clustering. Supervised methods employ instead class labels in order to improve discretization results.
- *Static vs. Dynamic.* Static discretization methods perform discretization, as a preprocessing step, prior to the execution of the learning/mining task. Dynamic methods instead carry out discretization on the fly.
- *Global vs. Local.* Global methods, such as binning, are applied to the entire n-dimensional space. Local methods, as the C4.5 classifier, produce partitions that are applied to localized regions of the instance space. A local method is usually associated with a dynamic discretization method.
- *Top-down vs. Bottom-up.* Top-down methods start with an empty list of cut-points (or split-points) and keep on adding new ones to the list by splitting intervals as the discretization progresses. Bottom-up methods start with the complete list of all the continuous values of the feature as cut-points and remove some of them by merging intervals as the discretization progresses.
- *Direct vs. Incremental.* Direct methods directly divide the range of a quantitative attribute in k intervals, where the parameter k is provided as input by the user. Conversely, incremental methods start from a simple discretization and improve it step by step in order to find the best value of k.

2.1.3 Issues of Smart spaces

The concept of Smart Space touches several fields of computer science. Smart Spaces are envisioned to be brought in our daily life where computing traditionally have not been before. Since the Smart Space system will be all around us at all times, it will create challenging demands on how users interact with it, how the devices communicate with each other and how it protects sensitive information. Having a user centric view, Smart Spaces provides an easy interaction with services and resources within it. Also, means of interaction should be non-intimidating and non-intrusive. Currently there are two major approaches to interacting with a Smart Space system. In the first case, the user does need to have any artificial device in order to interact. It suffices for the user to be present and have functional vocal cords. Such a system accepts camera and microphone input. In the second, the user must be provided with some kind of device. Human-computer interaction issues here are concerned with limited GUI space and efficient distribution of resource capabilities. A Smart Space system is populated with several heterogeneous devices. It must enable a generic way of locating devices and provide a minimal set of common communication primitives. There are basically two ways of solving resource locating. One is by taking a client/server approach, where the server has a list of all available resources and where they can be found. The other is to keep a decentralised approach

and let the resources keep a small execution environment where devices can download code. When this code is executed, the resource performs the desired effect. Finally, users may make sensitive information available without being conscious of it. How should the Smart Space protect the user from this? Also, in a Smart Space there are potentially several different kinds of sensors such as cameras or microphones. A user may want to remain incognito or be locatable. Finally, Smart Spaces also need to protect information that a user has given away consciously.

2.2 State of art

Knowledge is represented in AmI systems using models. The literature about representing models of human habits is wide. In this section, we will review the most adopted approaches, highlighting those formalisms that are human understandable, thus, being easy to validate by a human expert or by the final user (once the formalism is known). The main model types are:

- Bayesian classifiers;
- Hidden Markov Models (HMMs);
- Support Vector Machines (SVM);
- Artificial Neural Networks (ANNs);
- Business process mining.

Knowledge is represented in AmI systems using models. The literature about representing models of human habits is wide. In this section, we will review the most adopted approaches, highlighting those formalisms that are human understandable„ thus, being easy to validate by a human expert or by the final user (once the formalism is known).

Bayesian classification techniques are based on the well known Bayes theorem $P(H|X)=\frac{P(X|H)P(H)}{P(X)}$, where H denotes the hypothesis (e.g., a certain activity is happening) and X represents the set of evidences (i.e., the current value of context objects). As calculating P(X|H) can be very expensive, different assumptions can be made to simplify the computation. For example, naïve Bayes (NB) is a simple classification model, which supposes the n single evidences composing X independent (that the occurrence of one does not affect the probability of the others) given the situational hypothesis; this assumption can be formalized as $P(X|H) = \prod_{k=1}^{N} P(x_k|H)_i$. The inference process within the naïve Bayes assumption chooses the situation with the maximum a posteriori (MAP) probability. Hidden Markov Models (HMMs) represent one of the most widely adopted formalism to model the transitions between different states of the environment or humans. Here hidden states represent situations and/or activities to be recognized, whereas observable states represent sensor measurements. HMMs are a statistical model where a system being modelled is assumed to be a Markov chain, which is a sequence of events. A HMM is composed of a finite set of hidden states (e.g., st1, st, and st+1) and observations (e.g., ot1, ot, and ot+1) that are generated from states. HMMis built on three assumptions:

(i) each state depends only on its immediate predecessor, (ii) each observation variable only depends on the current state, and (iii) observations are independent from each other. In a HMM, there are three types of probability distributions: (i) prior probabilities over initial state p(s_0); (ii) state transition probabilities p(stjst1); and (iii) observation emission probabilities p(otjst). A drawback of using a standard HMM is its lack of hierarchical modeling for representing human activities. To deal with this issue, several other HMM alternatives have been proposed, such as hierarchical and abstract HMMs. In a hierarchical HMM, each of the hidden states can be considered as an autonomous probabilistic model on its own; that is, each hidden state is also a hierarchical HMM. HMMs generally assume that all observations are independent, which could possibly miss long-term trends and complex relationships. Conditional Random Fields—CRFs, on the other hand, eliminate the independence assumptions by modeling the conditional probability of a particular sequence of hypothesis, Y, given a sequence of observations, X; succinctly, CRFs model P(Y|X). Modeling the conditional probability of the label sequence rather than the joint probability of both the labels and observations P(X,Y), as done by HMMs, allows CRFs to incorporate complex features of the observation sequence X without violating the independence assumptions of the model. The graphical model representations of a HMM (a directed graph, Figure 2a) and a CRF (an undirected graph, Figure 2b) makes this difference explicit. A comparison between HMM and CRF is shown, where CRF outperforms HMM in terms of timeslice accuracy, while HMM outperforms CRF in terms of class accuracy[7].

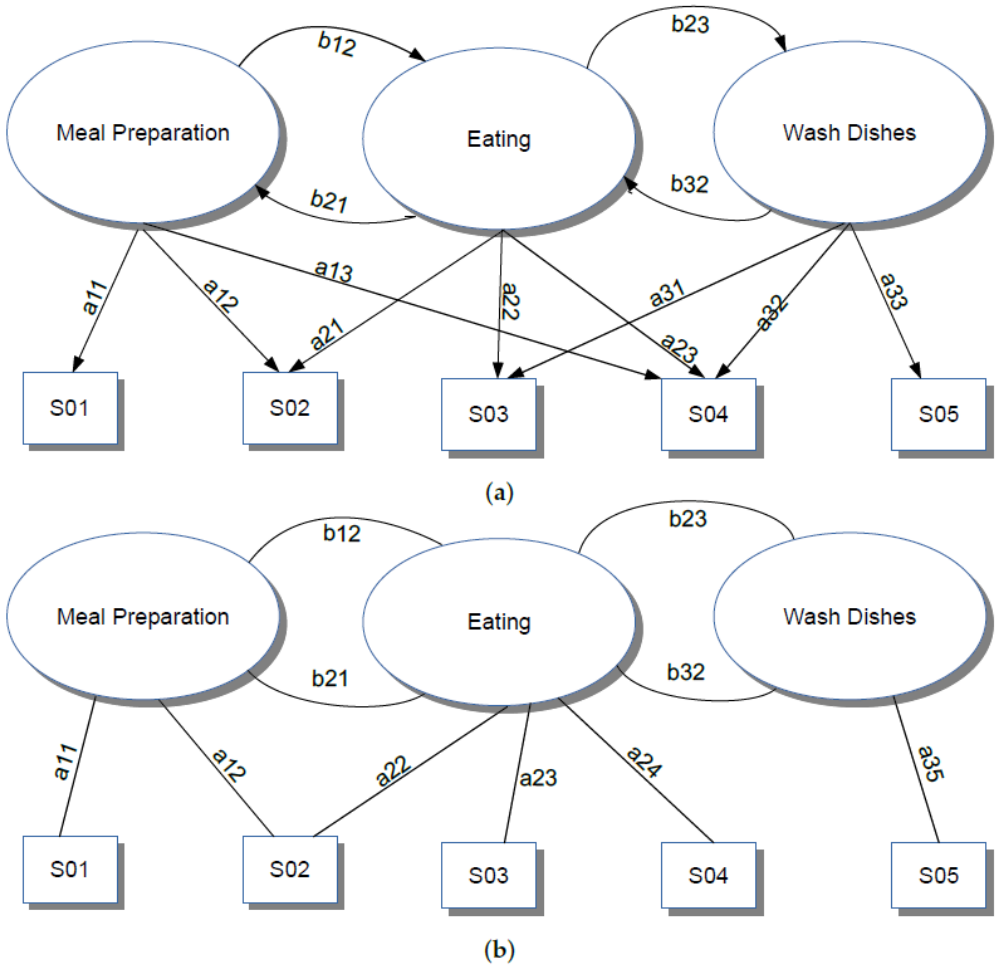


Figure 2.4. Examples of HMM and CRF models. Ellipses represent states (i.e., activities). Rectangles represent sensors. Arrows between states are state transition probabilities (i.e., the probability of moving from a state to another), whereas those from states to sensors are emission probabilities (i.e., the probability that in a specific state a sensor has a specific value). (a) HMM model example. Picture inspired by CASAS-HMM and CASAS-HAM. (b) CRF model example. Picture inspired by KROS-CRF.

Another statistical tool often employed is represented by Markov Chains (MCs), which are based on the assumption that the probability of an event is only conditional to the previous event. Even if they are very effective for some applications like capacity planning, in the smart spaces context, they are quite limited because they deal with deterministic transactions and modelling an intelligent environment with this formalism results in a very complicated model. Support Vector Machines (SVMs) allow to classify both linear and non-linear data. A SVM uses a non-linear mapping to transform the original training data into an higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane that separates the training data of one class from another. With an appropriate non-linear mapping to a sufficiently high dimension, data from two classes can always be separated. SVMs are good at handling large feature spaces since they employ overfitting protection, which does not necessarily depend on the number of features. Binary Classifiers

are built to distinguish activities. Due to their characteristics, SVMs are better in generating other kind of models with a machine learning approach than modelling directly the smart environment. For instance authors uses them combined with Naive Bayes Classifiers to learn the activity model built on hierarchical taxonomy formalism shown in Figure 3.

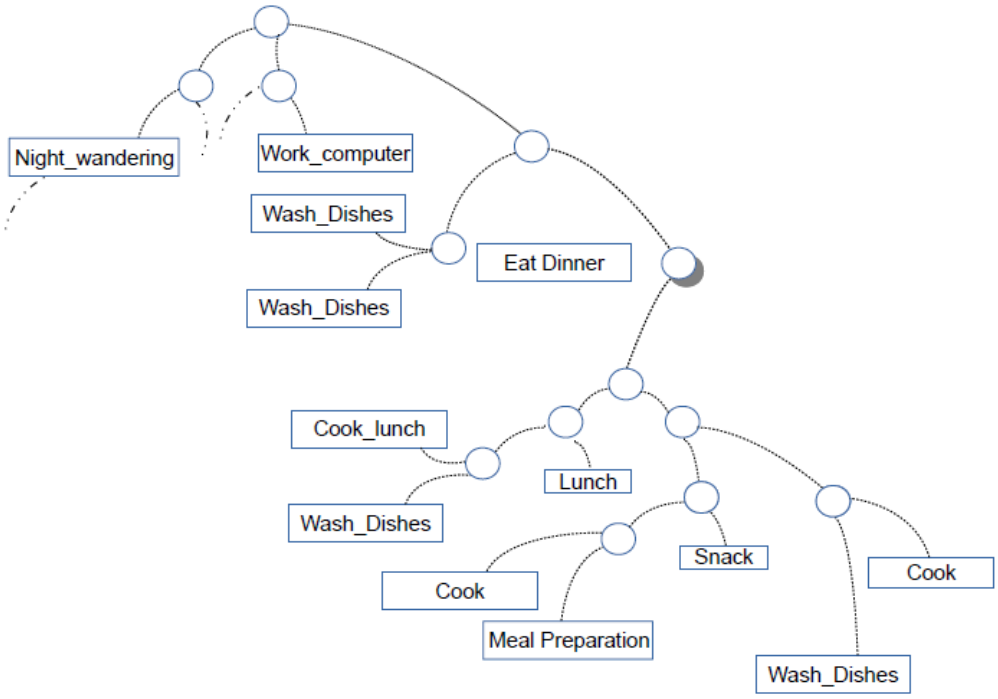


Figure 2.5. Cluster hierarchical model portion example obtained by exploiting SVMs.

Artificial Neural Networks (ANNs) are a sub-symbolic technique, originally inspired by biological neuron networks. They can automatically learn complex mappings and extract a non-linear combination of features. A neural network is composed of many artificial neurons that are linked together according to a specific network architecture. A neural classifier consists of an input layer, a hidden layer, and an output layer. Mappings between input and output features are represented in the composition of activation functions f at a hidden layer, which can be learned through a training process performed using gradient descent optimization methods or resilient backprogagation algorithms.

Models can be either manually defined (specification-based methods) or obtained through machine learning techniques (learning-based methods). In the first case, models are usually based on logic formalisms, relatively easy to read and validate (once the formalism is known to the reader), but their creation requires an heavy cost in terms of expert time. In the latter case, the model is automatically learned from a training set (whose labelling cost may vary according to the proposed solution) but employed formalism are usually taken from statistics, making them less immediate to understand. Another difference between the two approaches is that whereas specification-based methods use human actions as main modelling elements, learning-based ones directly refer to sensor measurements, thus loosing the focus on human

actions and making even more difficult to visually inspect and validate produced models. On the other hand, taking as input raw sensor measurements usually makes learning-based methods easier to apply in a practical context; whereas, in the vast majority of cases, specification-based methods do not face the problem of translating sensor measurements into actions. Applying methods originally taken from the area of business process mining to human habits may represent a compromise between specification-based and learning-based methods, provided that the gap between raw sensor measurements and human actions can be filled in by performing a log preprocessing step. Such a log preprocessing step may consist of simple inferences on data or complex machine learning algorithms.

2.3 Lack of datasets

The goal of this work is to identify the trajectories of a given sensor log, possibly coming from a smart environment populated by many users, that are related to the actions of individual users. The project can be divided in two main phase. In the first phase of the work we have developed a simulator for generating the trajectories of the paths belonging to different users inside of a smart space. The initial goal was to generate a dataset containing the trajectories of each user that moves in a smart house for several days. The issue was the lack of tool and simulator able to getting datasets ad-hoc for our project. In a previous thesis project online datasets had been used but was bare, with coarse granularity, hence ineffective to analyse later. The choice has been to create a simulator able to resolve this problem. The simulator "Home Designer" has been developed for two main purposes: generating accurate datasets of user's trajectories and graphing the paths of each user inside the house. The simulator allows to:

- Build the map of the house, but also in general of a smart space.
- Placing the targets inside the house. A target is an object of habitual use by the users, e.g. a TV, a bed or a desk.
- Placing the position sensors. We have simulated the behaviour of PIR sensors, (Passive InfraRed). These sensors are sensitive to the passage of a person. They cover the space with a cone of shadow and the length of the radius can be customized: the minimum radius is one meter.
- Simulate the trajectory of each user. The simulation can be customized, choosing the number of the users that move inside the house, the speed of each user, the type of the motion, constant or variable, the number of the simulation's days and the type of the trajectory, continuous or discrete. The discrete trajectories are generated enabling the PIR sensors on the simulator. In case of continuous trajectories the software save in the datasets the timestamp and the coordinates (x,y) for each position, in the bidimensional space of the map of the house.
- Graph the trajectory of each user. The simulator assigns to each user a different colour for each trajectory.

The simulation generates for each user a dataset that register all timestamps and positions of the path. In the discrete case, the dataset has three attributes: the timestamp, the sensor name of the triggered sensor and the status of the sensor. In the continuous case, the attributes of the dataset are the timestamp and the coordinates x, y. This model follows the definition of Sensor Log.

2.4 The issue of the multi-user

The second part of the work has been dedicated to the resolution of multi-user's issue. A Sensor Log is a log where are saved the timestamps and the positions of the user's trajectories. By its nature a Sensor Log contains all the trajectories of the users and doesn't distinguish them. The challenge is to extract the positions from the log and recreate the different paths made by each person. The analysis is carried out both on continuous and discrete trajectories.

The main problem of the classification of trajectories is the presence of crossing trajectories.

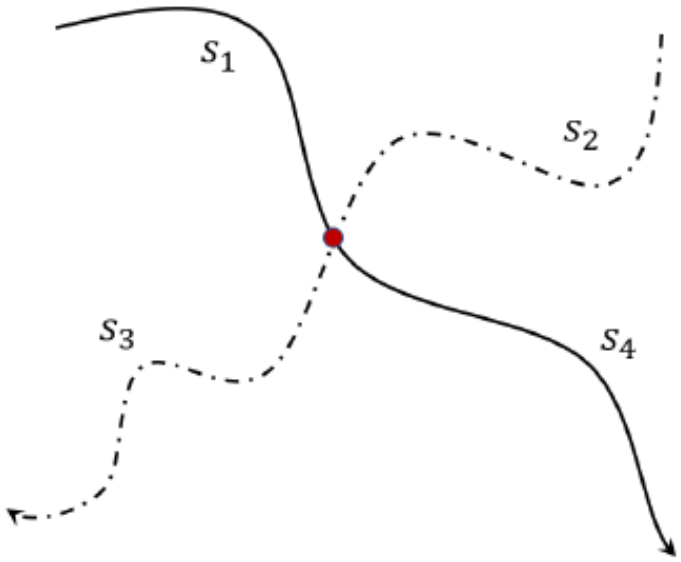


Figure 2.6. Two intersecting trajectories of two users.

Suppose we have two trajectories as in Figure 2.6. We divide the trajectories in the segments s1 and s2 before the intersection and in the segments s3 and s4 after. We need to find a method for understanding in which direction continue respectively both the segment s1 and s2, if in s3 o s4, and so to reconstruct the original trajectories. This problem is resolved both for the analysis with continuous trajectories and discrete.

2.4.1 The issue of the multi-user in the continuous-time systems

The initial goal of the analysis is comparing the dynamic continuous system versus the discrete system that characterize the movements of the users. Below we define what is a trajectory both in the continuous system and discrete.

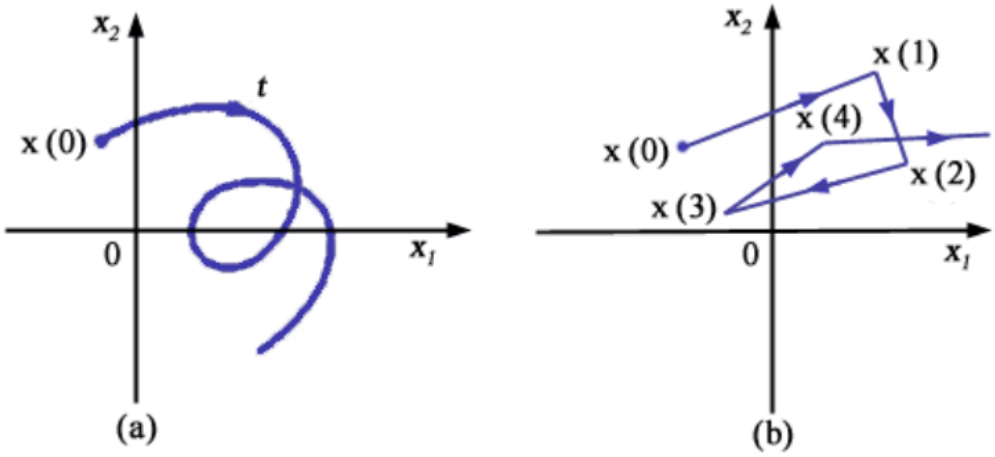


Figure 2.7. (a) Continuous trajectory, (b) discrete trajectory

We define what is theoretically a trajectory. A trajectory is the geometric line drawn in space from a moving point. The function x(•) identified is called movement, while the set {x(t), t≥0} in the state space R^n is called trajectory or orbit. In the case of continuous-time systems, the trajectory is therefore a line rooted at point x(0) and with a precise direction of travel, that of time (see Figure 4.1a). In the case of discrete-time systems, the trajectory is instead an ordered sequence of points {x(0), x(1), ...} which, for reasons of clarity, is however customary to join with rectilinear segments oriented as shown in Figure 4.1b.

In our project, we remember that a continuous trajectory can be define as a set of points with a fine granularity. A **discrete trajectory** is defined as a sequence of positions in the space calculated at the triggering of the PIR sensors. The discrete trajectory is called as a trajectory with coarse granularity or **rarefied trajectory**. The main difference between the **continuous** and **discrete** trajectories is in the level of granularity of the data: **coarse** in the discrete case depending from the the triggering of the sensors and **fine** in the continuous. A continuous trajectory is also called **dense** trajectory. Indeed the main goal of our project is to analyse the coarse and discrete trajectories generated by the movement of each user under the PIR sensors.

Regard to the previous example in Figure 2.6, we have two trajectories divided in four segments: s1, s2, s3, s4. In case of the continuous system, we have continuous trajectories in which the points that compose it are strictly close. For this reason, we can apply prediction algorithms for knowing the next positions at each step. We have chosen the Kalman Filter. Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. Kalman filter uses a system's dynamic model (e.g., physical laws of motion), known control inputs to that system, and multiple sequential measurements (such as from sensors) to form an estimate of the system's varying quantities (its state) that is better than the estimate obtained by using only one measurement alone. We use the Kalman Filter in this way. The algorithm gets the

last three points of each segment previous to the intersection: in the example in Figure 2.6 the algorithm takes these points from the segments s1 and s2. Then it applies the Kalman Filter using these points and it estimates the next points of the segment s1 and s2. The two next estimated positions that the filter returns are assigned to the segments s3 and s4 based on the minimum distance from the first points of s3 and s4. At the end, the algorithm can choose for each segment, namely s1 and s2, its successor segment after the crossing point: in our case the result will output two assignments <s1, s4> and <s2, s3>. In this way we can classify the trajectories of all users. The analysis and the details of this adopted algorithm will be reported in the fourth chapter.

The problem of the multi-user presents another critical issue: the parallel trajectories. We show an example of this problem in Figure 4.8.

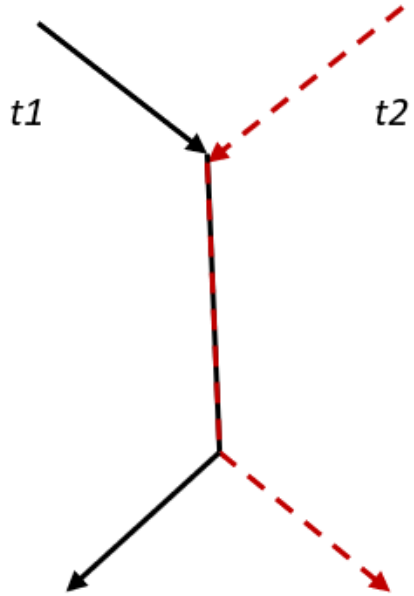


Figure 2.8. An example of two parallel trajectories t1, t2, one in black and the other with the red dotted line.

Since two trajectories for example t1 and t2 as in Figure cross in a point and then move in parallel, the Kalman Filter cannot be applied in case of parallelisms. The simple reason is that the estimated points coincide and it is impossible to assign these positions to each trajectory. The adopted solution of this critical issue will be discussed later.

2.4.2 The issue of the multi-user in the discrete-time systems

The problem of the multi-user presents many critical issues in the discrete dynamic system. We remember that a discrete trajectory is an ordered sequence of points {x(0), x(1), ..., x(n)}, and for each point correspond a timestamp {t(0), t(1), ..., t(n)} in the time. The issues for extracting, classifying and reconstructing the trajectories of all users present in a unique sensor are:

- the coarse granularity of the data that are the positions and the timestamps of the discrete trajectories;
- the presence of many crossing points and the impossibility due to the low granularity to apply any prediction algorithm for estimating the next positions;
- the presence of many parallel trajectories respect to the continuous case.

We have solved these problems with an algorithm which will discuss later.

Chapter 3

Design and realization of the simulator

In this chapter will be presented the design and the implementation's details of the simulator. First of all we recap the state of art regard to the simulators of trajectories and the existing mathematical models.

3.1 State of art

The literature about the existing mathematical models for representing the trajectories in the indoor smart environments is wide. In this paragraph we recap the main models that are used. We mention two mathematical models for smoothing the trajectories and making them continuous:

- Voronoi diagram;
- Spline interpolation.

A Voronoi diagram, from the name of Georgii Voronoi, also called Voronoi tessellation, Voronoi decomposition, or Dirichlet tessellation, consists of a partitioning of the plane into n polygons derived from n main points, where each polygon contains only one of the n main points and where every other point of the polygon is closer to the main point of the polygon than to all the other main points. The perimeter of each polygon is halfway between two main points[10].

We give a formal definition. Let $\mathbf{X}$ be a metric space with distance function d. Let $\mathbf{K}$ be a set of indices and let $(P_k)_k \in K$ be a tuple (ordered collection) of nonempty subsets (the sites) in the space $\mathbf{X}$. The Voronoi cell, or Voronoi region, $\mathbf{R}_k$, associated with the site $\mathbf{P}_k$ is the set of all points in $\mathbf{X}$ whose distance to $\mathbf{P}_k$ is not greater than their distance to the other sites $\mathbf{P}_j$, where j is any index different from k. In other words, if $\mathrm{d(x, A)} = \inf\{\mathrm{d(x,a)} \mid \mathrm{a}\in\mathrm{A}\}$ denotes the distance between the point x and the subset $\mathbf{A}$, then:

$$\mathbf{R}_k = \{\mathrm{x} \in \mathbf{X} \mid \mathrm{d(x,}\mathbf{P}_k\mathrm{)} \leq \mathrm{d(x,}\mathbf{P}_j\mathrm{)} \text{ for all } \mathrm{j}\neq\mathrm{k}\}$$

The Voronoi diagram is simply the tuple of cells $(R_k)_k \in K$. In principle, some

of the sites can intersect and even coincide (an application is described below for sites representing shops), but usually they are assumed to be disjoint. In addition, infinitely many sites are allowed in the definition (this setting has applications in geometry of numbers and crystallography), but again, in many cases only finitely many sites are considered.

In the particular case where the space is a finite-dimensional Euclidean space, each site is a point, there are finitely many points and all of them are different, then the Voronoi cells 3.1 are convex polytopes and they can be represented in a combinatorial way using their vertices, sides, two-dimensional faces, etc. Sometimes the induced combinatorial structure is referred to as the Voronoi diagram 3.2. However, in general the Voronoi cells may not be convex or even connected.

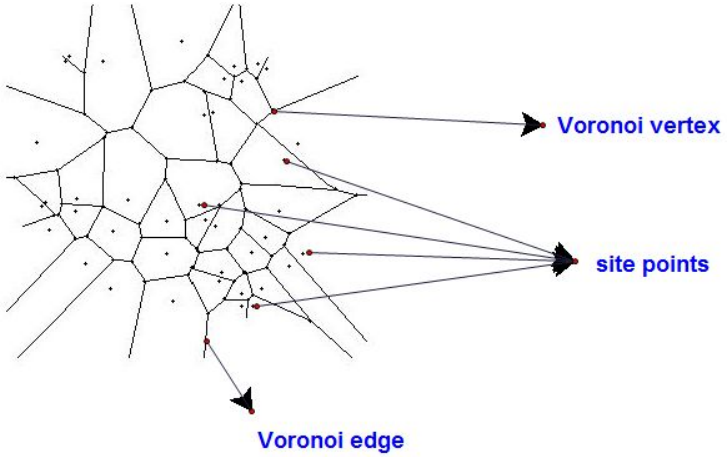


Figure 3.1. Voronoi cells.

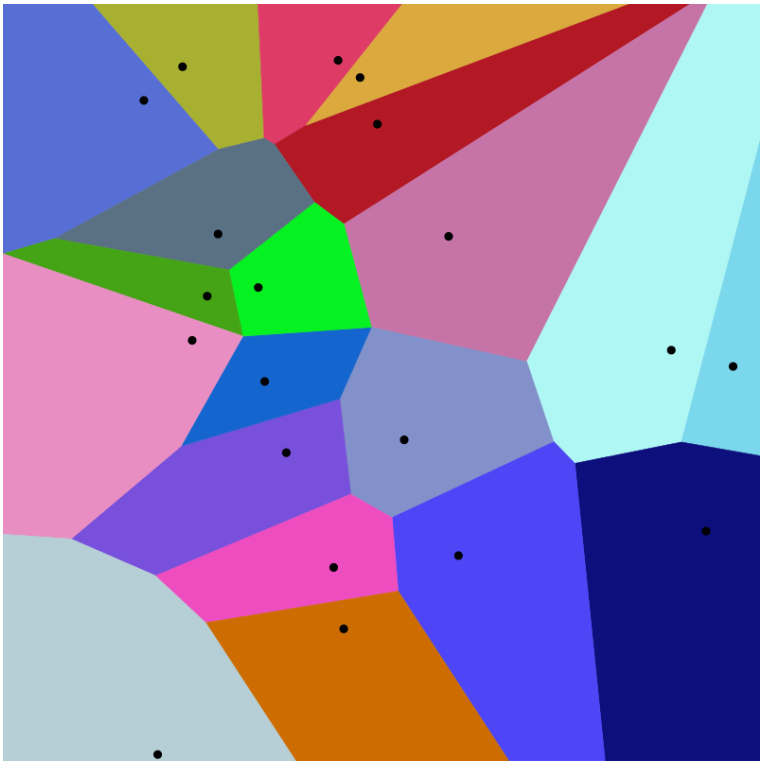


Figure 3.2. Voronoi diagram.

An application of the Voronoi Diagram is the trajectory optimization and smoothing for an UGV (Unmanned ground vehicle) path planning as reported in this paper. In the Figure 3.3 we can see the Voronoi Diagram cells with grey lines, the obstacles on the path that are the green objects and the source and the destination of the UGV's path that the two blue stars. The trajectory is tracked on the Voronoi edges and vertices and the result is a continuous trajectory[10].

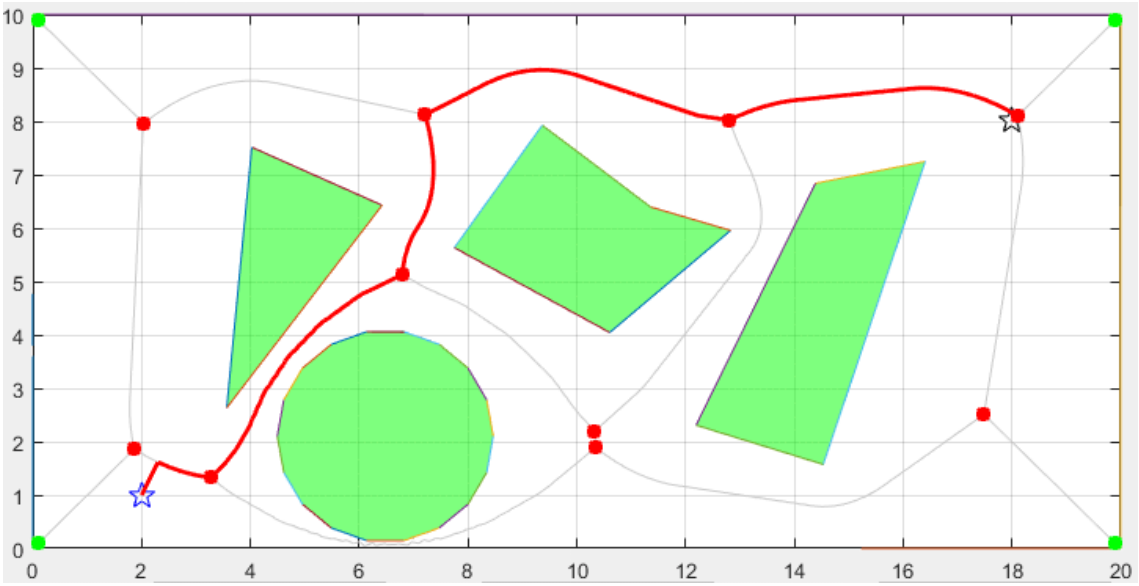


Figure 3.3. Voronoi diagram-based trajectories within two different homotopies. The path evaluation applies criteria of distance to obstacles (weight 80%) and the line-of-sight time (weight 20%) .

Another mathematical models is the Spline function. a spline is a special function defined piecewise by polynomials. In interpolating problems, spline interpolation is often preferred to polynomial interpolation because it yields similar results, even when using low degree polynomials, while avoiding Runge's phenomenon for higher degrees. In the computer science subfields of computer-aided design and computer graphics, the term spline more frequently refers to a piecewise polynomial (parametric) curve. Splines are popular curves in these subfields because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes through curve fitting and interactive curve design. The term spline comes from the flexible spline devices used by shipbuilders and draftsmen to draw smooth shapes.

In the paper [12] a set of techniques in robot navigation are reported, among which the spline function. The figure 3.4 represented in the paper shows a spline function applied on the x and y coordinates. The green path is the global path while the red path is the smooth path. The green dots are the control points.

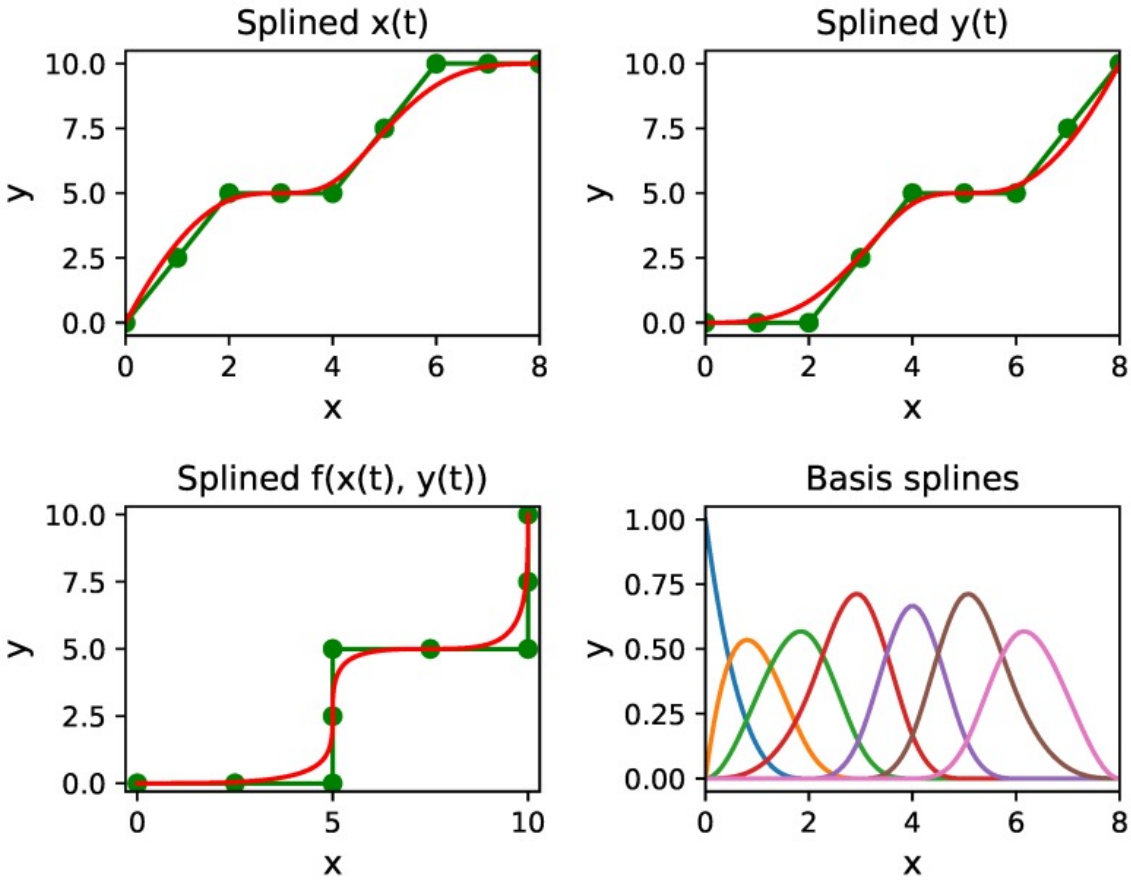


Figure 3.4. An example of path smoothing using B-Spline.

3.2 Design of the simulator

We have developed a simulator for reproducing the trajectories of the paths of each user inside a smart house. The simulator is a stand-alone software. It creates a log of entries of type <timestamp, position> for each user that moves inside the house. We have designed the simulator following a clear architecture.

3.2.1 The architecture

The architecture of the simulator is composed by three layers and one tier as in Figure 3.5.

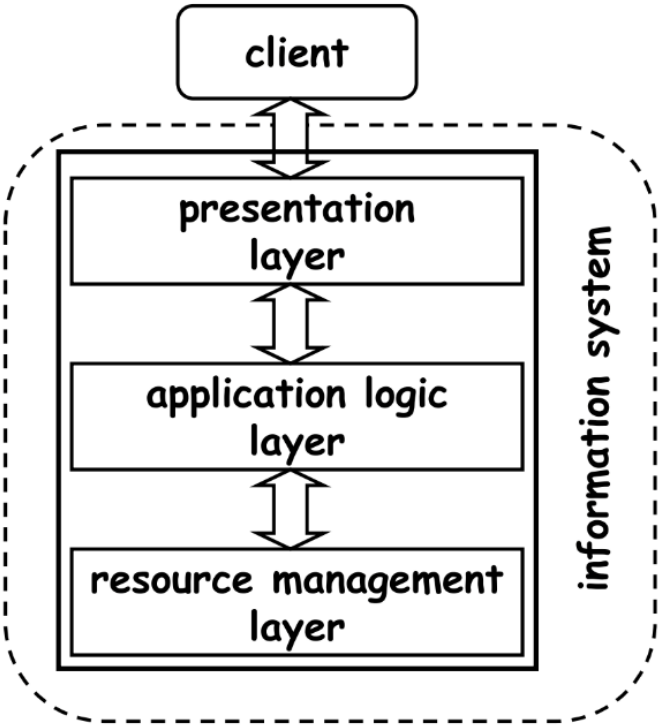


Figure 3.5. The one-tier architecture.

The logical layers are the presentation layer, the application logic layer and the resource management layer. Clients interact with the software through the presentation layer: this layer reproduces the front-end with the user interface. The application logic layer is the engine of the software: it is represented by the algorithms that run in back-end. The resource manager deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application logic. This is typically a database but it can also be a text retrieval system or any other data management system providing querying capabilities and persistence. In this case, the resource management layer is not a database but it's composed by several logs, that contain information about the topological features of the smart house. The three layers reside in one physical tier that is the personal computer. The one-tier in Figure 3.5 is showed as the continuous line that contains the three layers. Below we follows a top-down analysis of the entire architecture starting from the presentation layer.

3.2.2 The user interface

The user interface has been designed based on the tasks of the simulator. Mainly we have three sets of activity in the program: the simulation of user's trajectories, the graphing of the trajectories and the building of the 2D map of the smart house. Following this model, we have organized the UI in three part as in Figure 3.6.

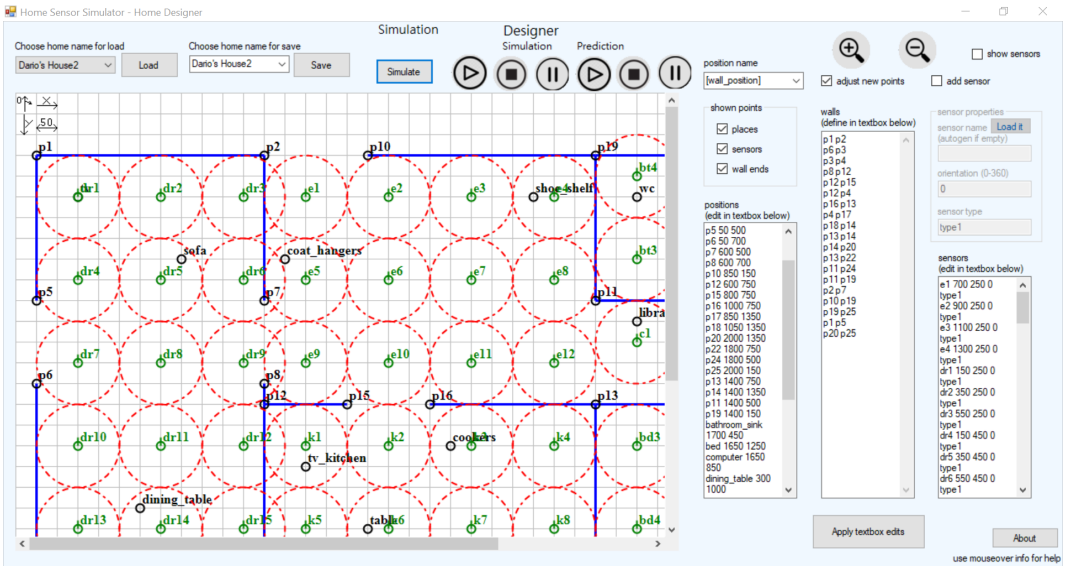


Figure 3.6. The graphical user interface of the simulator.

We analyse every part of the user interface below.

The first part concerns the building of the house's map. There are three text boxes: one dedicated to set up the positions, one for the walls and the last for the sensors. The text box for the positions allows to sets up the position of the crossing points of the walls and the targets: indeed there are the name of the position and the coordinates x and y, e.g. p1 50 150. The text box for the wall contains a set of couples of positions; indeed a wall is a line delimited by two points, e.g. "p1 p2". The last text box is dedicated to the sensors: there are the name of the sensor and the position, for example "e1 700 250". The buttons above "+" and "-" acts as zoom to the map of the house. The check box "show sensors" shows the sensors in the 2D map of the house. The other check boxes shows places and walls.

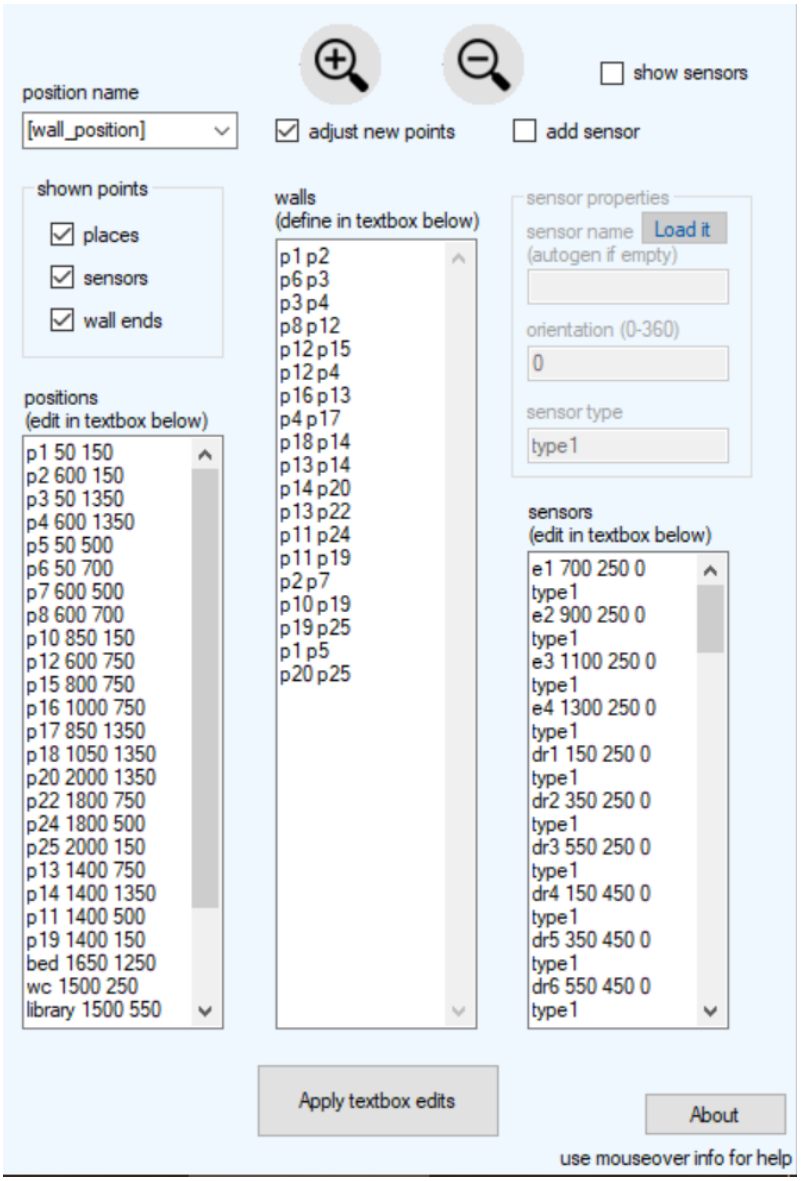


Figure 3.7. The commands for building and customizing the house.

The second container of the user interface includes the 2D map where the simulator draws the trajectories. The walls are represented with blue colour, the cone of the shadow of the sensor with dotted red lines and the targets with black points. The length of a unit on the x axis is 0.5 meters and it is the same for the y axis. It is represented in Figure 3.8. The map below has the PIR sensors with cone of shadow with a radius with one meter.

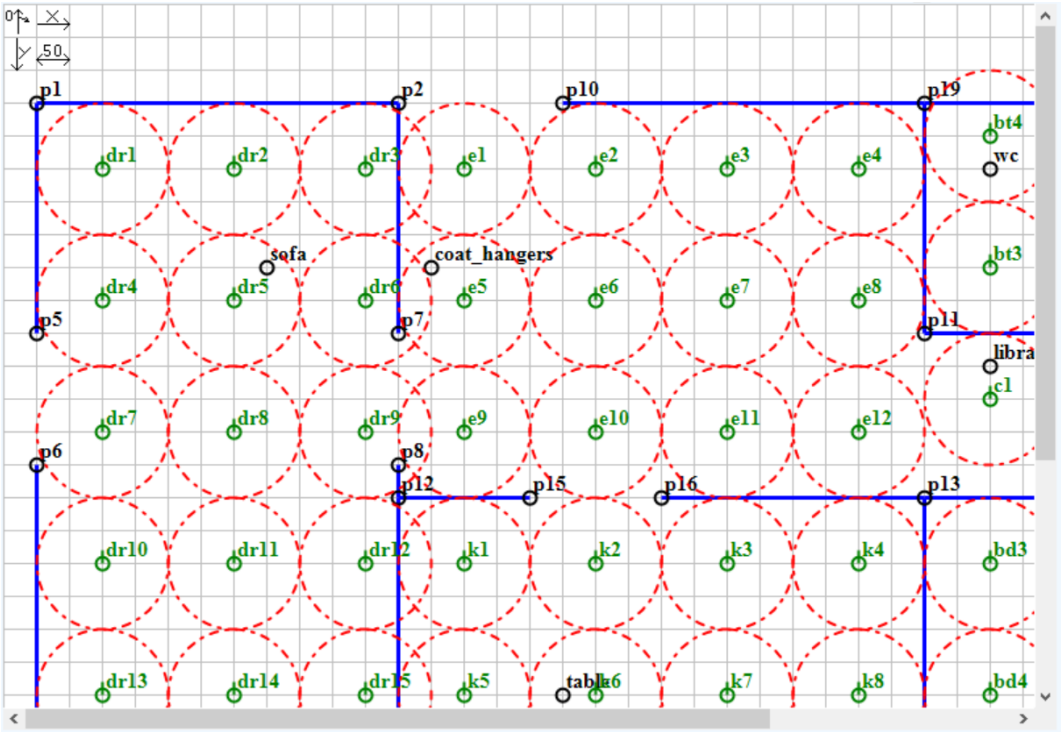


Figure 3.8. The 2D map of the smart house with the sensors with cone shadow of one meter in red for each room and the targets(e.g. the sofa, the table).

We can also set up another configuration with PIR sensors with cone of shadow with a radius of two meters.

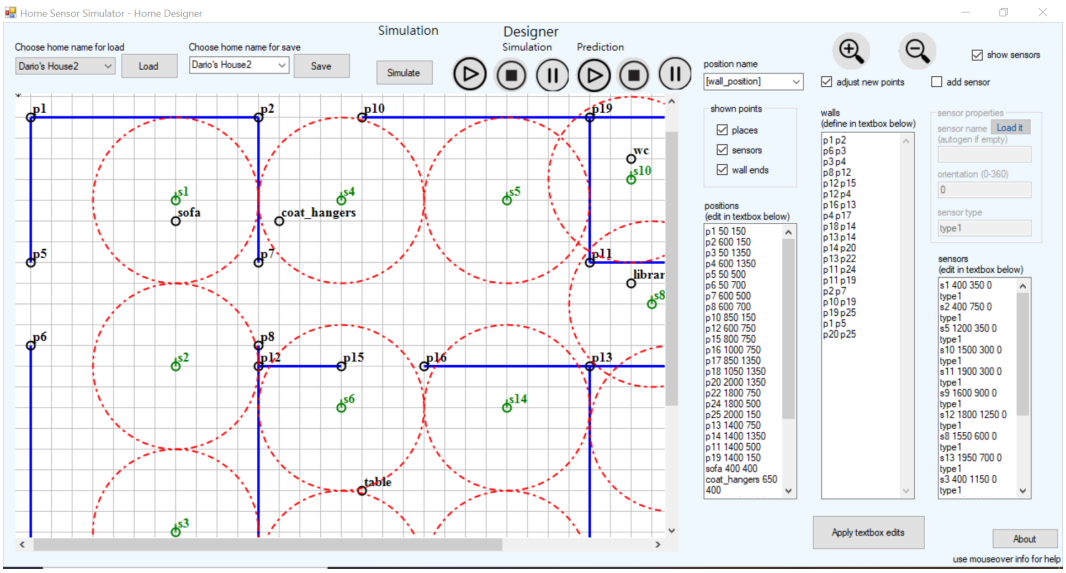


Figure 3.9. The 2D map of the smart house with the sensors with cone shadow of two meters in red for each room and the targets(e.g. the sofa, the table).

The last part contains the commands for the simulation: the selection of the house and the simulation form. It also contains the player for drawing both the simulated trajectories and predicted trajectories. This is showed in Figure 3.10.

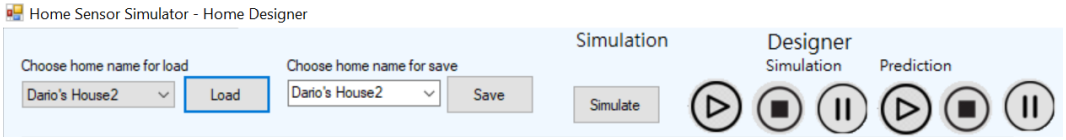


Figure 3.10. The commands for the simulation and the player.

At the left side there are two drop down menus for loading a model of house in the 2D map and saving a built house. At the right the panel is populated by two players: one for drawing in real time the simulated trajectories and the other for designing the predicted trajectories. The two players have the buttons for playing, pausing and stopping the graphic representation of the trajectories in real time. In each part of the user interface we use several control for interacting with the client. Below we report some examples of these controls: in order there are a Button, a TextBox, a ComboBox, a CheckBox and a Label.

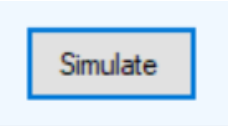


Figure 3.11. The button for the simulation.

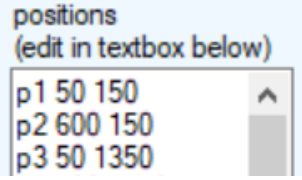


Figure 3.12. The TextBox where to set up the positions.

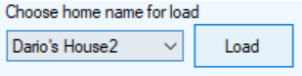


Figure 3.13. The ComboBox where to choose the house's model.

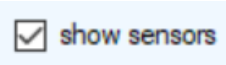


Figure 3.14. The CheckBox that shows the discrete sensors.

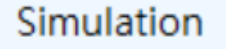


Figure 3.15. The Label of the simulation.

The trajectory of each user has a different colour. It starts from a source target and arrives in a destination target. We have four users, one with yellow trajectory, the second with red, and the others with black and brown trajectories. The four users move with a constant speed of 0.5 m/s. In the two figures below we can notice that each trajectory is **curvilinear** and **smoothed**. This effect is given by the implementation of the **Spline functions**, that performs the **interpolation** between each couple of positions. We show below several pictures representing examples of continuous (**dense**) trajectories and discrete (**rarefied**) trajectories. As we have explained in the second chapter with the definition of trajectory a discrete trajectory is a broken line.

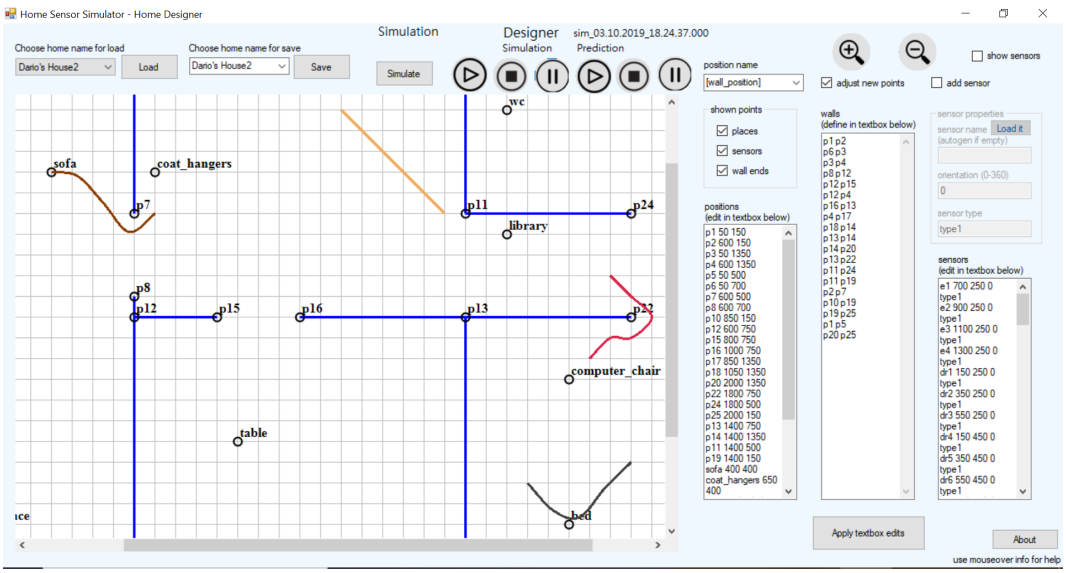


Figure 3.16. An example in real time of the graphic representation of four trajectories.

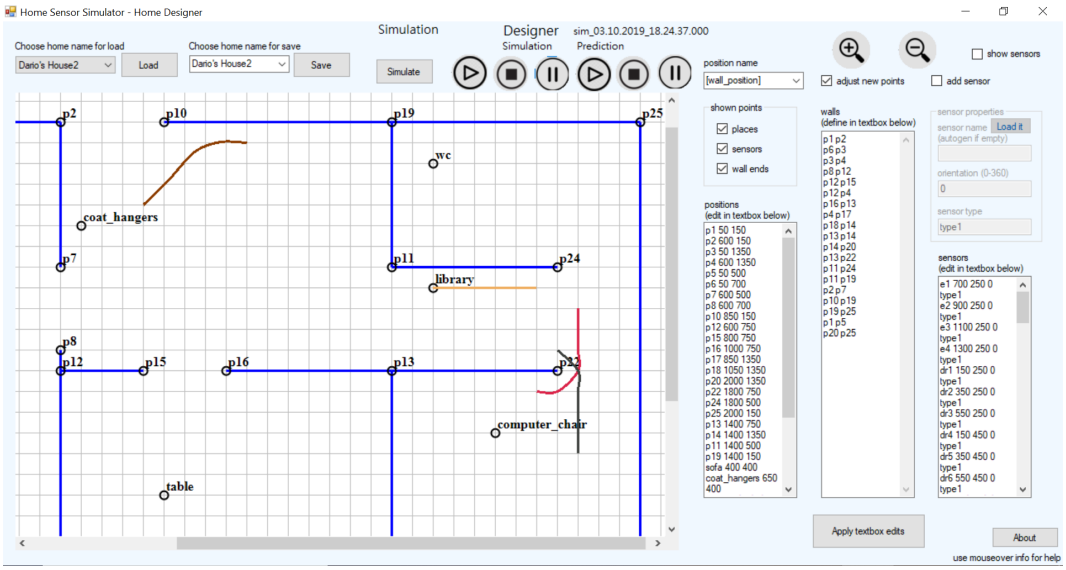


Figure 3.17. An example in real time of the graphic representation of four continuous of which two crossing trajectories.

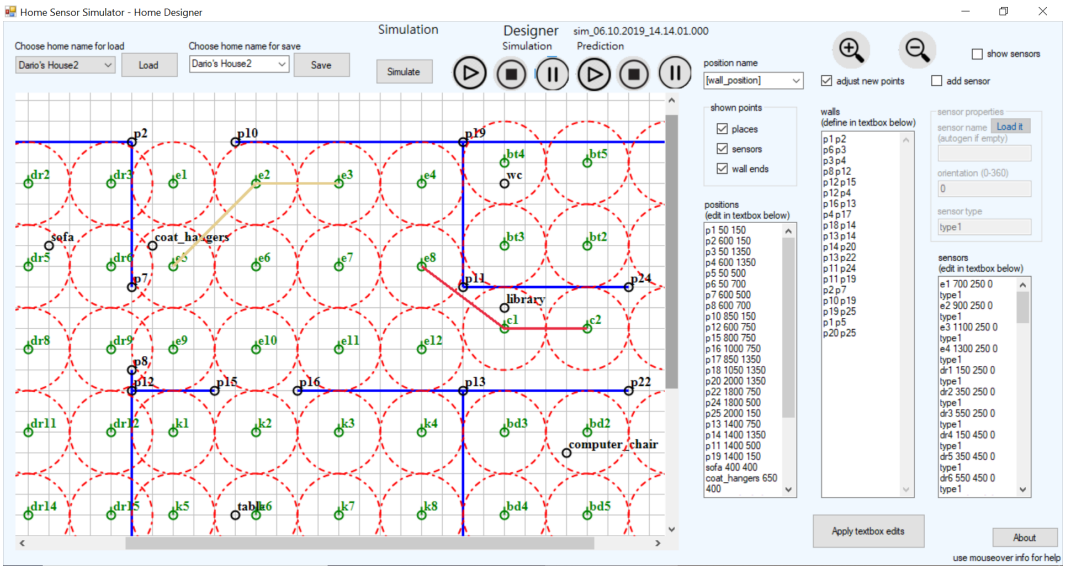


Figure 3.18. An example in real time of the graphic representation of two discrete and rarefied trajectories generate by the triggering of the PIR sensors due to the motion of the two users; one is red and the other is yellow. The PIR sensors in the map are the green points, and their area are the red dashed lines.

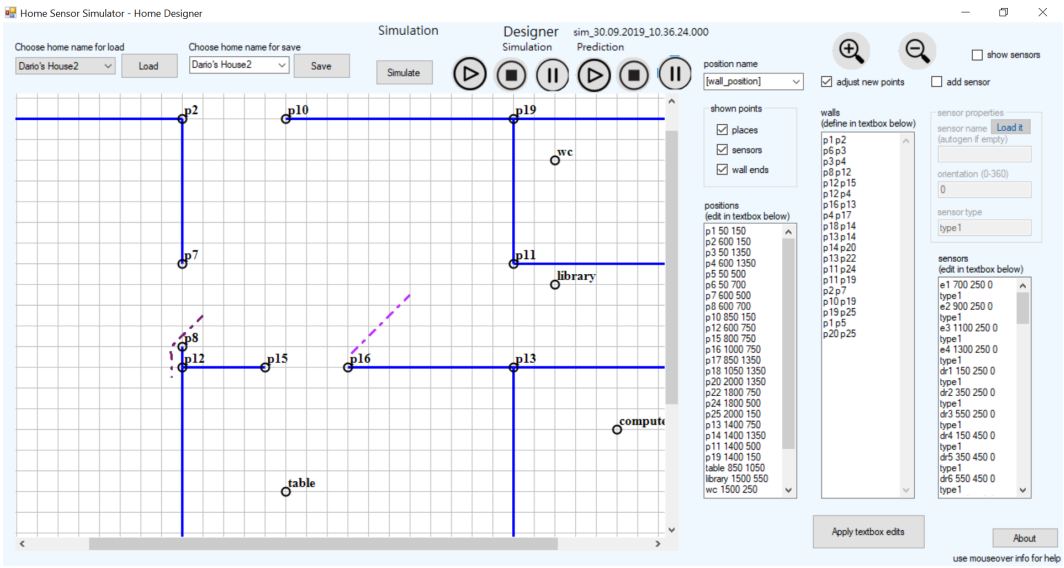


Figure 3.19. An example in real time of the graphic representation of two continuous trajectories with dashed lines predicted by the Kalman filter.

3.2.3 The application logic layer

The application logic layer is the engine of the simulator that runs in the back-end. The main tasks of the simulator are:

- Simulating the trajectory of each user that moves inside the house for several days.
- Graphing these trajectories for showing the simulation done and drawing the predicted trajectories given from an algorithm of prediction which we will discuss in the next chapter.

Here we show the activity diagram of a simulation.

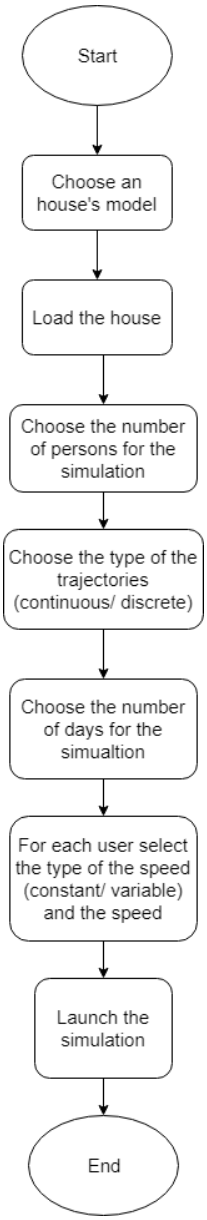


Figure 3.20. Activity Diagram of a simulation.

The graphic representation of the predicted trajectories and its implementation is the same of the simulation: we will only discuss for the graphic representation of the simulation. First of all, the client chooses a 2D model of the house. Then it selects the number of the persons for the simulation that move inside the house. The client sets up the type of trajectories,continuous or discrete, the number of the days of the simulation. Finally it selects the type of the speed, constant or variable, and since the speed is constant it chooses the value of the speed. With regard to the simulation, each user moves inside the house from a source target (habitual object) to a destination target tracking a path. We have designed an algorithm respecting this model. We describe the pseudocode of this algorithm below.

Algorithm Simulation's algorithm

Timer = 00:00:00
SourceTarget = random()
Path = ∅

while Timer <= 23:59:59
    DestinationTarget = random()
    Path = pathfinding(**SourceTarget**, **DestinationTarget**)
    write **Path** in the log in the form of <timestamp, position>
    Path = ∅
    SourceTarget = **DestinationTarget**
    DestinationTarget = random()
    increment **Timer**

This algorithm returns in output a log file with the trajectories of each user. While the timestamp doesn't arrive at midnight, the algorithm chooses a new random destination target and finds the minimum path with a pathfinding algorithm. We use the BFS pathfinding algorithm (Breadth-First Search). The path is constituted by entries of this type, <timestamp, position> according to the Definition of Sensor Log. In the case of continuous trajectories, the entries have the form of <timestamp, x y coordinates>. In the case of discrete trajectories, the algorithm writes in the logs <timestamp, sensor name, sensor status>; the sensor status is ON or OFF. The Simulation's algorithm is launched by the software one or more time based on the setted days of the simulation.

The graphic representation of the trajectories in the simulator follows this algorithm.

Algorithm Drawing of the trajectories

Input: The list of the trajectories of the users

Timer = 00:00:00

while Timer < 00:00:10
    for all couple of adjacent positions <p1, p2> in trajectory$_i$
        draw <p1, p2> line in the map
        increment **Timer**

while Timer <= 23:59:59
    for all couple of adjacent positions <p1, p2> in trajectory$_i$
        cancel the first lines of each user's trajectory in the map
        draw <p1, p2> line in the map
        increment **Timer**

This algorithm takes in input the list of the trajectories of the users read from the logs. It starts a timer and until the first ten seconds it draws the trajectory of each user. After this, the algorithm begins to cancel the old first drawn lines of each trajectory and then draws the new lines.

3.2.4 The resource management layer

The resource management layer is part of the back-end of the software. Typically it consists of a database but in this case we have several log files. Mainly we have these logs: the "house" log, the "obligatory places" and the logs of the simulations. The log "house" is a text file divided in several parts:

- the positions of the intersections of the walls.
- the positions of the targets.
- the walls that are tuples of points, that form a line.
- the positions of the sensors.

The obligatory places log is a text file that contains the minimum targets that the user must select for building the smart house. In this way, the client of the simulator can customize his smart home with his favourite features.

3.2.5 Project structure

The project structure is designed as in Figure 3.21 below.

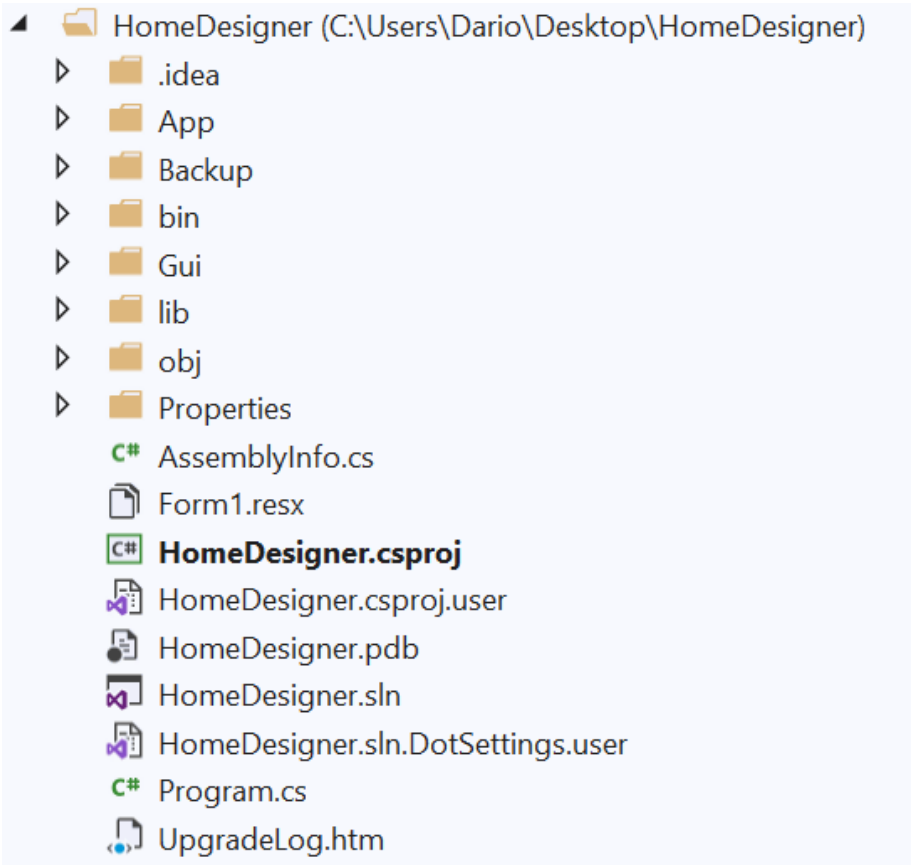
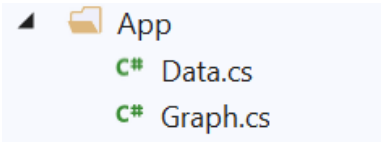


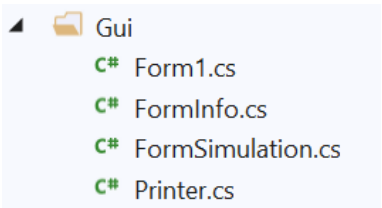
Figure 3.21. Project structure of the "HomeDesigner" Simulator.

Below we analyse the main directories and its components.

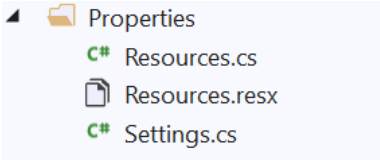
HomeDesigner/App



This folder contains the classes for the application logic of the software. **HomeDesigner/Gui**



This folder contains the classes for the user interface. Each class extends a Windows Form. **HomeDesigner/Properties**



The directory Properties has two classes, Setting and Resources. Settings acts as a base class from which to derive concrete wrapper classes for implementing the application settings functionality in Windows Forms applications. Resources class represents a resource manager that provides convenient access to specific resources of the culture settings during the execution phase.

3.3 Implementation of the simulator

The software implementation follows the design and the architecture of the simulator. In this way, we have divided this paragraph like the model of the architecture. Each layer is explained in every implementation detail.

3.3.1 Technologies

We have used C# programming language for the development of the simulator. C# is an object-oriented programming language developed by Microsoft within the .NET initiative. The three layers, namely the presentation layer, the application logic layer, the resource management layer, have been implemented in C#, using the .NET libraries. The simulator is a stand-alone software and it runs on Windows 10 OS.

We have developed the software using Visual Studio 2019 IDE[2]. Microsoft Visual Studio (or more commonly Visual Studio) is an integrated development environment (Integrated development environment or IDE) developed by Microsoft. Visual Studio is multilingual and currently supports the creation of projects for various platforms (including Mobile and Console). Extensions and add-ons can be created and used.

3.3.2 The user interface

In the user interface's implementation we have used several graphic components. We have organized the main screen with the Form Class. A Windows Form represents a window or dialog box that makes up an application's user interface. Inside the form we have implemented the various control components:

- Button represents a Windows button control, which responds to the Click event.
- TextBox represents a control that can be used to view or edit unformatted text.
- CheckBox allows the user to choose from a list of options.
- Label, generally used to provide a descriptive text.
- GroupBox represents a Windows control that displays a frame around a group of controls with an optional caption.
- ComboBox is a drop-down menu with selectable items.
- PictureBox, represents a control for displaying an image.
- ToolTip is a small rectangular popup window that appears with a brief descriptive hint.
- Panel, used to group control collections.
- Bitmap is an object used to work with images defined by pixel data.

- IContainer is an interface that provides functionalities for containers. Containers are objects that logically contain zero or more components.

These controls are attributes of the Form class. In the Figure 3.3.2 below we show a piece of code of this for an example.

```
public class Form1 : Form{

private IContainer components = (IContainer) null;
private PictureBox pictureBox1;
private Label label1;
private TextBox posTextBox;
private Label label2;
private Label label3;
private TextBox wallTextBox;
private Button open;
private Button save;
private Button disegna;
private Button disegnaStop;
private Button predizioneStop;
private Label labelSim;
private Label labelPred;
private CheckBox adjustCheckBox;

private Button pausePredizioneButton;
private Button pauseSimulazioneButton;

private Button simula;
private Button zoom;
private Button zoomMeno;
private Button disegnaPredizione;
private Button doEdits;
private CheckBox sensorCheckBox;
private TextBox orientationtb;
private TextBox sensortypetb;
```

The controls are object initialized in the *"InitializeComponent()"* function. This function makes two things: initialize the controls and adds the specified control to the control collection of this Form with *"this.Controls.Add(Control)"* as shown in Figure 3.3.2. The other functions in the Form class include listener for implementing the behaviour of the controls like buttons.

```
   private void InitializeComponent(){

        this.WindowState = FormWindowState.Maximized;
        this.components = (IContainer) new Container();
        ComponentResourceManager componentResourceManager =
        new ComponentResourceManager(typeof (Form1));
```

```
7           this.label1 = new Label();
8           this.posTextBox = new TextBox();
9           this.label2 = new Label();
10          this.label3 = new Label();
11          this.wallTextBox = new TextBox();
12          this.open = new Button();
13          this.save = new Button();
14          this.disegna = new Button();
15          this.disegnaPredizione = new Button();
16          this.disegnaStop = new Button();
17          this.predizioneStop = new Button();
18          this.pausePredizioneButton= new Button();
19          this.pauseSimulazioneButton= new Button();
20          this.simula = new Button();
21          this.zoom = new Button();
22          this.zoomMeno = new Button();
23          this.showSensors = new CheckBox();
24
25          this.labelSim=new Label();
26          this.labelPred=new Label();
27
28          this.doEdits = new Button();
29          this.pictureBox1 = new PictureBox();
30          this.adjustCheckBox = new CheckBox();
```

We have developed three Form classes for several panels: the main panel, the panel of the form for the simulation and the panel for setting up the graphic representation of the trajectories. The main panel is shown in Figure 3.6. Below there are the panels of the simulation's settings and the graphic representation of the trajectories.

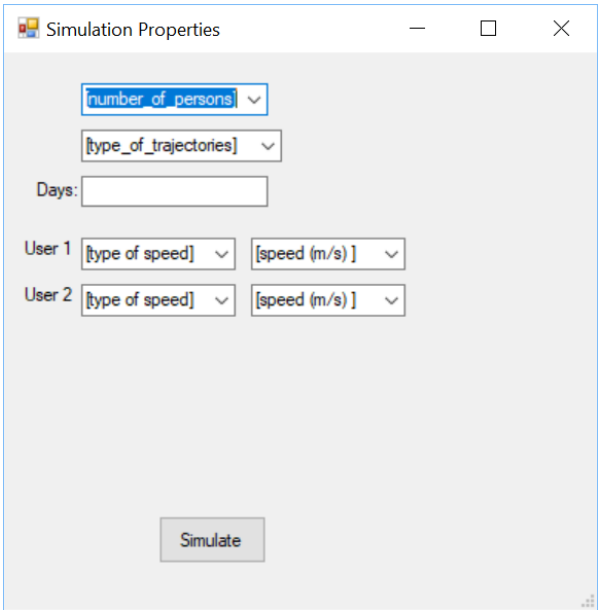


Figure 3.22. Simulation Properties panel.

The panel in Figure 3.22 allows to set up the properties of a simulation: the number of persons, the type of trajectories, the speeds through ComboBoxes and the duration of the days of the simulation through a TextBox.

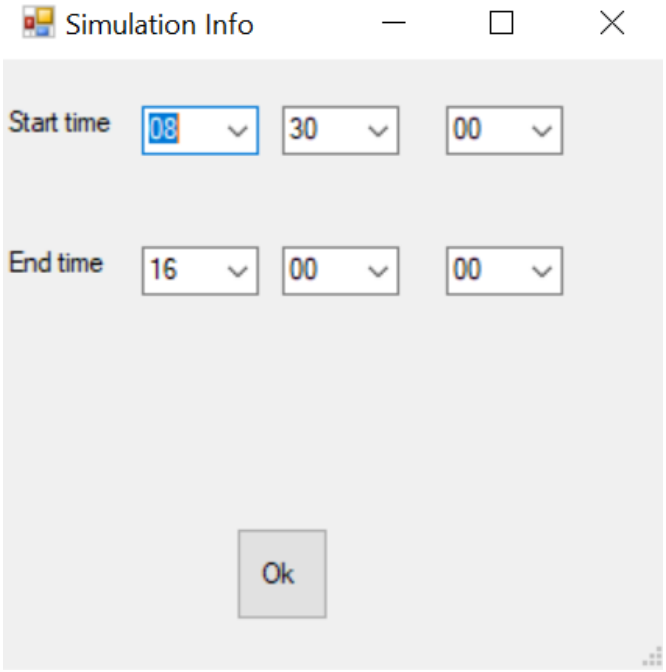


Figure 3.23. Simulation Info panel.

The panel in Figure 3.23 allows to decide what intervals of time the simulator must show during the graphic representation of the trajectories.

3.3.3 The application logic layer

We have already analysed the design and the algorithm of this part of the application. In this paragraph we describe the implementation details of the algorithms used in this layer. We remember that the Simulation's Algorithm generates a log file of entries of the type <timestamp, position> for each user.

3.3.3.1 Functions of the "Simulation's algorithm"

The two main functions for implementing this algorithm are *"SimulationHandler()"* and *"GeneratePathLog()"*. The first function takes in input a list of user, and for each user the moving speed the type of the trajectory (continuous or discrete) and the number of the simulation's days. This function makes the following tasks: it creates the list of the targets ,namely habitual objects inside the house, and then for each user it calls the *"GeneratePathLog()"* function. The *"GeneratePathLog()"* takes in input the list of the target objects, the type of trajectory, the speed. In the first the function creates a direct weighted graph with vertices and edges based on the parameter of the speed: each edge is long as the value of the speed. Every edge has weight of value one. For example, since the speed is 0.5 m/s, the edges of the graph are long 0.5 meters. Each vertex is the intersection of several edges. In the Figure 3.24 below, we show the creation of the object Graph. The parameters of the constructor are the list of the nodes and the list of the edges. After there is the beginning of the pathfinding algorithm. We have used the Generics for the implementation of the Graph class. The type of data used in this case is an integer and so every node has represented as integer number in the graph.

```
1  var graph = new Graph<int>(nodes, edg);
2  TextWriter log = (TextWriter)new StreamWriter(pathdir + "\\PathLog" + u + ".txt");
3  var now = DateTime.Now;
4  DateTime time = new DateTime(now.Year, now.Month, now.Day, 0, 0, 0, 0);
5  Position source, dest = null;
6  int indice = 0;
7  DateTime maxDate = time.AddDays(giorniSimulazione);
8  maxDate.AddHours(23);
9  maxDate.AddMinutes(59);
10 maxDate.AddSeconds(59);
11 maxDate.AddMilliseconds(0);
12
13 while (time!=maxDate){
14         if (time == maxDate){
15                 log.Close();
16                 return;
17         }
18
19
20         int s1 = 0;
21         Random random = new Random();
22         int idOggetto = random.Next(0, 5);
```

```
23          if (indice == 0){
24                  indice = 1;
25                  source = listaOggetti[idOggetto].Value;
26                  s1 = getNodi(source, d1);
27          }
28          else{
29
30                  source = dest;
31                  source.x = dest.x;
32                  source.y = dest.y;
33
34                  s1 = getNodi(source, d1);
35
36          }
37
38          dest = listaOggetti[4 - idOggetto].Value;
```

Figure 3.24. The creation of the object Graph.

The second part of the *"GeneratePathLog()"* function concerns the pathfinding algorithm. We have used the BFS algorithm (Breadth-first search). In our case, BFS has been implemented for searching the shortest path from a source target to a destination target. The algorithm of the simulation runs until it doesn't reach the time of 23:59:59 of the max simulation date. In this cycle it chooses a random source target and a random destination target; note that the random the source and the destination doesn't never coincide and so edge loops don't exist. The function of the pathfinding is the *"ShortestPathFunction()"*. It takes in input the Graph object and the source and returns a list of the nodes, that is the shortest path. After the search, it writes the path in form of <timestamp, position> in the log, that is a text file. In the Figure 3.25 there is a piece of code of the algorithm of the simulation described. In the next Figure 3.26 we show the implementation of the BFS algorithm.

```
int iddest = listaOggetti[4 - idOggetto].Key;
int dg = getNodi(dest, d1);
var startVertex = s1;
var shortestPath = ShortestPathFunction(graph, startVertex);
var d = dg;

List<int> path = shortestPath(d).ToList();
if (path.Count == 1){
continue;
}
for (int g = 0; g < path.Count; g++){

        if (velocitaVariabile){
                if (time.ToString("HH:mm:ss.fff").Equals("23:59:59.000")){

                        if (contaGiorni == giorniSimulazione){
                                log.Close();
                                return;
                        }
                        contaGiorni++;
                }
```

Figure 3.25. A part of the algorithm of the simulation.

```
public Func<T, IEnumerable<T>> ShortestPathFunction<T>(Graph<T> graph, T start) {
        var previous = new Dictionary<T, T>();
        var queue = new Queue<T>();
        queue.Enqueue(start);

        while (queue.Count > 0) {
                var vertex = queue.Dequeue();

                foreach(var neighbor in graph.AdjacencyList[vertex]) {

                if (previous.ContainsKey(neighbor))
                        continue;

                        previous[neighbor] = vertex;
                        queue.Enqueue(neighbor);
                }
        }

        Func<T, IEnumerable<T>> shortestPath = v => {
                var path = new List<T>{};

                var current = v;
                while (!current.Equals(start)) {
                        path.Add(current);
                        current = previous[current];
                };

                path.Add(start);
                path.Reverse();

                return path;
        };

        return shortestPath;
}
```

Figure 3.26. The implementation of the BFS algorithm.

3.3.3.2 Data structures of the "Simulation's algorithm"

The main data structures of the simulation are the graph, the adjacency matrix, the list of nodes, the list of edges. The graph is implemented as a class with a construct that takes in input the list of vertices and the list of edges. The attribute of this class is the adjacency matrix that is a dictionary with key a node and value an hashset that is the set of the adjacent nodes to the key. In the Figure 3.27 there is the implementation of the Graph class.

```
1  public class Graph<T> {
2          public Graph() {}
3          public Graph(IEnumerable<T> vertices, IEnumerable<Pair<T,T>> edges) {
4          foreach(var vertex in vertices)
5                  AddVertex(vertex);
6
7          foreach(var edge in edges)
8                  AddEdge(edge);
9          }
10
11         public Dictionary<T, HashSet<T>> AdjacencyList { get; } =
12         new Dictionary<T, HashSet<T>>();
13
14         public void AddVertex(T vertex) {
15                 AdjacencyList[vertex] = new HashSet<T>();
16         }
17
18         public void AddEdge(Pair<T,T> edge) {
19                 if (AdjacencyList.ContainsKey(edge.First) &&
20                  AdjacencyList.ContainsKey(edge.Second)) {
21                         AdjacencyList[edge.First].Add(edge.Second);
22                         AdjacencyList[edge.Second].Add(edge.First);
23                 }
24         }
25 }
```

Figure 3.27. The Graph class.

3.3.3.3 Functions of the algorithm "Drawing of the trajectories"

This algorithm draws on the 2D map the trajectories of each user that moves inside the house. Initially it draws the first ten lines of each trajectory of the users. We remember that a line is a couple of different positions with different timestamps. The algorithm assigns to each trajectory a colour. After drawing the ten lines, the first line is deleted and a line is made: this is repeated until the end of the reading of the log. The functions involved in this algorithm are in order *"DrawSimulation()"*, *"DrawerHandler()"* and *"DrawerHandlerPastLines()"*. The *"DrawSimulation()"* function draws several components of the 2d map: the wall, the sensors and the targets inside the map of the house. It takes in input the PictureBox of the 2D

map, the initial and end datetime of the simulation, a dictionary with key the timestamp and the value is a map of with key the position and value as a colour: the dictionary represents is a supporting data structure that acts as a Sensor Log, for each timestamp corresponds a list of positions of all trajectory of the users.

```
public Bitmap DrawSimulation(Form form, String folderPath,
        PictureBox pictureBox1, bool drawPlaces, bool drawSensors, bool drawWalls,
        DateTime timeInizio, DateTime timeFine, float zoomFactor){
```

Figure 3.28. DrawSimulation function.

The function returns a Bitmap, consisting of the pixel data of an image and its attributes, and it calls the *"DrawerHandler()"* as a thread: for each different trajectory a new thread starts with this function associated. *"DrawerHandler()"* has the task of drawing the paths. Each couple of positions represents a line of a trajectory. Initially the function draws the first ten lines and every line appears at each seconds.

```
foreach (KeyValuePair<Position, Color> keyValuePair in list){

        ThreadStart threadDelegate = delegate { bitmap = DrawerHandler(keyValuePair,
         list2, graphics, pictureBox1, cancellaLinee, dictionary, timeInizioApp,
         incremento, bitmap, graphicsOriginal, drawPlaces, drawSensors,
         drawWalls, "simulation", incrementoTM); ; };
        Thread newThread = new Thread(threadDelegate);
        newThread.Start();

        pictureBox1.Invoke((MethodInvoker)delegate{

                pictureBox1.Image = (Image)bitmap;

                pictureBox1.Refresh(); //questo refresh serve per restiture
                //al gui thread un risultato parziale e temporaneo dell'immagine

        });

        newThread.Join();
        // ogni linea viene graficata ogni mezzo secondo

}
```

Figure 3.29. Call of the threads of the DrawerHandler().

```
  if (p1.x == p2.x && p1.y == p2.y) //posizioni uguali quindi utente fermo
  {
        Brush brush = new SolidBrush(keyValuePair.Value);

        graphics.FillRectangle(brush, (float)p1.x / 2f, (float)p1.y / 2f, 4f, 4f);

        }else{
```

```
        if (tipoGraficazione.Equals("prediction")){
                pen.DashCap = System.Drawing.Drawing2D.DashCap.Round;
                // Create a custom dash pattern.
                pen.DashPattern = new float[] { 4.0F, 2.0F, 2.0F, 2.0F };
        }
        graphics.DrawLine(pen, (float)p1.x / 2f, (float)p1.y / 2f,
        (float)p2.x / 2f, (float)p2.y / 2f);

}
```

Figure 3.30. Graphic representation of the trajectories.

After it calls the *"DrawerHandlerPastLines()"*. This last function deletes the first line of each trajectory and draw a new line; it continues until the read of the last positions of all trajectories. We have implemented this mechanism in this way:

1. When the *"DrawerHandlerPastLines()"* function is called, it redesigns all the map of the house with the walls, the sensors and the places but without the trajectories and sets up a new bitmap to the graphics component.

2. After this the function redraws all the trajectories starting from the second line until the eleventh line; after it goes to 1. and then it repeats starting from the third line to the twelfth line and so on.

This mechanism is showed in Figure 3.31.

```
bitmap = drawPicture(drawPlaces, false, drawWalls);
graphics = Graphics.FromImage((Image)bitmap);
timeInizioApp = timeInizioApp.AddMilliseconds(incrementoTM);
List<KeyValuePair<Position, Color>> list1;
List<KeyValuePair<Position, Color>> list2;
list1 = dictionary[timeInizioApp.ToString("HH:mm:ss.fff")];

for (int g = incremento; g < incremento + 10; g++)
{

        timeInizioApp = timeInizioApp.AddMilliseconds(1000);
        list2 = dictionary[timeInizioApp.ToString("HH:mm:ss.fff")];

        foreach (KeyValuePair<Position, Color> k in list1)
        {
                DrawerHandlerPastLines(k, list2, graphics, tipoGraficazione);
        }
        list1 = list2;

}
```

Figure 3.31. DrawerHandlerPastLines implementation.

3.3.3.4 Interpolation of the trajectories

The "DrawerHandler" and "DrawerHandlerPastLines" functions draw the trajectory of each user. For obtaining smoothed and curvilinear trajectories, we have applied the Spline functions that we have previously presented in the first section of the chapter "State of art".

```
1
2           Pen pen = new Pen(keyValuePair.Value, 3f);
3           Position p1 = entry.Key;
4
5           Position p2 = keyValuePair.Key;
6           Position p3 = keyValuePair3.Key;
7           Position p4 = keyValuePair4.Key;
8           Position p5 = keyValuePair5.Key;
9           Position p6 = keyValuePair6.Key;
10          if (p1.x == p2.x && p1.y == p2.y) //posizioni uguali quindi utente fermo
11          {
12                  Brush brush = new SolidBrush(keyValuePair.Value);
13
14                  graphics.FillRectangle(brush, (float)p1.x / 2f,
15                  (float)p1.y / 2f, 4f, 4f);
16
17          }
18          else
19          {
20                  if (tipoGraficazione.Equals("prediction"))
21                  {
22                          pen.DashCap = System.Drawing.Drawing2D.DashCap.Round;
23
24                          // Create a custom dash pattern.
25                          pen.DashPattern = new float[] { 4.0F, 2.0F, 2.0F, 2.0F };
26                  }
27
28
29
30
31                  var xvec = new DenseVector(new double[] { (double)(p1.x),
32                          (double)(p2.x ),
33                          (double)(p3.x ), (double)(p4.x),
34                          (double)(p5.x),(double)(p6.x) });
35                  var yvec = new DenseVector(new double[] { (double)(p1.y ),
36                          (double)(p2.y),
37                          (double)(p3.y ), (double)(p4.y),
38                          (double)(p5.y),(double)(p6.y)});
39
40                  var cs = CubicSpline.InterpolateNatural(xvec, yvec);
41                  var x = new DenseVector(5);
```

```
42                  var y = new DenseVector(x.Count);
43                  List<PointF> listaCalcolati = new List<PointF>();
44                  double stepSize = (xvec[xvec.Count - 1] - xvec[0]) / (5 - 1);
45                  bool flagNaN = false;
46                  for (int i = 0; i < x.Count; i++)
47                  {
48                          x[i] = xvec[0] + i * stepSize;
49                          y[i] = cs.Interpolate(x[i]);
50                          if (Double.IsNaN(y[i]))
51                          {
52                                  graphics.DrawCurve(pen, new PointF[]{
53                                          new PointF((float)p1.x / 2f,(float)p1.y/2f),
54                                          new PointF((float)p2.x/2f,(float)p2.y/2f),
55                                          new PointF((float)p3.x/2f,(float)p3.y/2f),
56                                          new PointF((float)p4.x/2f,(float)p4.y/2f),
57                                          new PointF((float)p5.x/2f,(float)p5.y/2f),
58                                          new PointF((float)p6.x/2f,(float)p6.y/2f)
59                                  });
60                                  flagNaN = true;
61                                  break;
62                          }
63
64
65                          PointF pointF = new PointF((float)x[i] / 2,
66                          Math.Abs((float)y[i] / 2));
67                          listaCalcolati.Add(pointF);
68
69                  }
70                  if (flagNaN == false)
71                  {
72                          graphics.DrawCurve(pen, listaCalcolati.ToArray<PointF>());
73
74                  }
75
76          }
```

Figure 3.32. Interpolation and the drawing of the trajectories.

As we can see in the code, we take six different positions, that are the points of the trajectories. The goal is to generate between each of these points a set of positions for smoothing the trajectory.

We use a .Net library for implementing it, **Math.NET**. Math.NET is an open-source initiative to build and maintain toolkits covering fundamental mathematics, targetting advanced but also every day needs of .Net developers.

First of all for each position and its x coordinate we instantiate a DenseVector. The constructor of the DenseVector as showed, creates a new dense vector directly binding to a raw array. The array is used directly without copying; it's very efficient, but changes to the array and the vector will affect each other. The two array of the

x and y coordinates input into the function CubicSpline.InterpolateNatural(). This function creates a natural cubic spline interpolation from an unsorted set of (x,y) value pairs and zero second derivatives at the two boundaries. Then the algorithm instantiates other two dense vectors, var x and y, of length five. The x vector is populated with the xvec array recalculated. The y vector is populated applying the function Interpolate(): it takes in input a point p to interpolate and returns an interpolated value x(p). The new coordinates (x[i], y[i]) of the calculated point are added to a list and then drawn thanks to the function DrawCurve. This function draws the spline curve into the 2D map of the house.

3.3.3.5 Data structures of the algorithm "Drawing of the trajectories"

The most import data structure of this algorithm is the dictionary in input to the *"DrawSimulation()"* function as in Figure 3.33. This data structure is populated before calling the *"DrawSimulation()"* and it contains the logs of all users. The dictionary has the form "Dictionary <string, List< KeyValuePair<Position, Color> > > dictionary": each entry has a key that is the string of the timestamp, a value that is a list of key-value pairs namely a map. Each entry of the map has a key that is the position inside the 2D map of the house and the value is a colour assigned to the trajectory. In this way for each timestamp of the dictionary, we read the list that contains the positions of all the trajectories and the assigned colours. For each trajectory the algorithm chooses a colour and assigns it to each position. The *"DrawerHandler()"* matches two different positions with the same colour in the list of key-value pairs and finally it draws the line of the trajectory.

```
1 public Bitmap DrawSimulation(Form form, String folderPath,
2 PictureBox pictureBox1, bool drawPlaces, bool drawSensors, bool drawWalls,
3 DateTime timeInizio, DateTime timeFine, float zoomFactor){
```

Figure 3.33. DrawSimulation signature.

3.3.4 The resource management layer

The resource management layer is implemented with a series of logs. A log is a text file with the ".txt" format. We have three types of logs: the "house.txt", the "obligatoryplaces.txt", the "PathLog.txt". The "house.txt" log customizes the smart house in the simulator. It is divided in three sections: Positions, that are the points of the intersection of the walls and the positions of the targets, Walls namely a set of couples of positions and Sensors. In the Figure 3.34 below there is an example.

```
Positions
p1 50 150
p2 600 150
p3 50 1350
p4 600 1350
p5 50 500
p6 50 700
p7 600 500
p8 600 700
p10 850 150
p12 600 750
p15 800 750
p16 1000 750
p17 850 1350
p18 1050 1350
p20 2000 1350
p22 1800 750
p24 1800 500
p25 2000 150
p13 1400 750
p14 1400 1350
p11 1400 500
p19 1400 150
bed 1650 1250
wc 1500 250
library 1500 550
sofa 400 400
coat_hangers 650 400
table 850 1050

Walls
p1 p2
p6 p3
p3 p4
p8 p12
```

Figure 3.34. An example of the "house" log.

The "obligatoryplaces.txt" contains the selectable targets to add in the house. Regard to the logs of the simulation, they are contained in text files each for user called "PathLog.txt". In the case of continuous trajectories, the pathlog is organized in this way: timestamp, x and y coordinates as in Figure 3.35. In the case of discrete trajectories, the log has three fields in order: timestamp, name of the triggered sensor, status of the sensor as in Figure 3.36.

```
00:00:00.200 660 390
00:00:00.400 670 380
00:00:00.600 680 370
00:00:00.800 690 360
00:00:01.000 700 350
00:00:01.200 710 340
00:00:01.400 720 330
00:00:01.600 730 320
00:00:01.800 740 310
00:00:02.000 750 300
00:00:02.200 760 290
00:00:02.400 770 280
00:00:02.600 780 270
00:00:02.800 790 260
00:00:03.000 800 250
00:00:03.200 810 240
```

Figure 3.35. An example of the continuous PathLog.

```
2019-08-07      00:00:00.000000 dr5     ON
2019-08-07      00:00:02.000000 dr6     ON
2019-08-07      00:00:03.000000 dr6     ON
2019-08-07      00:00:05.000000 e5      ON
2019-08-07      00:00:06.000000 e5      ON
2019-08-07      00:00:07.000000 e5      ON
2019-08-07      00:00:09.000000 e2      ON
2019-08-07      00:00:10.000000 e2      ON
2019-08-07      00:00:11.000000 e2      ON
2019-08-07      00:00:13.000000 e3      ON
2019-08-07      00:00:14.000000 e3      ON
2019-08-07      00:00:15.000000 e3      ON
2019-08-07      00:00:17.000000 e8      ON
2019-08-07      00:00:18.000000 e8      ON
2019-08-07      00:00:19.000000 e8      ON
2019-08-07      00:00:21.000000 c1      ON
```

Figure 3.36. An example of the discrete PathLog.

3.3.5 Frameworks

We have used an external framework, dotConnect for SQLite, for the implementation of the sql queries to the database of the predicted trajectories. dotConnect for SQLite is a enhanced database connectivity solution built over ADO.NET architecture and a development framework with a number of innovative technologies. dotConnect offers a complete solution for developing database-related applications and web sites. It introduces new approaches for designing applications and boosts productivity of database application development. For the rest of the implementation we have used the API of .NET framework version 4.5. In the Figure 3.37 we show how the

simulator gets through dotConnect the data of the predicted trajectories from the external DB. Then these data are used to draw the predicted trajectories.

```
string mySelectQuery = "SELECT * FROM segmento";
SQLiteConnection sqConnection = new SQLiteConnection("DataSource="+filepath);
SQLiteCommand sqCommand = new SQLiteCommand(mySelectQuery,sqConnection);
sqConnection.Open();
SQLiteDataReader sqReader = sqCommand.ExecuteReader();
Random rand = new Random();
Dictionary<int, Color> idseg=new Dictionary<int, Color>();
try
{
        while (sqReader.Read())
        {
                Color color= Color.FromArgb(rand.Next(256),
                rand.Next(256),rand.Next(256));
                idseg.Add(sqReader.GetInt32(0), color);
        }

}
finally
{
        // always call Close when done reading.
        sqReader.Close();
        // always call Close when done reading.
        sqConnection.Close();
}
```

Figure 3.37. The query for reading predicted trajectories.

Chapter 4

Experimental setting for the continuous trajectories

In this chapter will be discussed the analysis in the continuous time system and the adopted algorithm. We introduce the theoretical aspects and the relative issues and then we show how to solve them and the results of the solution.

4.1 Introduction

The simulator "Home Designer" described in the third chapter outputs the trajectories of all users that move inside the smart house. We can have two type of the simulated trajectories: discrete and continuous trajectories.

In this chapter we analyse how to classify and distinguish the continuous and dense trajectory of each user in the continuous time system.

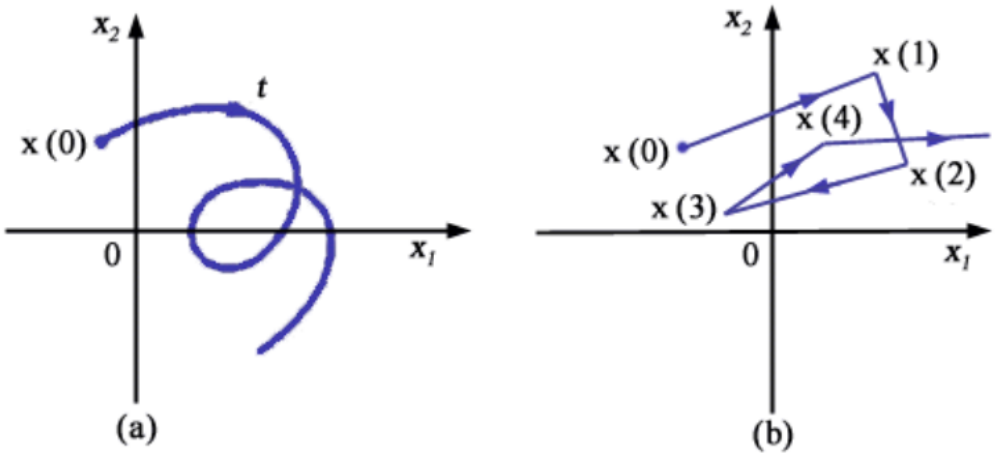


Figure 4.1. (a) Continuous trajectory, (b) discrete trajectory

A trajectory is the geometric line drawn in space from a moving point. The function x(•) identified is called movement, while the set {x(t), t≥0} in the state space R^n is called trajectory or orbit. In the case of continuous-time systems, the trajectory is

therefore a line rooted at point x(0) and with a precise direction of travel, that of time (see Figure 4.1a).

In our project, we remember that a continuous trajectory can be define as a set of points with a fine granularity. A **discrete trajectory** is defined as a sequence of positions in the space calculated at the triggering of the PIR sensors. The discrete trajectory is called as a trajectory with coarse granularity or **rarefied trajectory**. The main difference between the **continuous** and **discrete** trajectories is in the level of granularity of the data: **coarse** in the discrete case depending from the the triggering of the sensors and **fine** in the continuous. A continuous trajectory is also called **dense** trajectory. Indeed the main goal of our project is to analyse the coarse and discrete trajectories generated by the movement of each user under the PIR sensors.

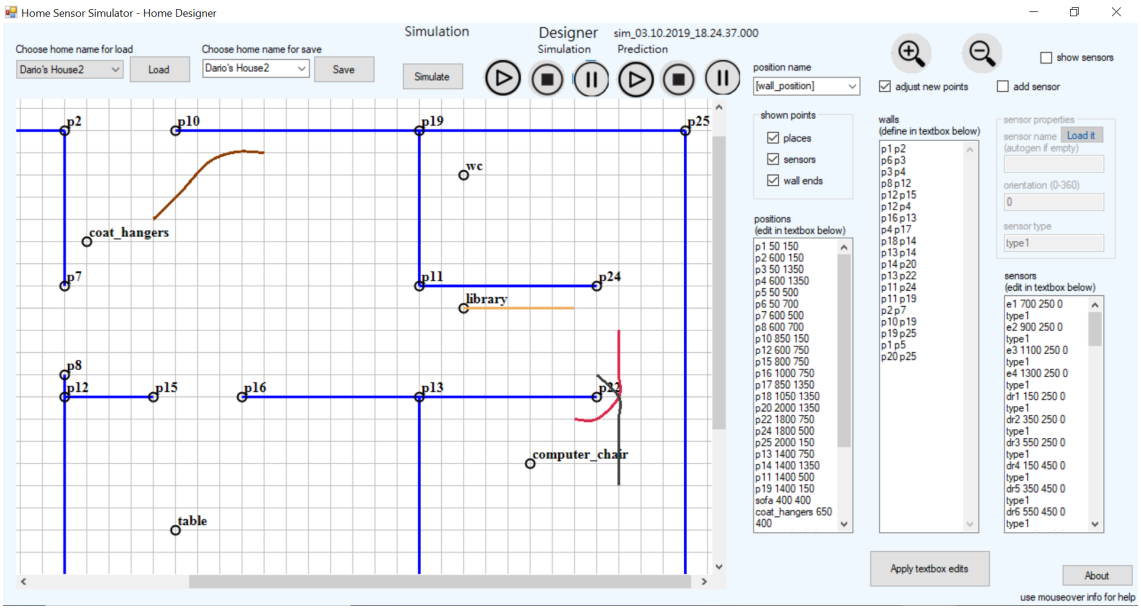


Figure 4.2. Continuous and dense trajectories

We show in Figure 4.2 an example of the continuous and dense trajectories generated from the simulator. In this chapter we analyse the **continuous** and **dense** trajectories generated by the "Home Designer" simulator. Now we briefly describe what is a dynamic continuous time system. A dynamic system is a mathematical model that represents an object (system) with a finite number of degrees of freedom that evolves over time according to a deterministic law. A dynamic system is identified by a vector in the phase space, the system state space, where "state" is a term that indicates the set of physical quantities, called state variables, that characterize the dynamics of the system. In classical mechanics an elementary example of a dynamic system is provided by a point that moves in space. The point is completely characterized by its position $r(t)$, a vector dependent from t, and from its velocity, $v(t) = r(t) = dr/dt$. The state of this system is the vector, where the state space is used and its elements represent all the possible states that the system can take. State space is also called phase space.

The temporal evolution of the point is therefore given by the two derivatives:

$r(t)=v(t),\ r(t)=v(t)=a,$

where a is the acceleration of the point (which depends on the sum of the forces to which it is subject). By defining:

$x(t)=(r(t),v(t))$

the motion of the point can be written with the ordinary autonomous equation:

$x(t)=f(x(t))$

Choosing an initial point and velocity $x_0 = (r_0,\ v_0)$ $x_0 = (r_0,\ v_0)$, or placing $x(t=0) \equiv x_0$, we get the evolution of the system starting from x_0 (cauchy problem for the differential equation).
All continuous-time dynamic systems are written in a similar way, possibly with f which explicitly depends on time:

$x(t)=f(x(t),t)\ x\in \mathbb{R}^n$

where f: $\mathbb{R}^n$ x $\mathbb{R} \rightarrow \mathbb{R}^n$ is an at least differentiable function. This system can be traced back to the autonomous one f: $\mathbb{R}^n \rightarrow \mathbb{R}^n$ with a change of variables. The solution (x_0, t) with t is the trajectory (orbit) followed by the system in the phase space starting from x_0.

More in general a dynamic system is defined in this way: a mathematical model of an object physical interacting with the world surrounding by two vectors of time-dependent variables t.

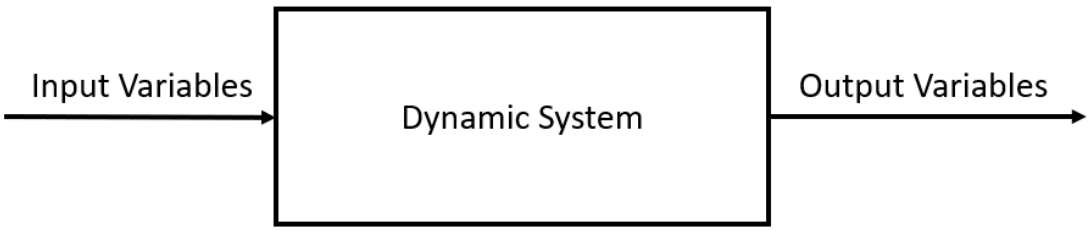


Figure 4.3. Dynamic system

The input variables are the actions performed on the system by agents external influencing their behaviour. The output variables are the quantities of the system under consideration that, for some reason, they are of interest. Cause-effect relationship exists between the variables. Normally the value of the input (cause) at a certain moment time is not sufficient to determine the value assumed by the output (effect) at the same time. The state variables are variables that describe the "internal

situation" of the system (determined by the past history) necessary for determine the output. We define the dependence of the output from the input and from the state:

$y(t) = h(x(t), u(t), t)$ where y(t) is the output vector, x(t) is the state vector and u(t) is the input vector.

The state evolution as a function of the input and state is:

$\frac{dx(t)}{dt} = f(x(t), u(t), t)$ that is the derivate of the state function at the instant t.

Given $x(t_0)$,state value at the initial time, and given $u(t)$, $t \geq t_0$, under certain regularity properties of $f(\cdot)$, then the equation of state defines the trend of $x(t)$, $y(t)$.

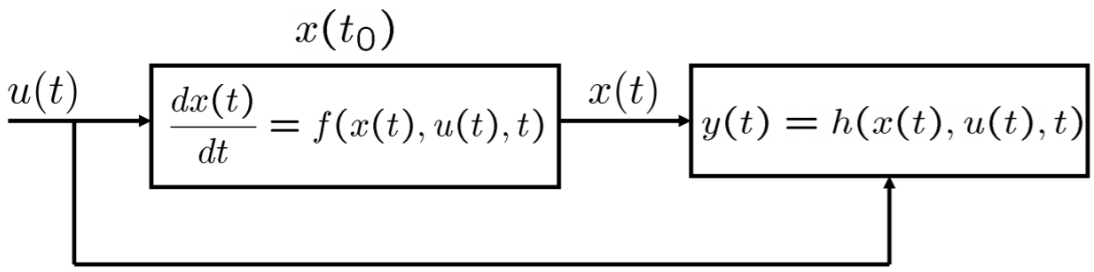


Figure 4.4. Dynamic system

A dynamic system is said "linear" if the functions f and h depend linearly from the state and input variables, i.e.:

$x(t) = A(t)x(t) + B(t)u(t)$
$y(t) = C(t)x(t) + D(t)u(t)$

where dim(A)= n x n, dim(B) = n x m, dim(C)= r x n, dim(D) = r x m.
(n = order of the model, m = number of the inputs, r = number of the outputs)

4.2 Creation of the Sensor Log

The simulator generates for each user a log of the trajectories, that doesn't correspond to the definition of sensor log. By definition a sensor log is a sequence of measurements of the kind <ts, s, v> vi where ts is the timestamp of the measurement, s ∈ S is the source sensor and v the measured value,which can be either nominal (categorical) or numeric (quantitative). Then by its nature a sensor log contains the trajectories of all users. For this reason we have implemented an algorithm for generating a sensor log for each simulation activity. This algorithm is similar both for the continuous and for the discrete trajectories regard to the design, except for the implementation details.

The algorithm takes in input the logs of each user and returns the complete sensor log. In the figure below we show what result we obtain.

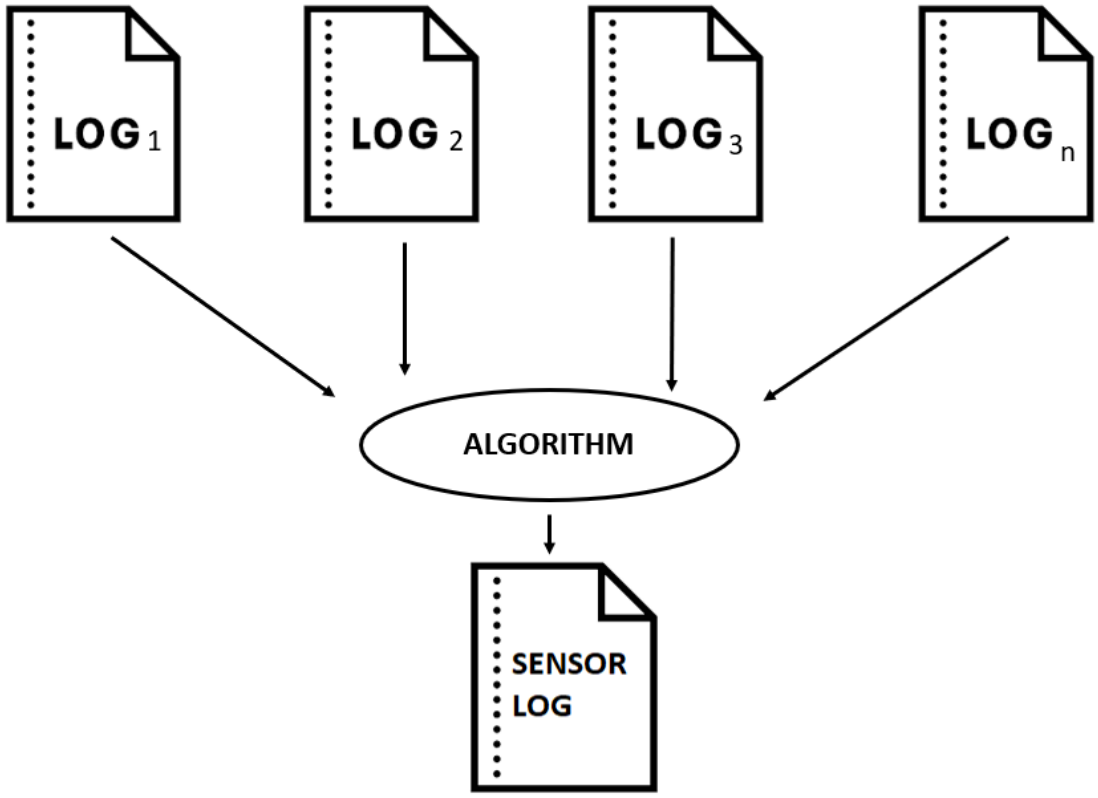


Figure 4.5. Creation of the sensor log.

4.2.1 Design of the algorithm

We describe the pseudocode of the algorithm of the creation of the sensor log below.

Algorithm Creation of the Sensor Log

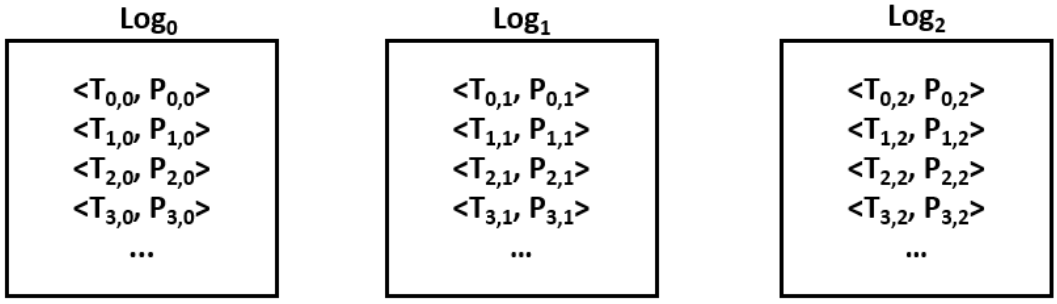
Input: The list of the logs of the user trajectories

Min = 0
List = ∅
foreach Log$_i$:
    List = **List** ∪ read first line append the entry <timestamp, position> of each first line to the list

while List ¬∅
    Min= calculate minimum timestamp(**List**) the entry with the minimum timestamp in the list
    write **Min** in the Sensor Log in the form of <timestamp, position>
    remove **Min** from **List** remove the minimum entry from the list
    NewLine = read line from the log with the minimum timestamp calculated
    List = **List** ∪ **NewLine**

The performance cost of this algorithm is O(nm), where n is the length, i.e. number of the line, of the biggest log file and m is the length of the support list.

For clarity we show an example of execution of the described algorithm. We have three logs, log0, log1 and log2 as in the Figure below: T is the timestamp and P is the position in the logs.



1. **List** = < <$T_{0,0}$, $P_{0,0}$>, <$T_{0,1}$, $P_{0,1}$>, <$T_{0,2}$, $P_{0,2}$> >;

2. **Min** = <$T_{0,0}$, $P_{0,0}$>;
3. write **Min** in the Sensor Log;
4. **List** = < <$T_{0,1}$, $P_{0,1}$>, <$T_{0,2}$, $P_{0,2}$> >;
5. **List** = < <$T_{1,0}$, $P_{1,0}$>, <$T_{0,1}$, $P_{0,1}$>, <$T_{0,2}$, $P_{0,2}$> >;
6. restart from 2. until the List is not empty and all the file pointers are null.

4.2.2 Implementation details

We have implemented the algorithm of the sensor log as a Python module. The logs of the user trajectories are text files. The sensor log in output is also a text file. The algorithm reads from files and keeps track of a list of file pointers for moving and reading one line at time from the exact log file.

The substantial difference in the implementation between the continuous and and discrete case is the write in the sensor log. In the continuous case the module writes each entry in this form < timestamp, coordinates>, e.g. <"00:00:00.000", 100 150>, and in the discrete it writes < timestamp, triggered sensor, sensor status >, e.g. < "10.09.2019 00:00:00.000", s1, ON>. We have used a python library "tkinter" for the filedialog for searching and selecting the directory of the simulation on which to create the sensor log.

4.3 Tracking and predicting the user trajectories

After we have obtained the sensor log, we have designed a tracking algorithm for reconstructing and classifying the trajectories of each user. The ideal goal of this part of the work is to classify and distinguish the original user trajectories in each log starting from the unique and complete sensor log as showed in the Figure 4.6. As we said, first of all we analyse the case of the continuous trajectories and in the next chapter the discrete case.

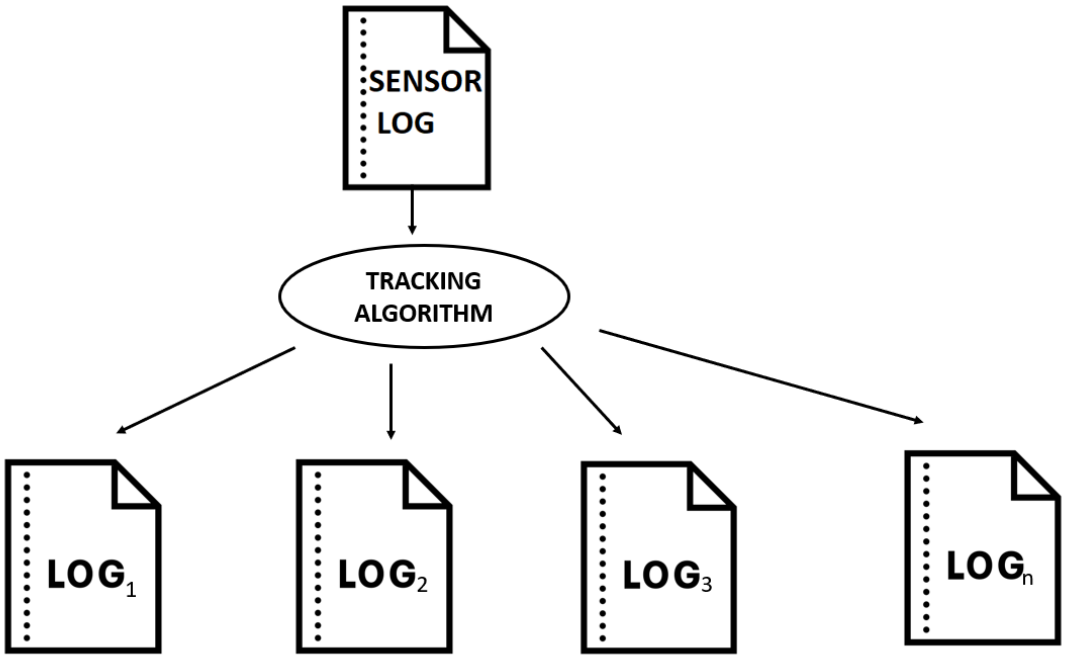


Figure 4.6. Ideal situation obtained from the tracking algorithm.

The sensor log contains the continuous trajectories of all users. The tracking algorithm outputs a series of segments, that are subsequences of the trajectories of the users as in the Figure 4.7.

The algorithm can be divided in two phase: the segmentation and the prediction phase. Initially we suppose that the users at the instant t_0 have a different position between them in the house: in our case t_0 coincides with first timestamp of the simulation namely ”00:00:00” in the sensor log. In this first step, the algorithm creates a new segment for each distinct position. Then it reads a new position from the sensor log and add it to the nearest segment.

When a segment has more than four positions, the algorithm executes the prediction phase with the Kalman filter. We must have at least three positions for each segment for applying the Kalman filter: the motivations will be explained in the section "Kalman filter". The filter estimates the next position, that is assigned to the compatible and nearest segment.

A segment is closed when a case of parallelism occurs as in the Figure 4.8: two or more parallel trajectories. In this case the algorithm cannot estimate and assign

the next position to the exact segment because two parallel trajectories contains the same positions at the same time. After the parallelism the algorithm creates and opens new segments. In the next paragraph we will discuss how the Kalman filter works and its application.

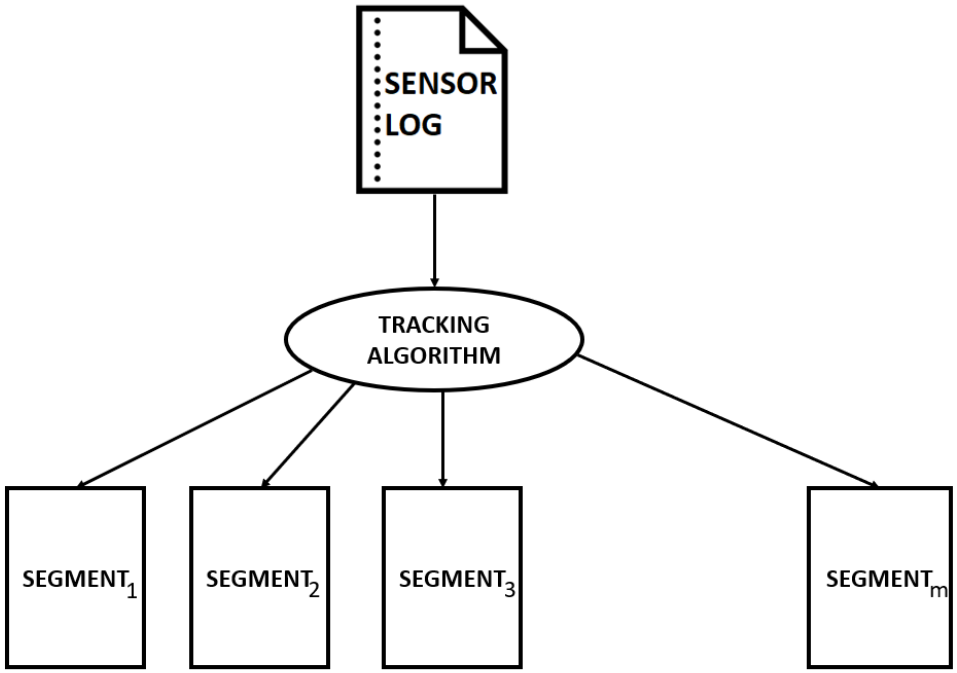


Figure 4.7. Real situation returned from the tracking algorithm.

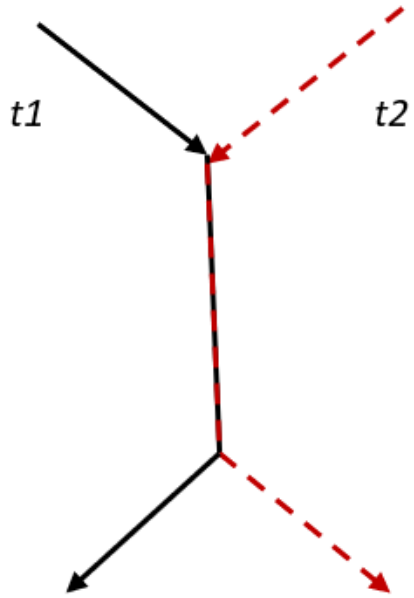


Figure 4.8. An example of two parallel trajectories t1, t2, one in black and the other with the red dotted line.

4.3.1 Kalman Filter

Kalman filter, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. The filter is named after Rudolf E. Kálmán, one of the primary developers of its theory.

The Kalman filter has numerous applications in technology. A common application is for guidance, navigation, and control of vehicles, particularly aircraft, spacecraft and dynamically positioned ships. Furthermore, the Kalman filter is a widely applied concept in time series analysis used in fields such as signal processing and econometrics. Kalman filters also are one of the main topics in the field of robotic motion planning and control, and they are sometimes included in trajectory optimization. The Kalman filter also works for modeling the central nervous system's control of movement. Due to the time delay between issuing motor commands and receiving sensory feedback, use of the Kalman filter supports a realistic model for making estimates of the current state of the motor system and issuing updated commands.

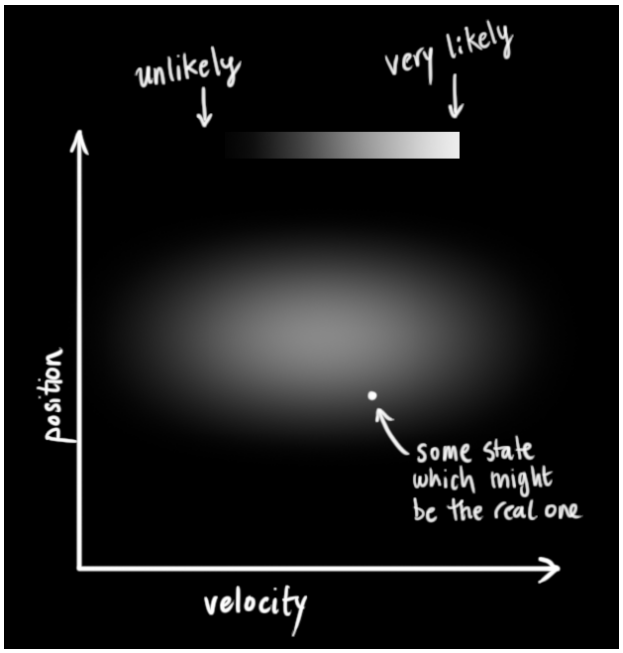
The algorithm works in a two-step process. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some amount of error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. The algorithm is recursive. It can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.

4.3.1.1 Definition of the problem

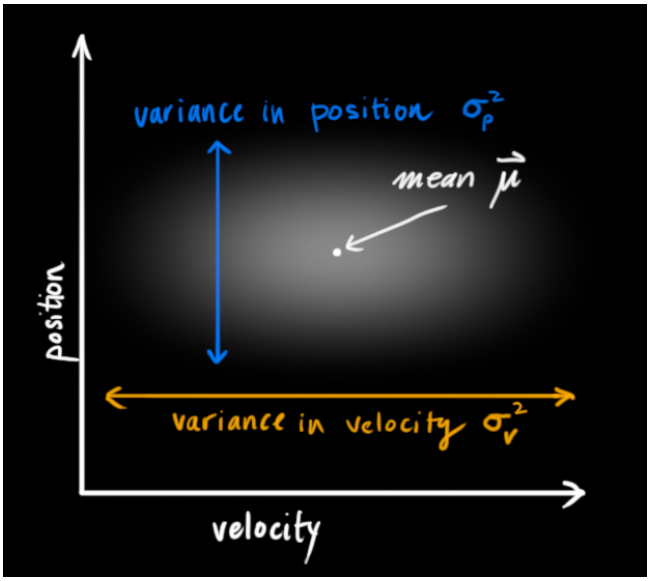
Suppose we have a simple state having only position and velocity.

$$\vec{\mathbf{x}} = \begin{bmatrix} p \\ v \end{bmatrix}$$

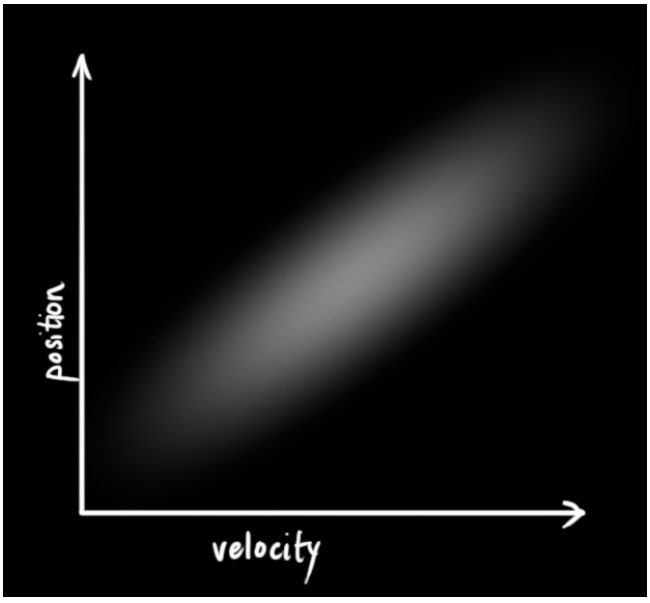
We don't know what the actual position and velocity are; there are a whole range of possible combinations of position and velocity that might be true, but some of them are more likely than others:



The Kalman filter assumes that both variables (position and velocity, in our case) are random and Gaussian distributed. Each variable has a mean value μ, which is the center of the random distribution (and its most likely state), and a variance σ_2, which is the uncertainty:

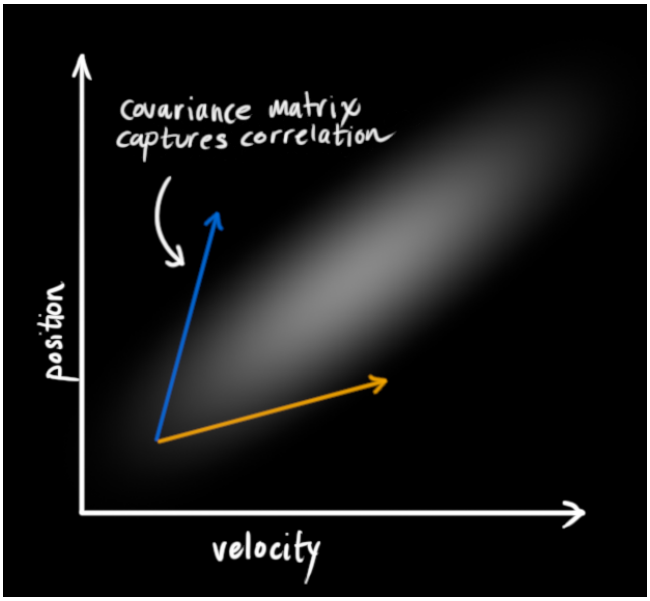


In the above picture, position and velocity are uncorrelated, which means that the state of one variable tells you nothing about what the other might be. The example below shows something more interesting: Position and velocity are correlated[13]. The likelihood of observing a particular position depends on what velocity you have:



This kind of situation might arise if, for example, we are estimating a new position based on an old one. If our velocity was high, we probably moved farther, so our position will be more distant. If we're moving slowly, we didn't get as far. This kind of relationship is really important to keep track of, because it gives us more information: One measurement tells us something about what the others could be. And that's the goal of the Kalman filter, we want to squeeze as much information from our uncertain measurements as we possibly can. This correlation is captured by something called a covariance matrix. In short, each element of the matrix $\Sigma_i j$ is

the degree of correlation between the *ith* state variable and the *jth* state variable. The covariance matrix is symmetric, which means that it doesn't matter if you swap i and j. Covariance matrices are often labelled "Σ", so we call their elements "Σ_{ij}".

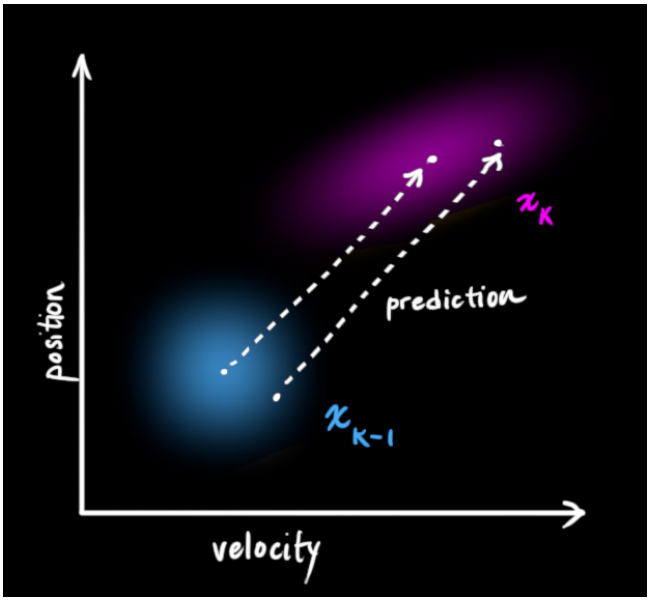


We are modelling our knowledge about the state as a Gaussian blob, so we need two pieces of information at time k: we'll call our best estimate $\hat{x_k}$ (the mean, elsewhere named μ), and its covariance matrix P_k.

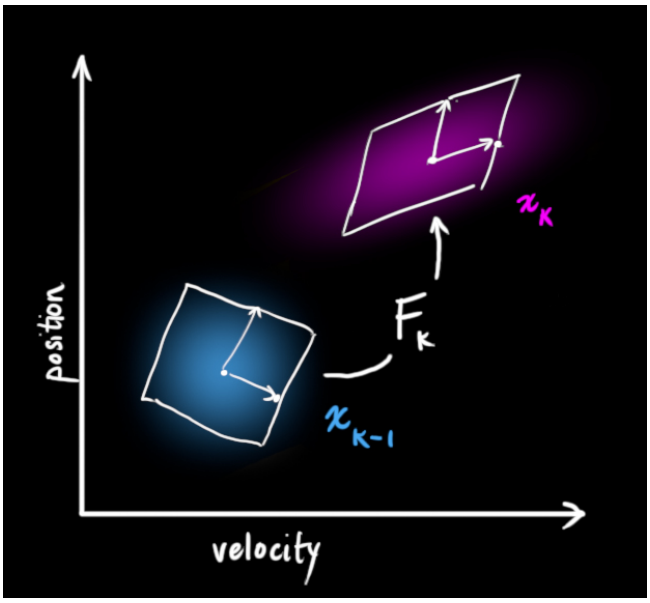
$$\hat{x_k} = \begin{bmatrix} position \\ velocity \end{bmatrix}$$

$$\hat{x_k} = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix}$$

Of course we are using only position and velocity here, but it's useful to remember that the state can contain any number of variables, and represent anything you want. Next, we need some way to look at the current state (at time **k-1**) and predict the next state at time **k**. Remember, we don't know which state is the "real" one, but our prediction function doesn't care. It just works on all of them, and gives us a new distribution:



We can represent this prediction step with a matrix, F_k:



It takes every point in our original estimate and moves it to a new predicted location, which is where the system would move if that original estimate was the right one. We use a matrix to predict the position and velocity at the next moment in the future; we apply a really basic kinematic formula[4]:

$$\begin{aligned} p_k &= p_{k-1} + \Delta t v_{k-1} \\ v_k &= \qquad v_k - 1 \end{aligned}$$

In other words:

$$\hat{x_k} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{x_{k-1}} = F_k \hat{x_{k-1}} \qquad (1)$$

We now have a prediction matrix which gives us our next state, but we still don't know how to update the covariance matrix. This is where we need another formula. If we multiply every point in a distribution by a matrix **A**, then what happens to its covariance matrix Σ? We give the identity:

$$\begin{aligned} &Cov(\mathrm{x}) = \Sigma \\ &Cov(\mathrm{Ax}) = \mathrm{A}\Sigma A^T \end{aligned} \qquad (2)$$

So combining (2) with equation (1):

$$\begin{aligned} \hat{x_k} &= F_k \hat{x_{k-1}} \\ P_k &= F_k \ P_{k-1} F_k^T \end{aligned}$$

4.3.1.2 External influence

In a real context the movement and the velocity of a person isn't constant. Each user accelerates and decelerates. If we know this additional information about what's going on in the world, we could stuff it into a vector called $\vec{\mu_k}$, do something with it, and add it to our prediction as a correction[4]. Let's say we know the expected acceleration a due to the throttle setting or control commands. From basic kinematics we get:

$$\begin{aligned} p_k &= p_{k-1} + \Delta \mathrm{t} v_{k-1} + \tfrac{1}{2} a \Delta t^2 \\ v_k &= \qquad v_k - 1 + a\Delta t \end{aligned}$$

In matrix form:

$$\hat{x_k} = F_k \hat{x_{k-1}} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} a = F_k \hat{x_{k-1}} + B_k \ \vec{\mu_k}$$

B_k is called the control matrix and $\vec{\mu_k}$ the control vector. For very simple systems with no external influence, they could be omitted.

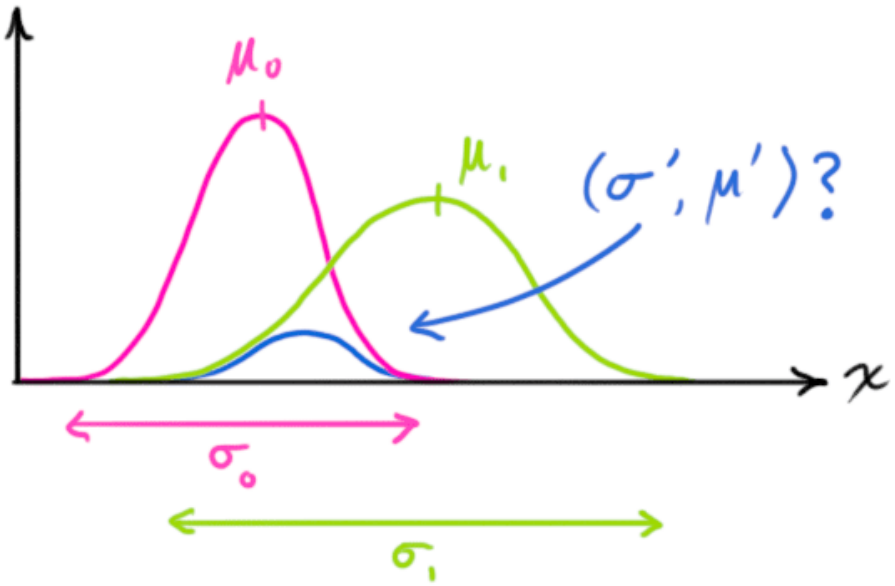
4.3.1.3 Combining Gaussians

Let's find that formula[4]. It's easiest to look at this first in one dimension. A 1D Gaussian bell curve with variance σ^2 and mean μ is defined as:

$$N(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \qquad (3)$$

We want to know what happens when you multiply two Gaussian curves together. The blue curve below represents the (unnormalized) intersection of the two Gaussian

populations:



$$N(x, \mu_0, \sigma_0) \cdot N(x, \mu_1, \sigma_1) = N(x, \mu', \sigma') \tag{4}$$

We can substitute equation (3) into equation (4) and do some algebra (being careful to renormalize, so that the total probability is 1) to obtain:

$$\mu' = \mu_0 + \frac{\sigma_0^2(\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2}$$
$$\sigma'^2 = \sigma_0^2 + \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2}$$

We can simplify by factoring out a little piece and calling it **k**:

$$\mathbf{k} = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \tag{5}$$
$$\mu' = \mu_0 + \mathbf{k}(\mu_1 - \mu_0) \tag{6}$$
$$\sigma'^2 = \sigma_0^2 - \mathbf{k}\sigma_0^2$$

We re-write equations (5) and (6) in matrix form. If Σ is the covariance matrix of a Gaussian blob, and $\vec{\mu}$ its mean along each axis, then:

$$\mathbf{K} = \Sigma_0(\Sigma_0 + \Sigma_1)^{-}1 \tag{7}$$
$$\vec{\mu}' = \vec{\mu_0} + \mathbf{K}(\vec{\mu_1} - \vec{\mu_0}) \tag{8}$$
$$\Sigma' = \Sigma_0 - \mathbf{K}\Sigma_0$$

K is a matrix called the Kalman gain, and we'll use it in just a moment.

4.3.1.4 Putting it all together

We have two distributions[4]: the predicted measurement with $(\mu_0,\Sigma_0)=(H_k\hat{x}_k,H_kP_kH_k^T)$, and the observed measurement with $(\mu_1,\Sigma_1)=(\vec{z}_k,R_k)$. We can just plug these into equation (8) to find their overlap:

$$\begin{aligned} H_k\hat{x}_k\text{'} &= H_k\hat{x}_k + \mathbf{K}(\vec{z}_k - H_k\hat{x}_k) \\ H_kP_k\text{'}H_k^T &= H_kP_kH_k^T - \mathbf{K}H_kP_kH_k^T \end{aligned} \tag{9}$$

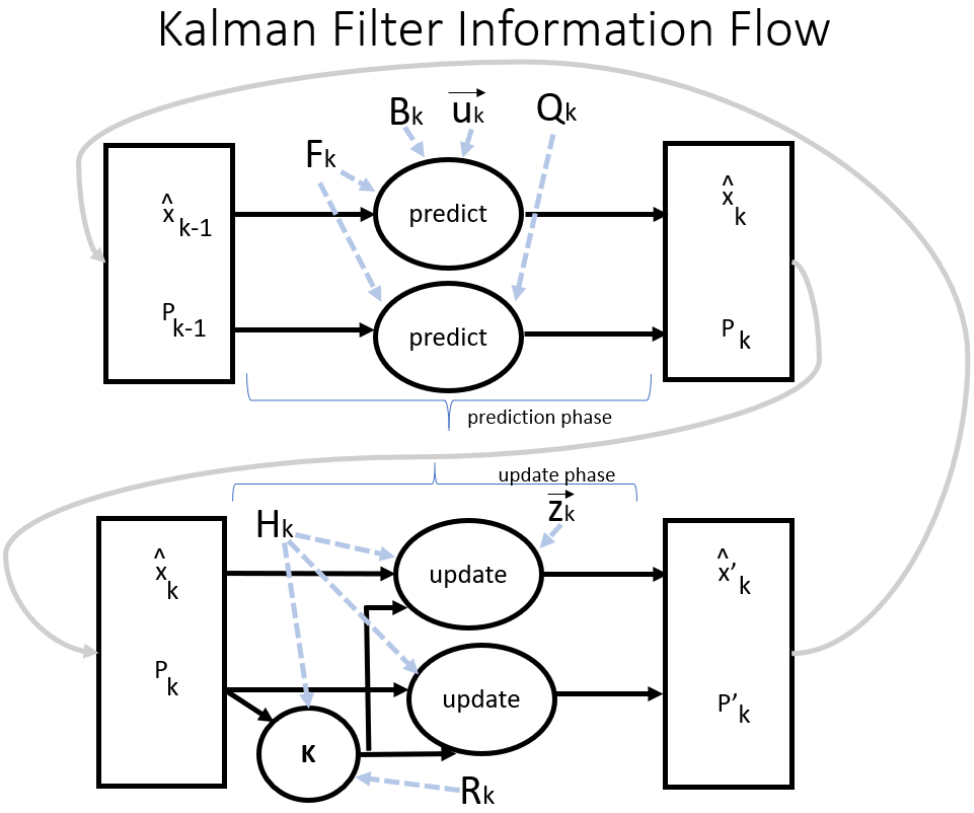
And from (7), the Kalman gain is:

$$\mathbf{K} = H_kP_kH_k^T(H_kP_kH_k^T + R_k)^{-1} \tag{10}$$

We can knock an H_k off the front of every term in (9) and (10) (note that one is hiding inside $\mathbf{K}$), and an H_k^T off the end of all terms in the equation for P_k'.

$$\begin{aligned} \hat{x}_k\text{'} &= \hat{x}_k + \mathbf{K}\text{'}(\vec{z}_kH_k\hat{x}_k) \\ P_k\text{'} &= P_k - \mathbf{K}\text{'}H_kP_k \\ \mathbf{K}\text{'} &= P_kH_k^T(H_kP_kH_k^T + R_k)^{-1} \end{aligned}$$

giving us the complete equations for the update step. $\hat{x}_k$' is our new best estimate, and we can go on and feed it (along with P_k') back into another round of **predict** or **update** as many times as we like.



4.3.2 Design of the algorithm

We describe the pseudocode of the algorithm of the Kalman filter for tracking the user paths and reconstruction the trajectories.

Algorithm Kalman Filter

Input: SensorLog

ListofSegments = $\emptyset$

while SensorLog $\neg\emptyset$:
read a line from **SensorLog**, namely $< timestamp_i, position_i >$
if $s_j \in$ **ListofSegments** is compatible with $<timestamp_i, position_i>$ and length(s_j) < 4:
$s_j = s_j \cup < timestamp_i, position_i >$
else if length(s_j) > 4:
if s_j contains parallel positions with $<timestamp_i, position_i>$:
close(**s**)
else:
EstimatedPosition = KalmanFilter(s_j, $<timestamp_i, position_i>$)
assign the $< timestamp_i, position_i >$ at the segment with the minimum distance with the **EstimatedPosition**
else:
s = create new segment
s = **s** $\cup$ $<timestamp_i, position_i>$
ListofSegments = **ListofSegments** $\cup$ **s**

4.4 Implementation of the algorithm

We have implemented the algorithms for the creation of the SensorLog and the Kalman filter using Python3.

4.4.1 Algorithm of the creation of the sensor log

The creation of the SensorLog is done by the "*trajectoriesmodule.py*" script. This algorithm takes in input the logs of the user trajectories, in the form of text file and returns the SensorLog that is also a text file. The algorithm as we have described in

the second section makes a shuffling of the data of each log of user trajectories. In the code below we show the opening and then the reading of the text files of the logs of the trajectories of the users. At the end the function "costructdatabase()" creates a relational database for the storage of the simulated trajectories. We create this database because after we have reconstructed the trajectories with the algorithm with the Kalman filter, the goal is testing this algorithm. The performance test requires the comparing between the simulated trajectories and the reconstructed trajectories: these operations are efficiently done reading the data from a db.

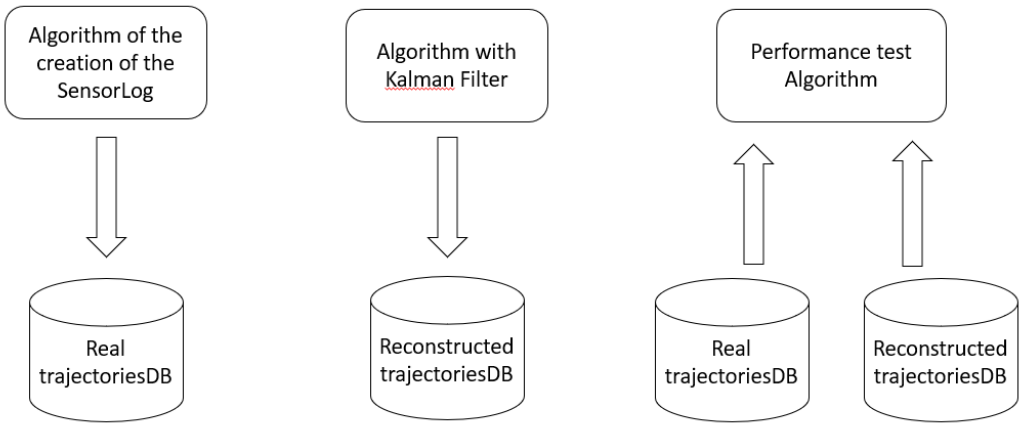


Figure 4.9. This schema shows the architecture described.

```
1           root = Tk()
2           root.directory = filedialog.askdirectory(initialdir="C:\\Users\\Dario\\
3           Desktop\\HomeDesigner\\bin\\Debug\\Log", title="Select file")
4           exists = os.path.isfile((root.directory+"\\DatasetPaths.txt"))
5           if exists:
6
7                   os.remove(root.directory+"\\DatasetPaths.txt")
8
9           exists2 = os.path.isfile(
10          (root.directory+"\\trajectoriesDB.db"))
11          if exists2:
12                  os.remove(root.directory+"\\trajectoriesDB.db")
13
14          existsRealDB = os.path.isfile(
15          (root.directory + "\\realTrajectoriesDB.db"))
16          if existsRealDB:
17                  os.remove(root.directory + "\\realTrajectoriesDB.db")
18
19          path= root.directory+"\\*.txt"
20
21
22          #lettura dei pathlog della simulazione
```

```
        files = glob.glob(path)
        listafile=list()
        flag=False
        listaSegmentiDB=list()
        for name in files:
                try:
                with open(name) as f:
                        f = open(name, "r")
                        listaSeg=list()
                        leggifile(f, listaSeg)
                        listaSegmentiDB.append(listaSeg)
                except IOError as exc:  # Not sure what error this is
                        if exc.errno != errno.EISDIR:
                                raise
        constructdatabase(listaSegmentiDB,root.directory)

```

The second phase that we present below is the shuffling phase and the relative creation of the SensorLog.

```
         while(True):
                if(j==0):
                        listapuntatorilinee=leggi_linea(listafile)
                j+=1
                if( not listapuntatorilinee):
                        datasetfile.close()

                        traj_reconstructor.recontruct_path_logs(root.directory)

                        return

                tsminimo=calcola_minimo_timestamp(listapuntatorilinee)
                parsedstring=tsminimo.split('-')
                timestampmin=parsedstring[0]
                filetsminimo=int(parsedstring[1])
                idfiletsminimo=int(parsedstring[2])
                posizione=(parsedstring[3])
                datasetfile.write(timestampmin+" "+posizione)

                ##rimuovo tsmin da listapuntatorilinee
                ##file tsminimo leggo una nuova linea e lo aggiungo
                ##in listapuntatorilinee
                parsedline=listafile[idfiletsminimo].readline().split(' ')
                line=parsedline[0]

                if(line=="" and len(listapuntatorilinee)!=0):
                        datasetfile.write( "\n")
```

```
28                  if (line == "" and len(listapuntatorilinee) == 0):
29                          return
30
31                  if (line != ""):
32
33                          listapuntatorilinee.append(line+"-"+str(idfiletsminimo)+"-"+
34                          parsedline[1]+" "+parsedline[2])
35
36                  del(listapuntatorilinee[filetsminimo])
```

4.4.2 Algorithm with the Kalman filter

The algorithm with the Kalman filter is implemented in the the "*trajreconstructor.py*" script. At the end the "*performancetest.py*" script is launched for testing the performance of the algorithm with the Kalman filter. Below we show the step when the kalman filter is launched for estimating the next position and the creation of the cluster that contains the previous positions[1].

```
1           if( s!="fine_segmento" and len(listasegmenti[i]) >= 4):
2                   parsed = listasegmenti[i][len(listasegmenti[i]) - 2].split(' ')
3                   lastPosition0 = Position()
4                   lastPosition0.x = float(parsed[1].replace(',', '.'))
5                   lastPosition0.y = float(parsed[2].replace(',', '.'))
6                   t0 = datetime.strptime(parsed[0], ("%H:%M:%S.%f"))
7
8                   parsed = listasegmenti[i][len(listasegmenti[i]) - 3].split(' ')
9
10                  lastPosition1 = Position()
11                  lastPosition1.x = float(parsed[1].replace(',', '.'))
12                  lastPosition1.y = float(parsed[2].replace(',', '.'))
13                  t1 = datetime.strptime(parsed[0], ("%H:%M:%S.%f"))
14                  cluster = Cluster()
15                  cluster.lista_posizioni = list()
16                  cluster.lista_posizioni.append(lastPosition1)
17                  cluster.lista_posizioni.append(lastPosition0)
18                  cluster.lista_posizioni.append(pos)
19                  cluster.id_segment = i
20                  lista_clusters.append(cluster)
21                  puntoIncrocio = pos
22                  timestampIncrocio = timestampPos
23                  tm = datetime.strptime(timestampPos, ("%H:%M:%S.%f"))
24                  print(str(pos.x) + " " + str(pos.y) + " " + timestampIncrocio)
25                  index = i
26                  d=kalman_filter.kalman_filter_position(cluster, p)
27                  ep= EstimatedPosition()
28                  ep.position=p
29                  ep.distanza=d
```

```
                ep.id_segment=i
                list_distances.append(ep)
                if (i == len(listasegmenti) - 1):
                        id_segment = GetPuntoVicino(list_distances)
                        listasegmenti[id_segment].append(timestamp + " " + str(p.x)
                        + " " + str(p.y))
                        list_distances.clear()
                continue
```

4.4.3 Kalman filter implementation details

We analyse the Kalman filter and its implementation in two case: when the velocity is constant and so the motion is uniform and when the motion is various. The second case obviously represents the real movements of one or more persons inside an house.

4.4.3.1 Uniform motion with constant velocity

For a system which is linear, and which has known dynamics i.e. if you know the state and inputs, you can predict the future state, it provides an optimal way of combining what you know about a system to estimate it's true state. The clever bit (which is taken care of by all the matrix algebra you see on pages describing it) is how it optimally combines the two pieces of information you have:

1. Measurements (which are subject to "measurement noise", i.e. sensors not being perfect)

2. Dynamics (i.e. how we believe states evolve subject to inputs, which are subject to "process noise", which is just a way of saying our model doesn't match reality perfectly).

We specify how sure we are on each of these (via the co-variances matrices R and Q respectively), and the Kalman Gain determines how much we should believe our model (i.e. our current estimate of our state), and how much we should believe our measurements. Without further ado let's build a simple model of your kite. What we propose below is a very simple possible model. Let's treat the user inside the house as a particle (obviously a simplification, a real person is an extended body, so has an orientation in 3 dimensions), which has four states which for convenience we can write in a state vector:

x = [x, x_dot, y, y_dot]

Where x and y are the positions, and the _dot's are the velocities in each of those directions. We are assuming there are two (potentially noisy) measurements, which we can write in a measurement vector:

z = [x, y]

We can write-down the measurement matrix

z = Hx ⇒ H = [[1, 0, 0, 0], [0, 0, 1, 0]]

We then need to describe the system dynamics. Here we will assume that no external forces act, and that there is no damping on the movement of the user (with more knowledge you may be able to do better, this effectively treats external forces and damping as an unknown/unmodeled disturbance). In this case the dynamics for each of our states in the current sample "k" as a function of states in the previous samples "k-1" are given as:

x(k) = x(k-1) + dt*x_dot(k-1)
x_dot(k) = x_dot(k-1)
y(k) = y(k-1) + dt*y_dot(k-1)
y_dot(k) = y_dot(k-1)

Where "dt" is the time-step. We assume (x, y) position is updated based on current position and velocity, and velocity remains unchanged. Given that no units are given we can just say the velocity units are such that we can omit "dt" from the equations above, i.e. in units of position_units/sample_interval (I assume your measured samples are at a constant interval). We can summarise these four equations into a dynamics matrix as (F discussed here, and transition_matrices in pykalman library[1]):

x(k) = Fx(k-1) ⇒ F = [[1, 1, 0, 0], [0, 1, 0, 0], [0, 0, 1, 1], [0, 0, 0, 1]]

The dynamics captured in the model above are very simple. Taken literally we say that the positions will be updated by current velocities (in an obvious, physically reasonable way), and that velocities remain constant (this is clearly not physically true, but captures our intuition that velocities should change slowly)[2].

```
1  def kalman_filter_position(cluster,position):
2          c=Cluster()
3          c.lista_posizioni=cluster.lista_posizioni
4          c.id_segment=cluster.id_segment
5          c.lista_posizioni.append(position)
6          a= list()
7
8          for ps in c.lista_posizioni:
9                  tup1 =(ps.x,ps.y)
10                 #print(tup1)
11
12                 a.append(tup1)
```

```
13
14
15          measurements = np.asarray(a)
16
17          initial_state_mean = [measurements[0, 0],
18                  0,
19                  measurements[0, 1],
20                  0]
21
22          transition_matrix = [[1, 1, 0, 0],
23                                                  [0, 1, 0, 0],
24                                                  [0, 0, 1, 1],
25                                                  [0, 0, 0, 1]]
26
27          observation_matrix = [[1, 0, 0, 0],
28                                                  [0, 0, 1, 0]]
29
30          time_before = time.time()
31          n_real_time = 3
32
33          kf3 = KalmanFilter(transition_matrices =
34                                              transition_matrix,
35                                              observation_matrices =
36                                              observation_matrix,
37                                              initial_state_mean =
38                                              initial_state_mean,)
39          means, covariances = kf3.filter(measurements)
40          kf3 = kf3.em(measurements, n_iter=5)
41          (filtered_state_means, filtered_state_covariances) =
42          kf3.filter(measurements)
43          #print("Time to build and train kf3: %s seconds" % (time.time() -
44          #time_before))
45          n_timesteps = 4
46          n_dim_state =  4
47          filtered_state_means2 = np.zeros((n_timesteps, n_dim_state))
48          filtered_state_covariances2 = np.zeros((n_timesteps, n_dim_state,
49          n_dim_state))
50          i=0
51          for t in range(0,4):
52                  if t == 0:
53                          filtered_state_means2[t] = kf3.initial_state_mean
54                          filtered_state_covariances2[t] =
55                          kf3.initial_state_covariance
56                  if(t+1<4):
57
58                          filtered_state_means2[t + 1],
59                          filtered_state_covariances2[t + 1] = (
```

```
60                          kf3.filter_update(means[-1],
61                          covariances[-1], measurements[t+1]))
62
63
64                  i = i + 1
65
66          pos_calcolata=Position()
67          pos_calcolata.x=filtered_state_means2[3][0]
68          pos_calcolata.y=filtered_state_means2[3][2]
69          #print(str(position.x)+" "+str(position.y)+"posvera")
70          #print(str(pos_calcolata.x)+" "+str(pos_calcolata.y)+"poscalcolata")
71          distanza= GetDistanza(pos_calcolata,position)
72
73          return distanza
```

4.4.3.2 Various motion

In various motion, acceleration does not remain constant over time. If the material point moves on a rectilinear trajectory and the module of its speed does not remain constant over time, we are talking about various motion. In this case the parameters of the Kalman filter have to be changed respect to the uniform motion. We set up the observation covariance matrix R, that represents a measurement uncertainty. This measurement uncertainty indicates how much one trusts the measured values. Since we measure the position and the velocity, this is a 2 x 2 matrix. This matrix is defined as measurement noise covariance matrix R.

```
def kalman_filter_position(cluster,position):
        c=Cluster()
        c.lista_posizioni=cluster.lista_posizioni
        c.id_segment=cluster.id_segment
        c.lista_posizioni.append(position)
        a= list()
        #print(len(c.lista_posizioni))
        for ps in c.lista_posizioni:
                tup1 =(ps.x,ps.y)
                #print(tup1)

                a.append(tup1)


        measurements = np.asarray(a)


        initial_state_mean = [measurements[0, 0],
        0,
        measurements[0, 1],
        0]

        transition_matrix = [[1, 1, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 1],
        [0, 0, 0, 1]]

        observation_matrix = [[1, 0, 0, 0],
        [0, 0, 1, 0]]

        time_before = time.time()
        n_real_time = 3
        observation_covariance =[

        [ 100, 0],
        [ 0, 100]]
        kf3 = KalmanFilter(transition_matrices = transition_matrix,
```

```
observation_matrices = observation_matrix,

initial_state_mean = initial_state_mean,
observation_covariance= observation_covariance,
em_vars=['transition_covariance', 'initial_state_covariance']
)
means, covariances = kf3.filter(measurements)


kf3 = kf3.em(measurements, n_iter=8)
(filtered_state_means, filtered_state_covariances) =
kf3.filter(measurements)
#print("Time to build and train kf3: %s seconds" % (time.time() -
#time_before))
n_timesteps = 4
n_dim_state =  4
filtered_state_means2 = np.zeros((n_timesteps, n_dim_state))
filtered_state_covariances2 =
np.zeros((n_timesteps, n_dim_state,
n_dim_state))
i=0
for t in range(0,4):
        #print("ciao")

        if t == 0:
                filtered_state_means2[t] = kf3.initial_state_mean
                filtered_state_covariances2[t] =
                kf3.initial_state_covariance

        if(t+1<4):

                filtered_state_means2[t + 1],
                filtered_state_covariances2[t + 1] = (
                kf3.filter_update(
                means[-1],
                covariances[-1],
                measurements[t+1]

                )
                )
        #print(filtered_state_means2[i])

        i = i + 1


pos_calcolata=Position()
pos_calcolata.x=filtered_state_means2[3][0]
```

```
        pos_calcolata.y=filtered_state_means2[3][2]
        #print(str(position.x)+" "+str(position.y)+"posvera")

        #print(str(pos_calcolata.x)+" "+str(pos_calcolata.y)+"poscalco")

        distanza= GetDistanza(pos_calcolata,position)

        return distanza
```

4.5 Tests and performance

We have tested the performances of the algorithm with the Kalman filter changing several parameters: the module of the velocity, the type of the motion, uniform or various, the number of the persons moving in the house. The performances are tested using "performance_test.py" script. The accuracy of this algorithm is calculated in this way:

$$accuracy(\%) = \frac{number\_of\_right\_decisions \ * \ 100}{number\_of\_total\_decisions}$$

If we have a simulated trajectory t and a crossing point p, there is a right decision since exists a segment s_i in the reconstructed trajectories with the equal positions to t before and after the crossing point p; else we have a wrong decision.

4.5.1 Uniform motion

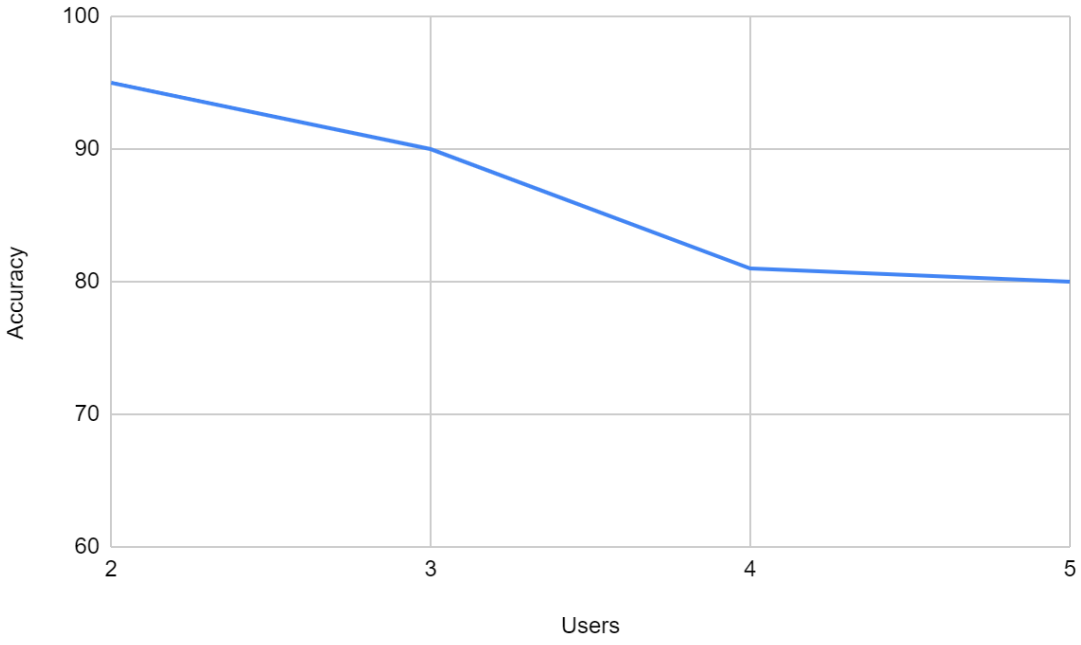


Figure 4.10. The graph shows the accuracy-number of users relation, with a constant velocity in module of 0.25 m/s.

In the Figure 4.10 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 0.25 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after four users the accuracy decreases considerably under 90%. This effect is due to a greater presence of crossing trajectories, and the difficulty of the algorithm to estimate the right positions.

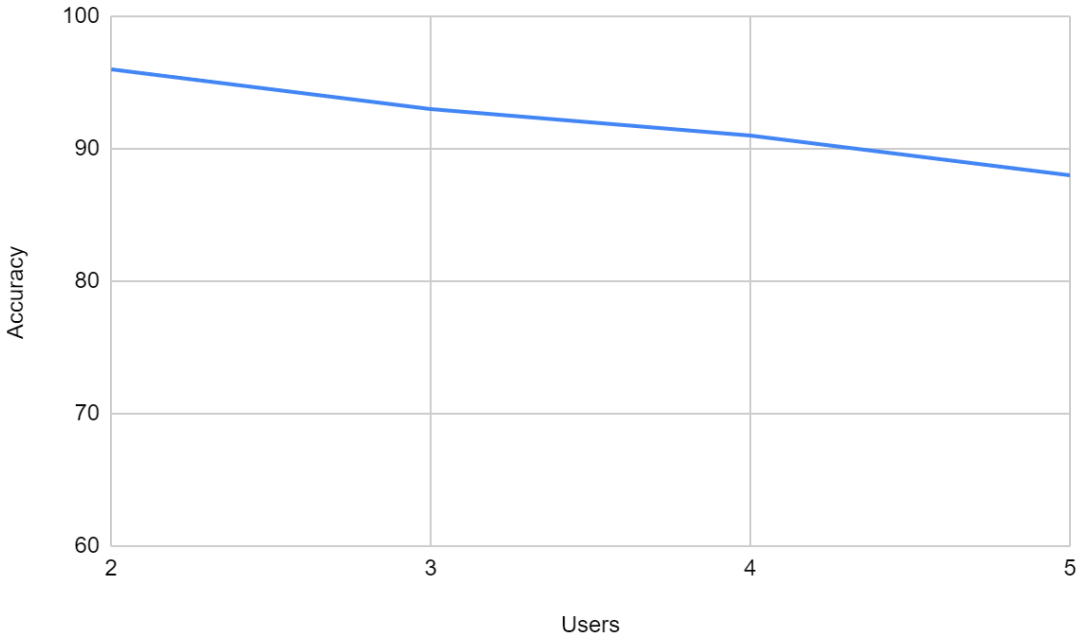


Figure 4.11. The graph shows the accuracy-number of users relation, with a constant velocity in module of 0.5 m/s.

In the Figure 4.11 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 0.5 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after four users the accuracy decreases considerably under 90%. This effect is due to a greater presence of crossing trajectories, and the difficulty of the algorithm to estimate the right positions.

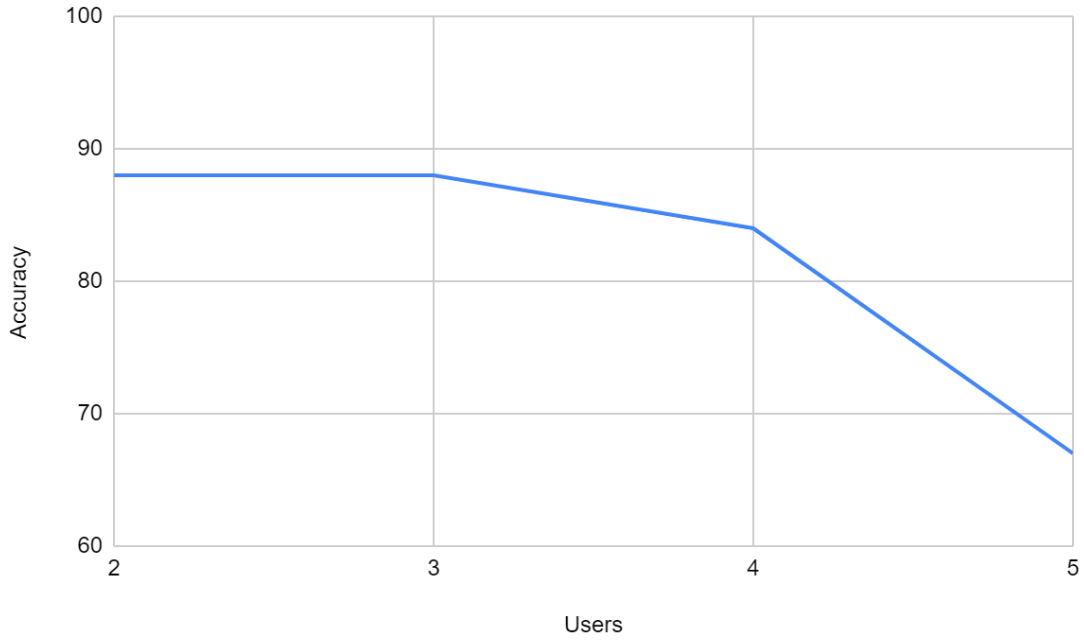


Figure 4.12. The graph shows the accuracy-number of users relation, with a constant velocity in module of 1 m/s.

In the Figure 4.12 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 1 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after five users the accuracy decreases considerably under 80%. This effect is due to a greater presence of crossing trajectories, and the difficulty of the algorithm to estimate the right positions. The accuracy in general is lower respect to the previous graphs because in this case the distances between the positions of the trajectories is greater, that is a major discretization. This effect causes more errors when the Kalman filter must predict the next position.

4.5.2 Various motion

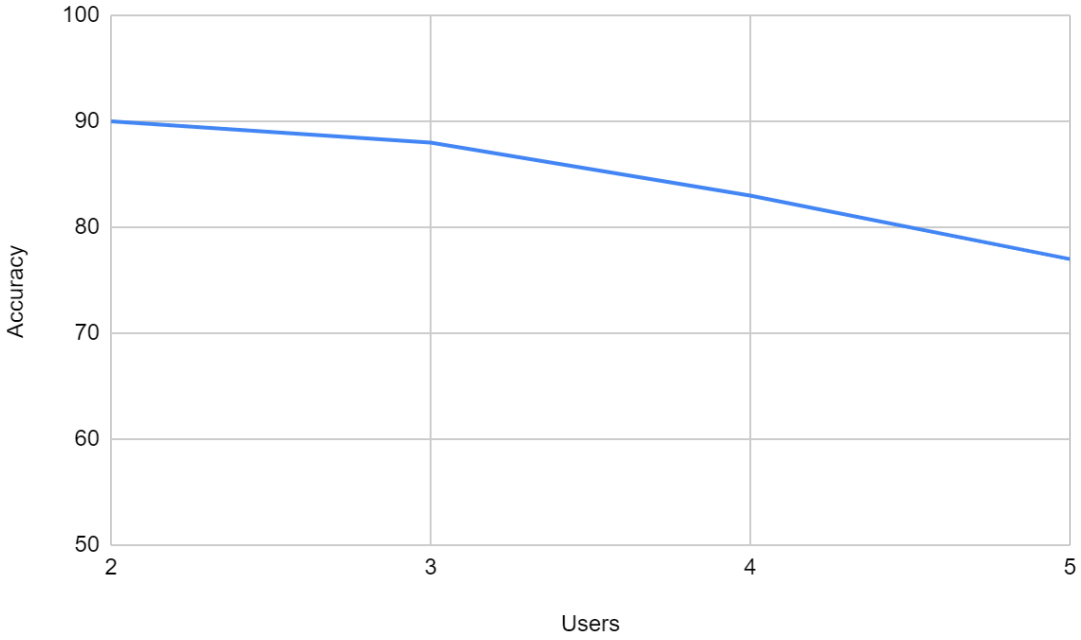


Figure 4.13. The graph shows the accuracy-number of users relation, with a variable velocity.

In the Figure 4.13 we show the relation between the number of the users and the accuracy. The users move with a various motion. We can notice that augmenting the number of the users expecially after four users the accuracy decreases under 80%. This effect is due to a greater presence of crossing trajectories, and the difficulty of the algorithm with the Kalman filter to estimate the right positions.

We can conclude with two considerations. The first consideration is that the algorithm with the Kalman filter estimates and predicts the positions of the crossing trajectories with a good accuracy in the case of uniform motion respect to the various motion. The second result is that the accuracy is very low in the case of the uniform motion with a velocity of 1 m/s. In this case the positions have a distance of one meter among them, and therefore this a major level of discretization. This result implies that we can use with a good precision the algorithm with the Kalman filter in the case of the dense and continuous trajectories and not with the rarefied and discrete trajectories, due to the high discretization of the data. The problem with the rarefied and discrete trajectories will be analysed in the next chapter.

Chapter 5

SVM for the discrete trajectories

5.1 Introduction

The goal of this part of the project is to use supervised learning techniques to identify the sub- sequences of a given sensor log, possibly coming from a smart environment populated by many users, that are related to the actions of individual users. These information can be fed as input to many other kinds of analysis. For instance, we could see these sub-sequences as traces and mine the habits of the various users in the environment by applying Process Mining (a discipline that sits between data mining and process modeling and analysis and, hence, can be considered one of the links between data science and process science) as depicted in. In these settings, the real challenge is to deal with unlabelled sensor logs to create an effective training set (made of positive samples only, in this case) for some supervised learning model. Indeed, due to the presence of many users in the environment, the traces corresponding to their actions could be interleaved and, since we rely on PIR measures only, we have no indications about which user an entry in the log is related to. Besides, even though we were able to tell which user a measure refers to, we would still have to deal with the lack of *case-ids* , that are necessary to apply most of process discovery techniques. For *case-ids* we mean the identifiers of individual real executions of a business process. Many approaches have been proposed to deal with unlabelled event logs but they are suitable only for well-structured and established (and often simple) business process[15]. Due to the highly-variable nature of human behaviour (i.e. the presence of a huge amount of noise in a sensor log), our task is way more challenging. The core of our work consists in a (batch) algorithm for sensor logs segmentation, such that the smaller chunks (i.e. sub-sequences, traces) that are produced can be considered (up to a certain degree of confidence) as related to the actions of individual users. The resulting segmentation of the given log is intended to be used as the training set of a supervised learning model, such as a One-class SVM (due to the lack of negative samples), in order to provide an on-line method for the detection of single-user segments. The validation of such a model has been conducted by taking into account only some combinations of those segments that were detected as corresponding to crossings between users trajectories. The

software produced during our work has been developed in Python 3.

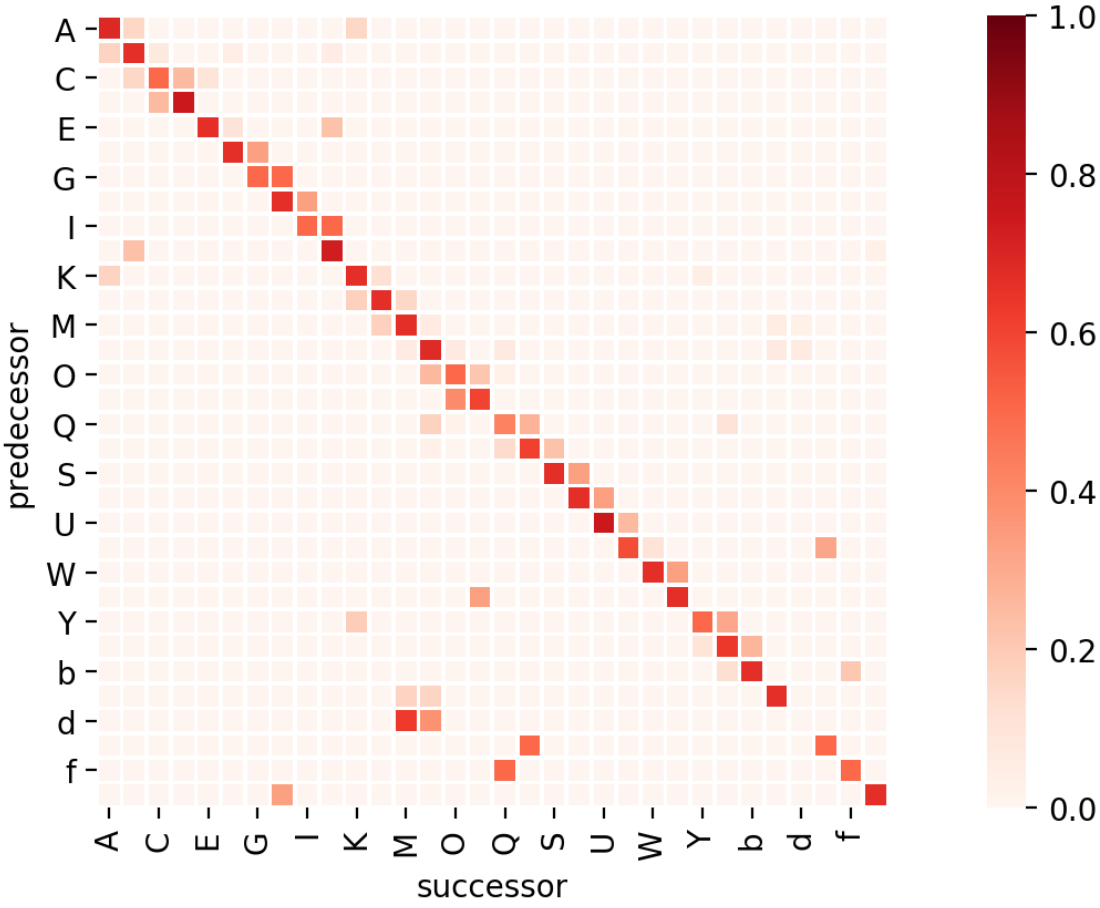


Figure 5.1. Topological compatibility matrix for the complete sensor log.

5.2 Sensor Log Segmentation

In this section, we introduce the key concepts that are the foundations of our segmentation algorithm. The underlying assumption of our approach is that there exists a locality correlation among the sensors triggered by the actions of an individual user. In other words, we expect the single-user sub-sequences of the sensor log to be constituted by measurements coming from sensors that are placed close to each other.

Algorithm Sensor Log Segmentation

$\boldsymbol{S} = \emptyset$

for all $m \in L$ **do**
    if there exists a single *open* $s \in \boldsymbol{S}$ *compatible* with m **then**
        $s = s \parallel m$
    if there exists *open* s_1,..., $s_n \in \boldsymbol{S}$ *compatible* with m **then**
        mark s_1,..., $s_n \in \boldsymbol{S}$ as *closed*
        create a new *open* segment *s'* initialized with m
        $\boldsymbol{S} = \boldsymbol{S} \cup s'$
    if there exists no *open* $s \in \boldsymbol{S}$ *compatible* with m **then**
        create a new *open* segment *s'* initialized with m
        $\boldsymbol{S} = \boldsymbol{S} \cup s'$
mark all *open* $s \in \boldsymbol{S}$ as *closed*
remove all $s \in \boldsymbol{S}$ such that $|s| < N$

5.2.1 Segmentation Algorithm

To support the locality correlation assumption, we devised a method to decide whether two measurements in a sensor log are *compatible*. Indeed, we compute the extent at which the sensors they come from are likely to be in direct succession in the given sensor log. Then, given a fixed threshold, we use this result to decide whether measurements may belong to the same single-user sub-sequence. To do that, we need to compute the compatibility degree for each possible pair of sensors occurring in the sensor log. Let M be a *topological compatibility matrix*, such that M_{ij} is the probability that a measurement m_i (from sensor i) is directly followed by a measurement m_j (from sensor j) in a single-user segment, computed as the number of occurrences of the pattern $\mathrm{m}_i \rightarrow \mathrm{m}_j$ divided by the number of measurements coming from i in the log. Given a threshold, we say that m_j is compatible with m_i if $M_{ij} >$ t (not compatible, otherwise). In **??**, the topological compatibility matrix for the sensor log considered during our work is shown. As one can clearly see, the highest compatibility values are distributed along the diagonal of the matrix. This result indeed supports the initial assumption (i.e. locality correlation) since,

in general, a sensor is followed either by itself or by another one that is placed nearby (notice that sensors placed in the same area of the environment have similar identifiers and result close to each other when alphabetically ordered). Besides, we noticed that the highest diagonal values are related with objects that, in general, are involved il long interactions with users (e.g. the bed). Provided a way to quantify the compatibility between two sensor measurements, we can trivially extend this notion to segments by saying that a measurement is compatible with a segment if it is so with the last measurement in the segment. Algorithm 1 describes the procedure we use to compute the single-user segments and that takes as inputs: a sensor log L, the topological compatibility matrix M related to L, a compatibility threshold T and a noise threshold N. Notice that M and T are needed to decide about the compatibility. We interpret the execution of a B-step as the intersection of different users trajectories, as we can see in the Figure 6.1. Each time a B-step occurs we save the pointers to the segments that are marked as closed in a separated collection B_1. Basically, when multiple users cross the same point, we need to solve the problem of tracking the continuations of their individual trajectories. Indeed, the segments in B_1 are treated as single-user and we have to consider all their possible continuations after the crossing. Notice that in this way we implicitly prune the set of possible solutions by discarding the (incoherent) possibility that segments in B_1 follow each other, since it would mean to go back in time. Then, for each new segment open between the current B-step and the next that is compatible with at least one segment in B_2, we add it to another collection B_2. This collection indeed contains all the possible continuations after the crossing. Hence, the Cartesian product B_1 X B_2 contains all the possible trajectories that could be occurred, that contribute to the validation set for our One-class SVM. Conversely, the training set consists in the (overlapping) n-gram representation of the resulting segments. Notice that we assumed the noise threshold N to be greater or equal to the length of a single n-gram.

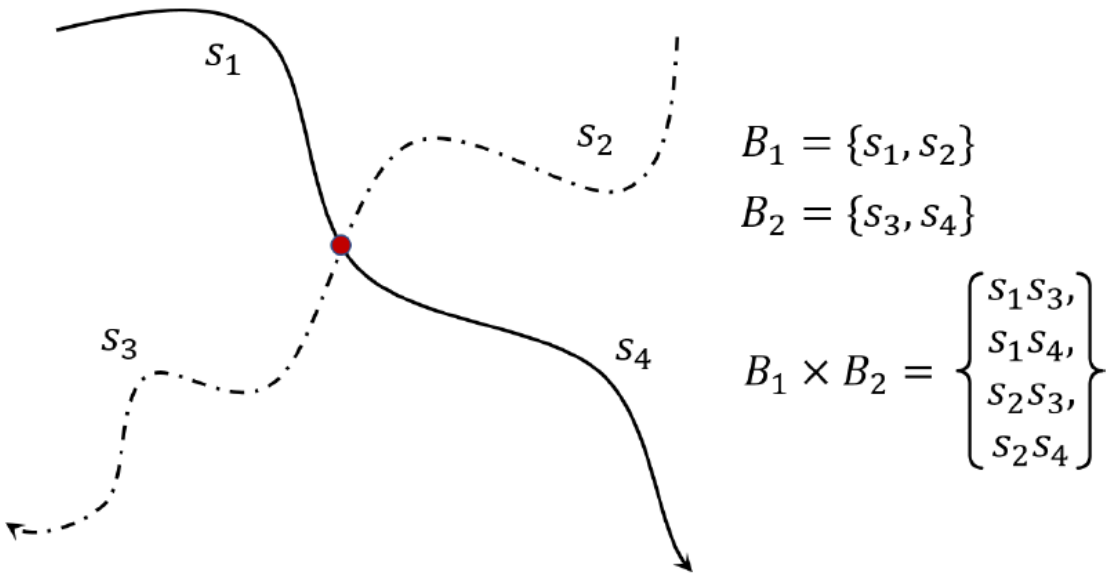


Figure 5.2. Two (intersecting) single-user trajectories and the corresponding B-step.

5.2.2 One-Class SVM

First look at our problem situation; we would like to determine whether (new) test data is member of a specific class, determined by our training data, or is not. Why would we want this? Imagine a factory type of setting; heavy machinery under constant surveillance of some advanced system. The task of the controlling system is to determine when something goes wrong; the products are below quality, the machine produces strange vibrations or something like a temperature that rises. It is relatively easy to gather training data of situations that are OK; it is just the normal production situation. But on the other side, collection example data of a faulty system state can be rather expensive, or just impossible. If a faulty system state could be simulated, there is no way to guarantee that all the faulty states are simulated and thus recognized in a traditional two-class problem. To cope with this problem, one-class classification problems (and solutions) are introduced. By just providing the normal training data, an algorithm creates a (representational) model of this data. If newly encountered data is too different, according to some measurement, from this model, it is labelled as out-of-class. We will look in the application of Support Vector Machines to this one-class problem. Our problem is very similar. We have datasets of simulated and real trajectories and the goal is to group the simulated discrete trajectories from the wrong and unreal trajectories, that are outliers[3]. One-class SVM is an unsupervised algorithm that learns a decision function for novelty detection: classifying new data as similar or different to the training set[14].

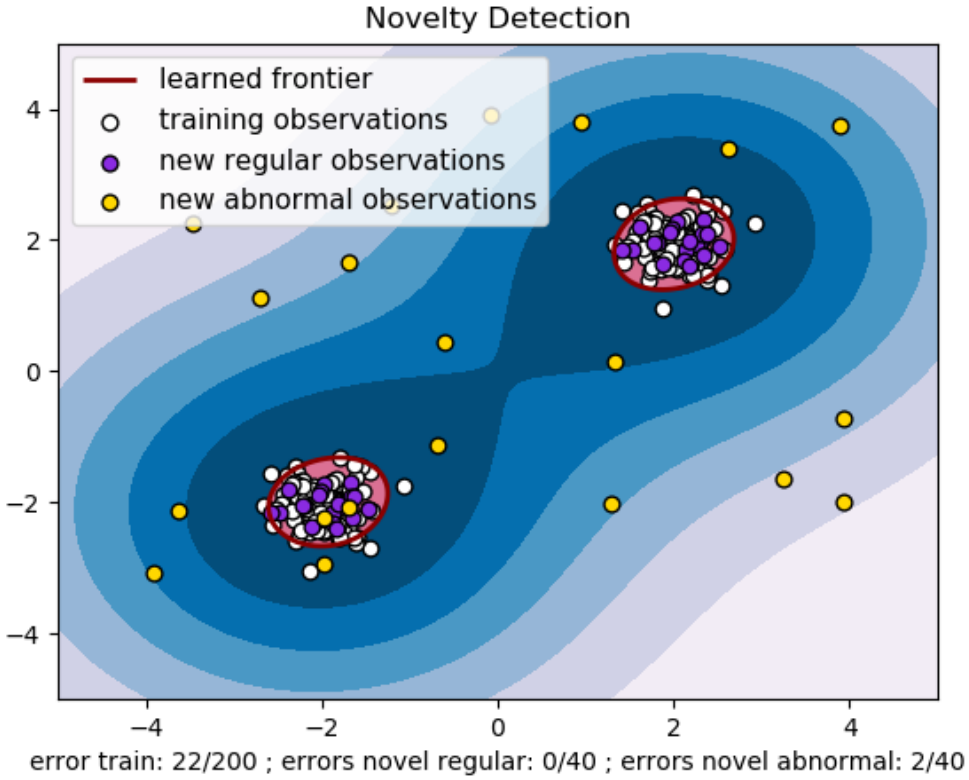


Figure 5.3. Example of One-Class SVM with non-linear kernel (RBF).

5.2.3 Spectrum Kernel

To apply One-class SVM to our problem, we use a particular kind of string kernel, the so-called Spectrum Kernel. This kernel function is designed to be very simple and efficient to compute[9].

In this case, the input space X is constituted by all finite length sequences of characters from an alphabet A, such that $|A| = l$. Given k ≥ 1, the k-spectrum of a sequence of characters is the set of all the k-length (contiguous) sub-sequences that it contains.

The feature map $\Phi_k : X \to \mathbb{R}^{l^k}$ is indexed by all possible sub-sequences a of length k from alphabet A and it is defined as:

$$\Phi_k(x) = \phi_k(x)_{a \in} A^k$$

where $\Phi_a(x)$ are the *occurrences* of sub-sequence a in sequence x. Thus, $\Phi_k(x)$ is a weighted representation of the k-spectrum of input sequence x. Hence, the k-spectrum kernel is defined as:

$$K_k(x, y) = <\Phi_k(x), \Phi_k(y)>$$

Notice that feature vectors are *sparse*: the number of non-zero coordinates is

bounded by $|x|$ - k + 1. This fact leads to the possibility of applying very efficient approaches when computing the kernel function.

5.2.4 Example of the Spectrum Kernel

The k-spectrum kernel is the kernel used in One-Class SVM algorithm for classifying proteins and in our project the discrete trajectories. It is also called string kernel because it works with sequences of characters[5]. Suppose we have a spectrum kernel K and two sequences of characters (strings), s and t:

s= "statistics" t= "pastapistan"

We apply a 3-spectrum kernel, hence we have to find all the subsequences of length 3 in each sequence s and t:

$K_k(s, t)$ = occurrences of common subsequences of length k.
Hence $K_3(s, t)$ = occurrences of common subsequences of length 3.

3-spectrum kernel

s: sta, tat, ati, tis, ist, sti, tic, ics
t : pas, ast, sta, tap, api, pis, ist, sta, tan

	sta	tat	ati	tis	ist	sti	tic	ics	pas	ast	tap	api	pis	tan
Φ(s)	1	1	1	1	1	1	1	1	0	0	0	0	0	0
Φ(t)	2	0	0	0	1	0	0	0	1	1	1	1	1	1

The result in this example is:

$K_3(s, t) = 1{\cdot}2 + 1{\cdot}1 = 3$

5.3 Implementation

The implementation can be divided in two main part: the implementation of the segmentation algorithm and the One-class SVM algorithm. In the first part we implement the sensor log segmentation described in the second section. First of all we have the *"simplify_sensor_log"* script. This module takes in input the SensorLog and returns a new version of SensorLog in which we obtain a text file with a series of characters e.g. "AAABBEEHHGGGGG". Each character represents a sensor and the instant when it is triggered and the character$_i$ matches with <timestamp, sensor name> entry$_i$ of the original sensor log. We show the function that makes the translation.

```
1   def simplify_sensor_log(sensor_log, readable=True):
2   """
3   Translate the given sensor log in a symbols sequence,
4   such that sequence classification techniques can be applied.
5   Notice that, for the sake of readability,
6   we allows only for a maximum number of distinct symbols equals to the size
7   of English alphabet (that is enough according to the scope
8    of this project). Then, in that case, a mapping between
9   sensor ids and letters is automatically computed.
10
11  :type sensor_log: file
12  :param sensor_log: the tab-separated file containing the sensor log.
13  :param readable: whether the mapping between sensor ids
14  and letters has to be computed or not.
15  """
16  file_basename = os.path.splitext(sensor_log.name)[0]
17  dest = file_basename + '_simplified.txt'
18  dest_dict = file_basename + '_simplified_dict.txt'
19  src_reader = csv.reader(sensor_log, delimiter=LOG_ENTRY_DELIMITER)
20  sensor_id_dict = {}
21
22  with open(dest, 'w') as simplified_log:
23          entry = next(src_reader, None)
24          while entry is not None:
25                  sensor_id = entry[SENSOR_ID_POS]
26                  print(sensor_id)
27
28                  if readable:
29                          try:
30                                  translation = sensor_id_dict[sensor_id]
31                          except KeyError:
32                                  translation = SYMBOLS[len(sensor_id_dict)]
33                                  sensor_id_dict[sensor_id] = translation
34                  else:
35                          translation = sensor_id
```

```
36
37                          simplified_log.write(translation + '\n')
38                          entry = next(src_reader, None)
39
40          with open(dest_dict, 'w') as simplified_log_dict:
41                  for k, v in sensor_id_dict.items():
42                          simplified_log_dict.write('%s \t\t %s\n' % (v, k))
```

Then the second module that we have implemented is the "segment_sensor_log.py". It initially creates the topological compatibility matrix. The matrix is built starting from a log of a user: in the code below "PathLog1_simplified.txt" generated previously by the simulator. Then it performs the log segmentation, in which the sensor log is the text file "DatasetPaths_simplified.txt".

```
1           if __name__ == '__main__':
2                   import time
3                   from datetime import timedelta
4                   SRC_LOG_UNICO = os.path.join(DATA_FOLDER, 'PathLog1_simplified.txt')
5
6                   SRC = os.path.join(DATA_FOLDER, 'DatasetPaths_simplified.txt')
7                   COMPAT_THRESHOLD_ = 0.1
8                   NOISE_THRESHOLD_ = 3
9                   SENSOR_ID_POS_ = 0
10                  NGRAMS_LENGTH = 3
11
12                  if NOISE_THRESHOLD_ < NGRAMS_LENGTH:
13                          raise ValueError('The minimum sequence length must be
14                          greater or equal than n-grams length.')
15
16          start_time = time.time()
17          print('Building topological compatibility matrix...')
18          with open(SRC_LOG_UNICO, 'rb') as log:
19                  tcm = TopologicalCompatMatrix(sensor_log=log,
20                  sensor_id_pos=SENSOR_ID_POS_)
21          #print(np.matrix(tcm.prob_matrix))
22
23          print('Performing log segmentation...')
24          with open(SRC, 'rb') as log:
25                  ssl = SegmentedSensorLog(sensor_log=log, top_compat_matrix=tcm,
26                  sensor_id_pos=SENSOR_ID_POS_,
27                  compat_threshold=COMPAT_THRESHOLD_,
28                  noise_threshold=NOISE_THRESHOLD_)
29
30
31          elapsed_time = (time.time() - start_time)
32          print('Segmentation time:', timedelta(seconds=elapsed_time))
33
34
```

```
35          max_vector_length_ = build_sequence_clf_training_set(ssl, SENSOR_ID_POS_,
36          NGRAMS_LENGTH)
37
38          # build a validation set for a sequence classifier
39          build_sequence_clf_validation_set(ssl, SENSOR_ID_POS_, NGRAMS_LENGTH,
40          max_vector_length_)
41
42          # show segmented log statistics
43          ssl.plot_stats()
```

The function "_find_segments" in the code below performs the segmentation algorithm described in the pseudocode in the second section "Sensor Log Segmentation". It takes in input the SensorLog and returns an object, the SegmentedSensorLog, formed by b_step sets, that are sets of segments. Each b_step set is composed by a subset of closed segments, that are the segments before the crossing of the trajectories, and a subset of compatible segments, that follow the crossing.

```
1           def _find_segments(self, sensor_log):
2                   """
3                   Find segments in the given sensor log.
4
5                   :type sensor_log: file
6                   :param sensor_log: the tab-separated file
7                   containing the sensor log.
8                   """
9                   sensor_log_reader = csv.reader(sensor_log,
10                  delimiter=LOG_ENTRY_DELIMITER)
11                  open_segments = []
12                  for measure in sensor_log_reader:
13
14                          sensor_id = measure[self.sensor_id_pos]
15
16                          # find compatible open segments
17                          compat_segments_idxs =
18                          self._get_compat_segments_indices(open_segments, sensor_id)
19
20                          # check compatibility results
21                          if len(compat_segments_idxs) == 1:
22                                  # only one compat segment exists,
23                                  #append the measure
24                                  segment_idx = compat_segments_idxs[0]
25                                  open_segments[segment_idx].append(measure)
26
27                          else:
28                                  if len(compat_segments_idxs) > 1:
29                                  # if many compat segments exist, close them (B-step)
30                                          self._close_segments(open_segments,
31                                          compat_segments_idxs)
```

```
32
33                                          # open new segment and append the measure
34                                          new_segment = [measure]
35                                          open_segments.append(new_segment)
36
37                                          # check whether the new segment is compatible
38                                          #with at least a segment in last B-step
39                                          if self.b_steps:
40                                                  if self._get_compat_segments_indices(
41                                                  self.b_steps[-1].closed_segments,
42                                                  sensor_id):
43
44                                                          # add new segment to last B-s
45                                                          self.b_steps[-1]
46                                                          .add_compat_segment(new_segme
47
48                                                          self.b_steps[-1].crossing_poi
49                                                          .append(str(measure[0])+ "_"
50                                                          +str(sensor_log_reader.line_n
51
52
53                  # close remaining open segments
54                  self._close_segments(open_segments)
```

The second part regards the One-class SVM algorithm[11]. It is implemented in the "train_svm.py" script, as showed in the code below. Initially the dataset generated from the sensor log segmentation is loaded. Then all the segments with length less than three are filtered and removed from the dataset. At the end the algorithm performs the training phase calling the OneClassSVM function.

```
1           if __name__ == '__main__':
2
3                   NOISE_THRESHOLD = 10
4
5                   print('Loading dataset...')
6                   clf_input = SequenceClassifierInput(cached_dataset=
7                   '1568968144_3_7306_GOOD')
8                   train_data, test_data, *_ = clf_input.get_spectrum_train_test_data()
9                    # ignoring labels
10
11                  # SequenceClassifierInput splits the dataset in train
12                  and test by default.
13                  # Since the validation is performed separately, we join the splits.
14                  train_data = train_data + test_data
15
16                  # Filter out short sequences from dataset.
17                  print('Filtering dataset...')
18                  filter_dataset(train_data, NOISE_THRESHOLD, clf_input.ngrams_length)
```

```
19
20                  print('Training One-class SVM...')
21                  clf = svm.OneClassSVM(kernel=occurrence_dict_spectrum_kernel)
22                  start_time = time.time()
23                  clf.fit(train_data)
24                  elapsed_time = (time.time() - start_time)
25                  print('\tTime:', timedelta(seconds=elapsed_time))
26                  print('\tNoise threshold:', NOISE_THRESHOLD)
27
28                  print('Creating model dump...')
29                  model_checkpoint_time = str(int(time.time()))
30                  model_checkpoint_filename = os.path.join(TRAINED_MODELS_FOLDER,
31                  model_checkpoint_time + PICKLE_EXT)
32                  joblib.dump(clf, model_checkpoint_filename)
33                  print(model_checkpoint_filename)
```

5.4 Experiments

We have tested the performances of the One-Class SVM, changing several parameters: the module of the velocity, the type of the motion, uniform or various, the number of the persons moving in the house, the radius of the PIR sensors. The dataset includes the motions of the users moving inside the house for three days. The performances are tested using "validate_svm.py" script. This module has been implemented using a Python library "joblib". Joblib allows to save the model to file and load it later in order to make predictions. The model is stored in a pickle file, ".pkl". Pickle is the standard way of serializing objects in Python. We use the pickle operation to serialize the machine learning algorithm and save the serialized format to a file. After we load this file to deserialize the model and use it to make new predictions. The model is saved to file and load it to make predictions on the unseen test set. The accuracy of this algorithm is calculated in this way:

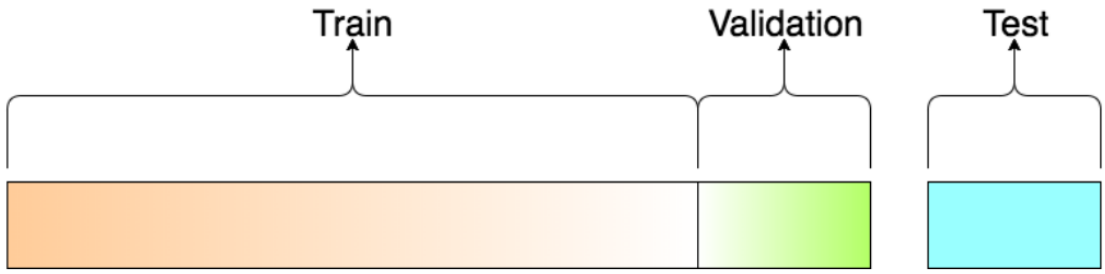
$$accuracy(\%) = \frac{good\_sequences\_number \ * \ 100}{total\_sequences\_number}$$

Below we show an example.

```
1           if __name__ == '__main__':
2
3           NOISE_THRESHOLD = 15
4
5           print('Loading model dump...')
6           predictions_filename = os.path.join(TRAINED_MODELS_FOLDER, '1561476910.pkl')
7           clf = joblib.load(predictions_filename)
8
9           print('Loading validation data...')
10          clf_input = SequenceClassifierInput(cached_dataset=
11          '1561471958_3_26387_GOOD')
```

```
12          train_data, test_data, *_ = clf_input.get_spectrum_train_test_data()
13          # ignoring labels
14
15          # SequenceClassifierInput splits the dataset in train and test by default.
16          # We join them to perform validation.
17          validation_data = train_data + test_data
18
19          # Filter out short sequences from dataset.
20          print('Filtering dataset...')
21          filter_dataset(validation_data, NOISE_THRESHOLD, clf_input.ngrams_length)
22          print('\tFiltered dataset size:', str(len(validation_data)))
23
24          # compute predictions and show stats
25          print('Computing predictions...')
26          start_time = time.time()
27          predictions = clf.predict(validation_data)
28          elapsed_time = (time.time() - start_time)
29
30          total_sequences_num = len(validation_data)
31          good_sequences_num = sum(1 for _ in filter(lambda x: x == 1, predictions))
32           # count positive predictions
33          print('\tTime:', timedelta(seconds=elapsed_time))
34          print('\tFraction of good sequences: {:3.1f}%'.format(good_sequences_num /
35          total_sequences_num * 100))
36
37          # dump results
38          print('Dumping predictions...')
39          predictions_info = [str(int(time.time())), 'l_min',
40          str(NOISE_THRESHOLD), 'predictions']
41          predictions_filename = os.path.join(DATA_FOLDER,
42          FILENAME_SEPARATOR.join(predictions_info) + PICKLE_EXT)
43          with open(predictions_filename, 'wb') as dump:
44          pickle.dump(predictions, dump)
```

Briefly we explain what is validation dataset, training set and test dataset and their uses. The training set is the sample of data used to fit the model.The validation dataset is the sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration. The test dataset is the sample of data used to provide an unbiased evaluation of a final model fit on the training dataset. The Test dataset provides the gold standard used to evaluate the model. It is only used once a model is completely trained(using the train and validation sets). The validation set is used to evaluate a given model, but this is for frequent evaluation. Many a times the validation set is used as the test set, but it is not good practice. The test set is generally well curated. It contains carefully sampled data that spans the various classes that the model would face, when used in the real world.



A visualisation of the splits

5.4.1 Uniform motion with radius of the sensors one meter

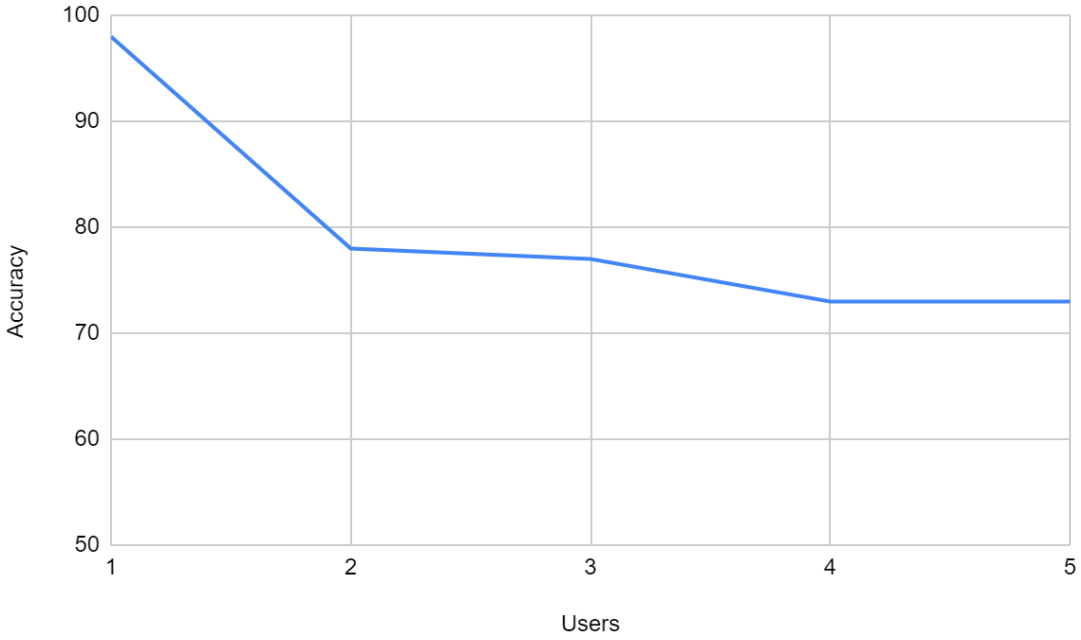


Figure 5.4. The graph shows the accuracy-number of users relation, with a constant velocity in module of 0.25 m/s.

In the Figure 5.4 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 0.25 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after four users the accuracy decreases considerably under 75%. This effect is due to a greater presence of crossing trajectories and the big growth of the number of the segments.

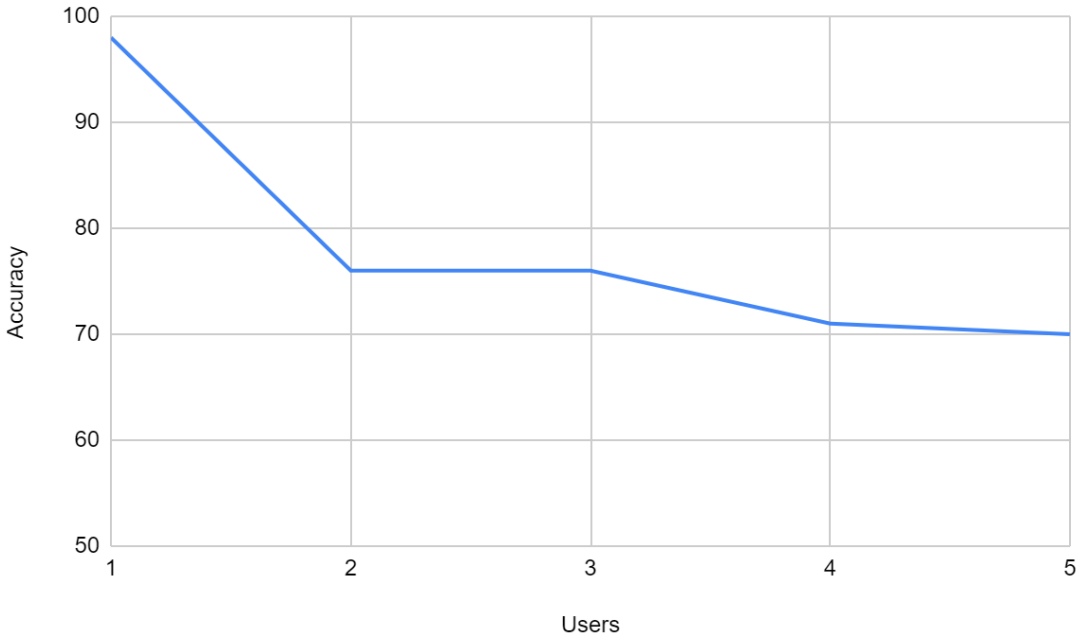


Figure 5.5. The graph shows the accuracy-number of users relation, with a constant velocity in module of 0.5 m/s.

In the Figure 5.5 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 0.5 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after four users the accuracy decreases considerably under 75%. This effect is due to a greater presence of crossing trajectories and the big growth of the number of the segments. It is very similar to the previous case.

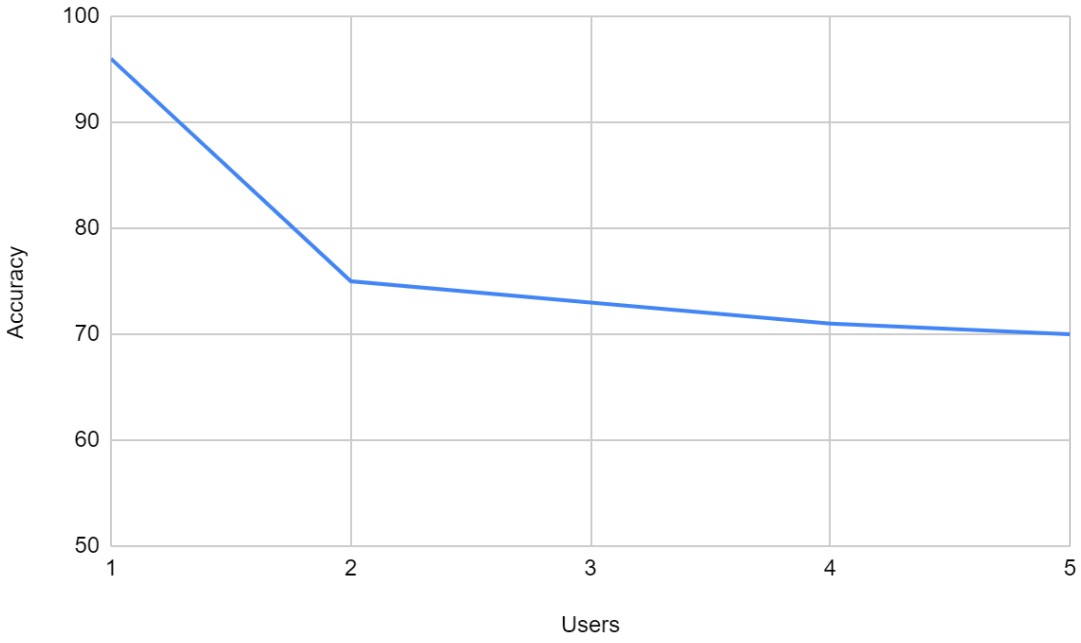


Figure 5.6. The graph shows the accuracy-number of users relation, with a constant velocity in module of 1 m/s.

In the Figure 5.6 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 1 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after four users the accuracy decreases considerably under 75%. This effect is due to a greater presence of crossing trajectories and the big growth of the number of the segments. In this case another cause is the high level of discretization of the trajectories: the positions in the trajectory are a meter away among them.

5.4.2 Uniform motion with radius of the sensors two meters

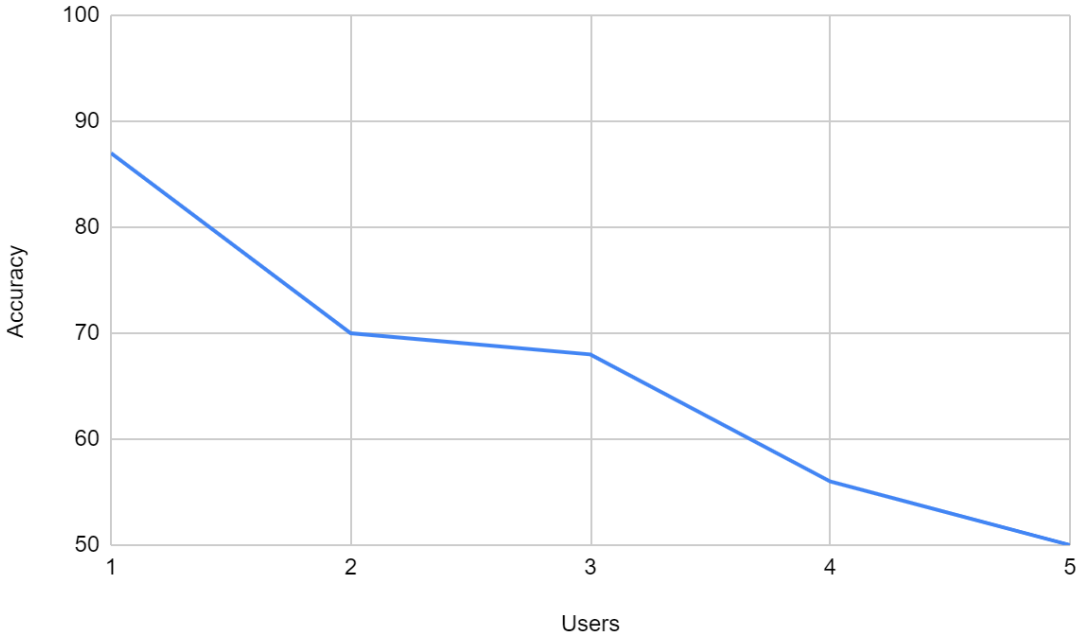


Figure 5.7. The graph shows the accuracy-number of users relation, with a constant velocity in module of 0.25 m/s and radius of the PIR sensors 2 meters.

In the Figure 5.7 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 0.25 m/s, the motion is uniform. We can notice that augmenting the number of the users expecially after two users the accuracy decreases considerably under 70%. This effect is due to a greater presence of crossing trajectories and the big growth of the number of the segments. In this case another cause is the high level of discretization of the trajectories: the sensors that generate the trajectories have a cone of shadow of 2 meters. This implies a greater inaccuracy.

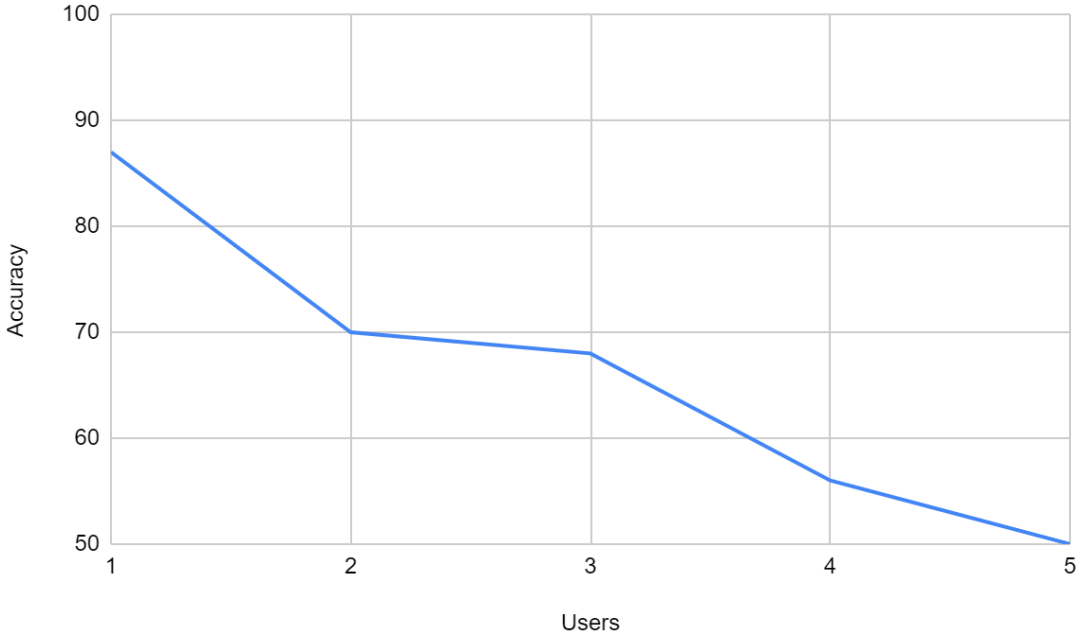


Figure 5.8. The graph shows the accuracy-number of users relation, with a constant velocity in module of 0.5 m/s and radius of the PIR sensors 2 meters.

In the Figure 5.8 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 0.5 m/s, the motion is uniform. We can notice that augmenting the number of the users after two users the accuracy decreases considerably under 70%. This effect is due to a greater presence of crossing trajectories and the big growth of the number of the segments. In this case another cause is the high level of discretization of the trajectories: the sensors that generate the trajectories have a cone of shadow of 2 meters. This implies a greater inaccuracy. After the number of four users we can notice that the accuracy is between 60% and 50%, hence it is very imprecise for a good classification of the trajectories.

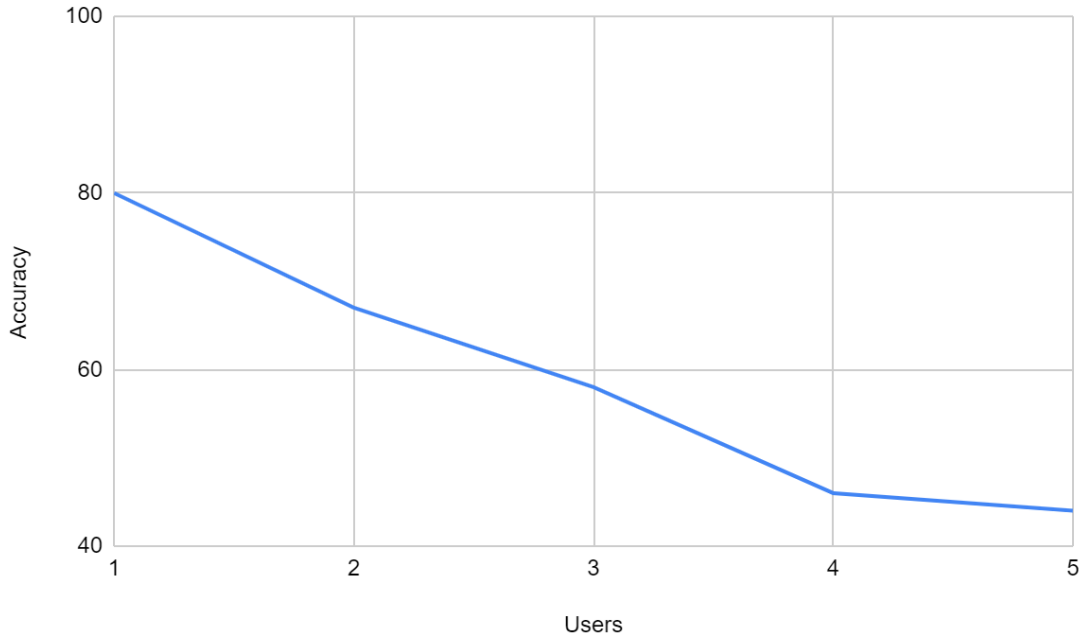


Figure 5.9. The graph shows the accuracy-number of users relation, with a constant velocity in module of 1 m/s and radius of the PIR sensors 2 meters.

In the Figure 5.9 we show the relation between the number of the users and the accuracy. In this graph the speeds of all users in the house are constants with a velocity of 1 m/s, the motion is uniform. We can notice that augmenting the number of the users after two users the accuracy decreases considerably under 70%. This effect is due to a greater presence of crossing trajectories and the big growth of the number of the segments. In this case another cause is the high level of discretization of the trajectories: the sensors that generate the trajectories have a cone of shadow of 2 meters. This implies a greater inaccuracy. After the number of four users we can notice that the accuracy is between 50% and 40%, hence it is very imprecise for a good classification of the trajectories.

5.4.3 Various motion

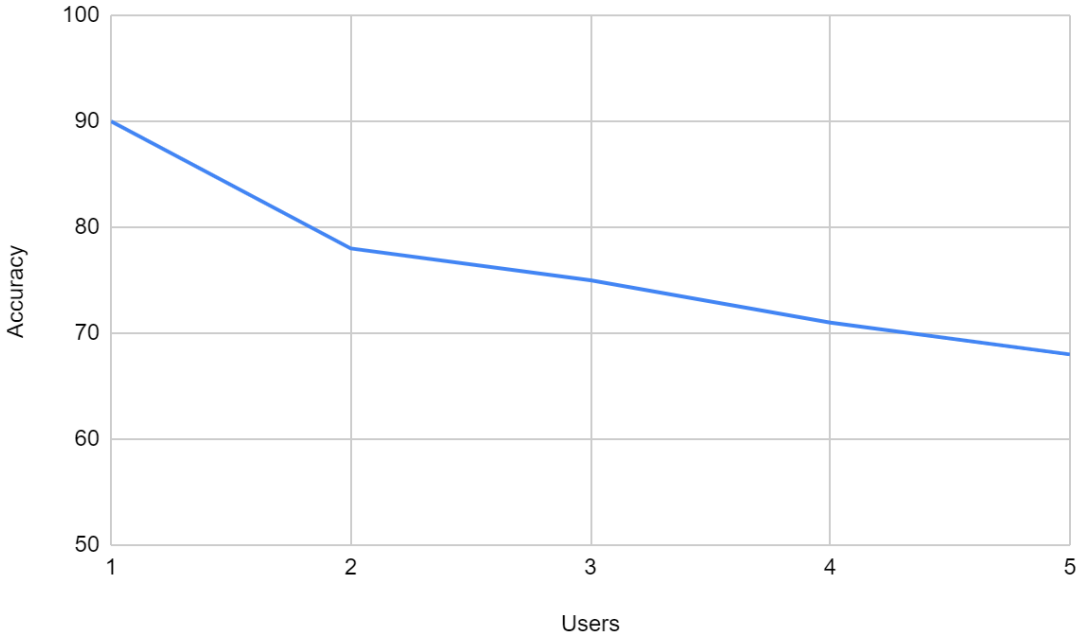


Figure 5.10. The graph shows the accuracy-number of users relation, with a variable velocity and the radius of the PIR sensors of one meter.

In the Figure 5.10 we show the relation between the number of the users and the accuracy. The users move with a various motion. We can notice that augmenting the number of the users expecially after four users the accuracy decreases under 70%. This effect is due to a greater presence of crossing trajectories, and the big growth of the number of the segments.

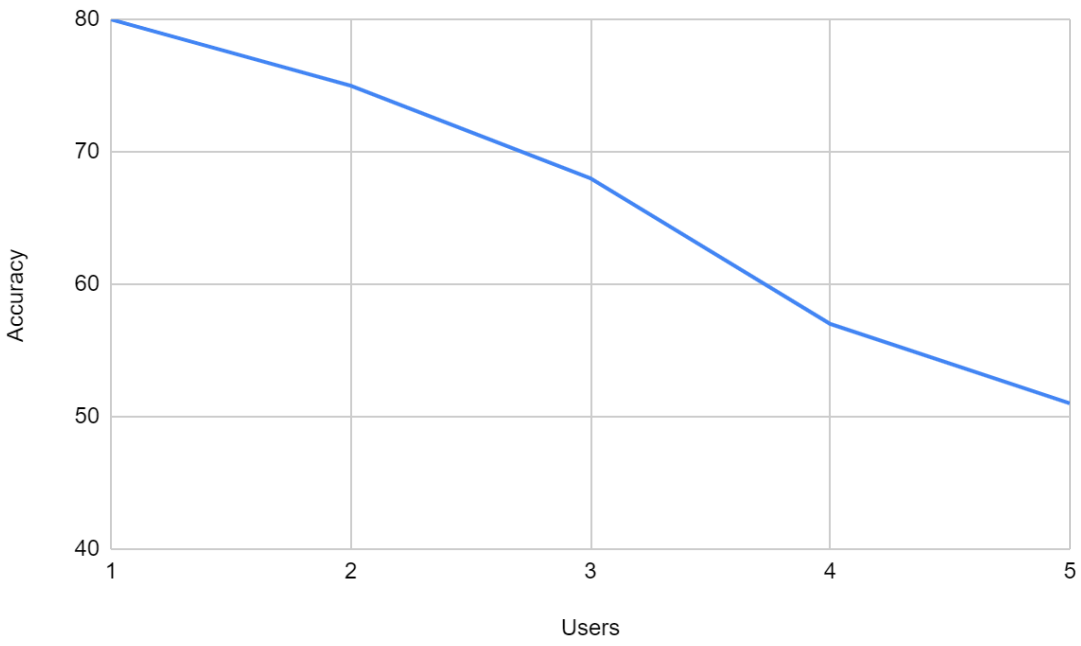


Figure 5.11. The graph shows the accuracy-number of users relation, with a variable velocity and the radius of the PIR sensors of two meters.

In the Figure 5.11 we show the relation between the number of the users and the accuracy. The users move with a various motion. We can notice that augmenting the number of the users expecially after three users the accuracy decreases under 70%. This effect is due to a greater presence of crossing trajectories, and the growth of the number of the segments. Another consideration is that with one user the accuracy is equal to 80%. In this case the cause is the high level of discretization of the trajectories: the sensors that generate the trajectories have a cone of shadow of 2 meters. This implies a greater inaccuracy. After the number of four users we can notice that the accuracy is between 60% and 50%, hence it is very imprecise for a good classification of the trajectories.

5.5 Conclusion

We can conclude with two considerations. The first result is that using sensors with a radius of the cone shadow with 2 meters gives a bad accuracy, also with three users. This issue can be solved using sensors with a radius of one meter. Another consideration is that both in the case of uniform and variable velocity the accuracy with sensors of one meter is between the 70% and 80% with multiple users. This result can be a little improved changing several details: increasing the size of the dataset and so the number of the days the simulation. Another improvement can be the implementation of the algorithm of the segmentation. Data structures more performing can be used for the storage of the segments and so to obtain a best execution time of the algorithm.

Chapter 6

A naïve algorithm for the discrete trajectories

6.1 Introduction

We have considered by the results of the One-class SVM algorithm to design and implement a naïve algorithm more performing and without the machine learning. This algorithm reuses the concept and the technique of the segmentation and works on the **discrete trajectories**. It can be divided in two main phase. In the first part the algorithm segments the sensor log and returns the set of the segments. Then in the second phase it takes in input the segments and outputs the joined segments in each crossing point. For the join of the sequences in the intersections it applies the paths of the users that occur more frequent. We analyse it more in details in the next section.

6.2 Design of the algorithm

In this section we describe the naïve algorithm used for the reconstruction and the classification of the users trajectories. The first part of the algorithm consists of the segmentation of the sensor log. We show the pseudocode below.

Algorithm Sensor Log Segmentation

$\boldsymbol{S} = \emptyset$

for all $m \in L$ **do**
    if there exists a single *open* $s \in \boldsymbol{S}$ *compatible* with m **then**
        $s = s \parallel m$
    if there exists *open* s_1,..., $s_n \in \boldsymbol{S}$ *compatible* with m **then**
        mark s_1,..., $s_n \in \boldsymbol{S}$ as *closed*
        create a new *open* segment *s'* initialized with m
        $\boldsymbol{S} = \boldsymbol{S} \cup s'$

if there exists no *open* $s \in \boldsymbol{S}$ *compatible* with m **then**
create a new *open* segment *s'* initialized with m
$\boldsymbol{S} = \boldsymbol{S} \cup s'$
mark all *open* $s \in \boldsymbol{S}$ as *closed*
remove all $s \in \boldsymbol{S}$ such that $|s| < N$

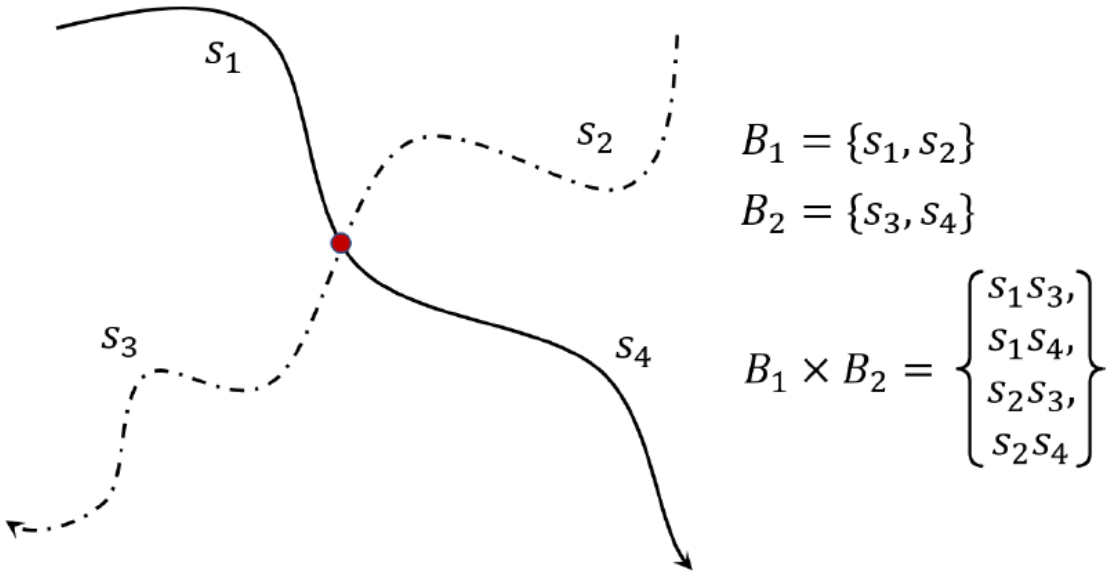


Figure 6.1. Two (intersecting) single-user trajectories and the corresponding B-step.

The algorithm returns a set of crossing segments, called B-step sets as in Figure 6.1, B_1, B_2. We will call B_1 as the "closed segments" and B_2 as the "compatible segments".

In the second phase we have to join and make the cartesian product of each segment, for obtaining the B-step set B_1 X B_2, that corresponds to the all possible combinations of the reconstructed trajectories. Then for each joined sequence of the B-step set B_1 X B_2, the algorithm extracts the subsequences composed by three positions before the crossing point and three after. For example as in Figure 6.1, s_1s_3 is a joined segment, and we suppose that it is formed by this sequence of the positions "AAABBEHHHTTT" and the crossing position is E. The correspondent subsequence will be s = "ABBEHHHT". Then the algorithm gets all the subsequences from the joined segments e.g. s_1s_3, s_1s_4, s_2s_3,s_2s_4. It calculates the occurrences of the subsequence s:

$s = \sum_{i=0}^{n} s_i$ where n is the number of the segments in which appears s

After this in the next step, the algorithm chooses the subsequence with the greater occurrences between its identical closed segments. For example, the subsequence s belongs to the joined segment s_1s_3 and its closed segment is s_1. The algorithm compares the occurrences of s with the subsequence s' belonging to s_1s_4 and with the closed segment s_1. As in Figure 6.1 the hoped result should be that the occurrences of s' is greater than s. In this case the algorithm chooses s_1s_4 and removes from the set of the segments the combination s_1s_3.

Below we show the pseudocode of this second part of the algorithm.

Algorithm

Input : **SSL** // Segmented Sensor Log

SSLJOINED = ∅ // Segmented Sensor Log with joined segments
FINALSET = ∅ // final set of the trajectories

for all *B-step set* ∈ *SSL* **do**
  for all *closed segment* ∈ *B-step set* **do**
    for all *compatible segment* ∈ *B-step set* **do**
      B_1 X B_2 = closed segment ∪ compatible segment //the cartesian product set
      SSLJOINED = **SSLJOINED** ∪ B_1 X B_2

for all *B-step set* ∈ **SSLJOINED do**
  for all *segment* ∈ *B-step set* **do**
    subsequence = GetSubsequence(*segment*)
    occurrences = CalculateOccurrences(subsequence)
    FINALSET = **FINALSET** ∪ ChooseSegmentGreaterOccurrences(occurrences, *B-step set*)

We suppose that m is the greater size of the segmented sensor log, l is the greater size of the list of the closed segments and h is the greater size of the list of the compatible segments. The computational cost of the algorithm in the worst case is O(mlh) and so linear.

6.3 Implementation of the algorithm

The algorithm has been implemented in Python3. The phase of the reconstruction of the paths is in the "reconstruct_paths.py". We show the code below.

```
def reconstruct(ssl):

        sensor_log= SensorLog()


        for i in range(0, len(ssl.b_steps)):

                b_step= B_step()

                b_step.closed_segments=[]
                b_step.compat_segments=[]

                for j in range(0, len(ssl.b_steps[i].closed_segments)):
                        if (len(ssl.b_steps[i].closed_segments[j]) <= 4  ):
                                continue
                        s = ""
                        for h  in range(0,len(ssl.b_steps[i].closed_segments[j])):
                                s+=ssl.b_steps[i].closed_segments[j][h][0]

                                b_step.closed_segments.append(s)

                        for m in range(0, len(ssl.b_steps[i].compat_segments)):
                                if(len(ssl.b_steps[i].compat_segments[m])<=4  ):
                                        continue

                                x = ""
                                for u in range(0, len(ssl.b_steps[i].compat_segments[
                                        x+=ssl.b_steps[i].compat_segments[m][u][0]
                                        b_step.compat_segments.append(x)

                        b_step.crossing_points=ssl.b_steps[i].crossing_points

                sensor_log.b_steps.append(b_step)


        final_array=[]

        for i in range(0, len(sensor_log.b_steps)):

                b_step=[]
                for j in range(0, len(sensor_log.b_steps[i].closed_segments)):

```

```
                        for m in range(0, len(sensor_log.b_steps[i].compat_segments)):
                                if(len(sensor_log.b_steps[i].closed_segments[j]+
                                sensor_log.b_steps[i].compat_segments[m])!=0):
                                joined_segment=Joined_Segment()
                                joined_segment.id_closed=j
                                joined_segment.id_compat=m
                                joined_segment.subsequence=sensor_log.b_steps[i].
                                closed_segments[j][-3:]+sensor_log.b_steps[i].
                                compat_segments[m][:3]

                                joined_segment.segment=sensor_log.b_steps[i].
                                closed_segments[j]+
                                sensor_log.b_steps[i].compat_segments[m]
                                joined_segment.index_crossing=
                                len(sensor_log.b_steps[i].closed_segments[j])-1
                                joined_segment.position_crossing=get_crossing_point(
                                sensor_log.b_steps[i].crossing_points,
                                sensor_log.b_steps[i].compat_segments[m][:1])
                                b_step.append( joined_segment)
                                if( b_step):
                                        final_array.append(b_step)
        print(len(final_array))

        array_joined_segments=[]
        for i in range(0, len(final_array)):
                for j in range(0, len(final_array[i])):
                        joined_segment=final_array[i][j]
                        joined_segment.occurences=calculate_occurences(
                        joined_segment.subsequence, sensor_log)

        for i in range(0, len(final_array)):
                b_step=[]
                for j in range(0, len(final_array[i])):
                        joined_segment = final_array[i][j]
                        max_joined_segment=calculate_max_occurences(joined_segment,
                        final_array[i])
                        if(max_is_contained(max_joined_segment, b_step)==False):

                                b_step.append(max_joined_segment)
                                array_joined_segments.append(b_step)


        construct_database(array_joined_segments,
        "C:\\Users\\Dario\\Desktop\\multi-user-segmentation-master\\
        \\data\\trajectoriesDB.db")

        return
```

6.4 Test and results

We have tested the performances of the algorithm, changing several parameters: the module of the velocity, the type of the motion, uniform or various, the number of the persons moving in the house. The dataset includes the motions of the users moving inside the house for three days. Each PIR sensor has the radius of one meter. The performances are tested using "validate_performance.py" script. The accuracy of this algorithm is calculated in this way:

$$accuracy(\%) = \frac{good\_sequences\_number \ * \ 100}{total\_sequences\_number}$$

We show below the code of the validation.

```
def validate_algorithm():
        list_log_files=read_path_logs()
        conn = sqlite3.connect("C:\\Users\\Dario\\Desktop\\
        multi-user-segmentation-master\\data\\trajectoriesDB.db")
        conn.set_authorizer(select_authorizer)
        c = conn.cursor()

        array = c.execute("SELECT * FROM segment;").fetchall()
        final_array=[]
        for segment, orders_iter in itertools.groupby(array,lambda x: x[2]):
                b_step = list(orders_iter)
                final_array.append(b_step)
                conn.close()

        correct_decisions=0
        wrong_decisions = 0
        for i in range(0, len(final_array)):
                for j in range(0, len(final_array[i])):
                        segment = final_array[i][j][1]
                        crossing =  final_array[i][j][3]
                        index_crossing = int( final_array[i][j][4])
                        if (find_segment(segment, crossing, list_log_files,
                        index_crossing)):
                                correct_decisions += 1
                                break
                        if (j==len(final_array[i])-1 and find_segment(segment,
                        crossing, list_log_files, index_crossing)==False):
                                wrong_decisions += 1

        print(correct_decisions)
        print(wrong_decisions)
        accuracy=(correct_decisions*100)/(correct_decisions+wrong_decisions)
        print(accuracy)
        return
```

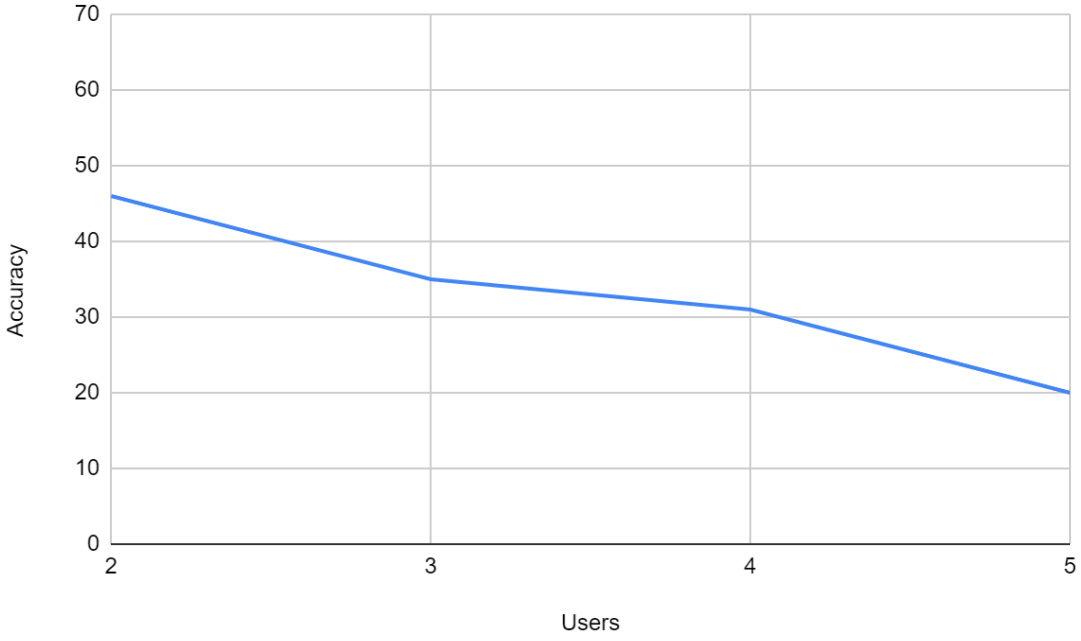


Figure 6.2. The graph shows the accuracy-number of users relation, with the constant velocity of 0.25 m/s.

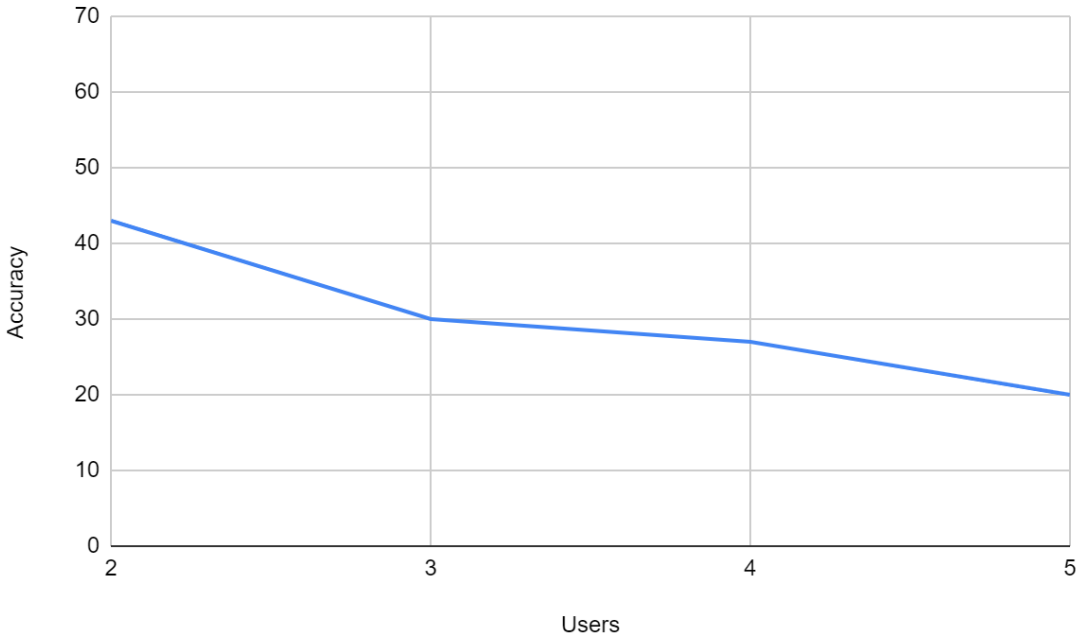


Figure 6.3. The graph shows the accuracy-number of users relation, with the constant velocity of 0.5 m/s.

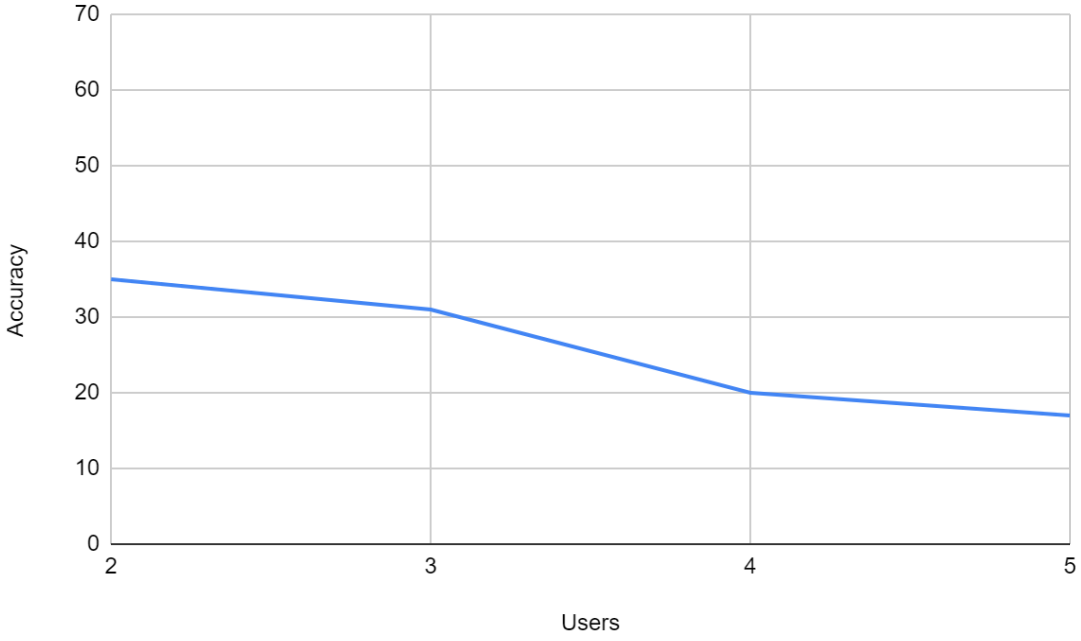


Figure 6.4. The graph shows the accuracy-number of users relation, with the constant velocity of 1 m/s.

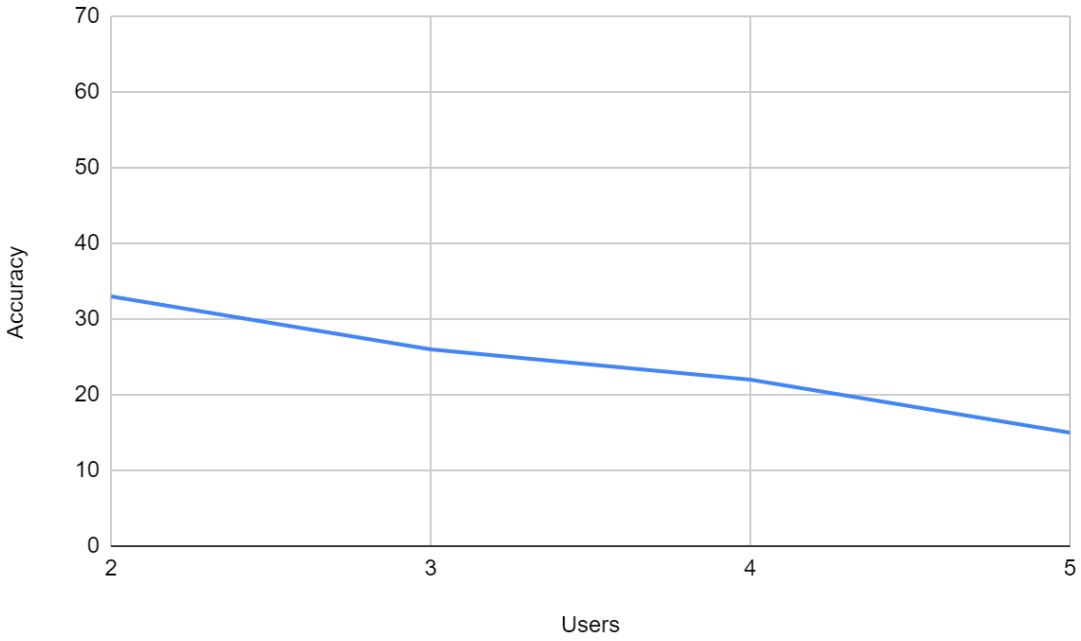


Figure 6.5. The graph shows the accuracy-number of users relation, with a variable velocity.

The results of this attempted algorithm are not good. The accuracies in the several analysed cases doesn't reach either the 60%. We can conclude that the best method for solving the issue of the classification of trajectories is the One-class SVM.

Chapter 7

Conclusion

The thesis work has returned several important results. I have developed a simulator for emulating a smart space and the habits of the users that move inside. The simulator allows to generate the trajectories of the users, to draw them. We can create the 2D map of different indoor environments e.g. house, office. The software allows to customize these spaces, changing the habitual objects, the targets e.g. the bed or sofa in a smart house. It permits to configure the sensors for generating the discrete trajectories and it give the possibility to generate trajectories with different features: velocity, granularity, type of motion. The graphic representation of the trajectories is very accurate using the spline functions, for obtaining smoothed, curvilinear and more realistic trajectories. Another important aspect is that the simulator works as a generator of the datasets, in this case datasets of trajectories. It can be improved changing the granularity of the data, creating a best graphical user interface, adding the possibility to customize the habits of each person that lives in the smart space. This software can be used in the future for other researches in the field of the smart environments and the home automation.

Another important consideration is that the analysis of the continuous trajectories and the utilization of algorithms as Kalman filter cannot be applied in the discrete trajectories generated by the triggering of the PIR sensors. The result implies that we can use with a good precision the algorithm with the Kalman filter in the case of the dense and continuous trajectories and not with the rarefied and discrete trajectories, due to the high discretization and so a coarse granularity of the data. In this case machine learning algorithms as One-Class SVM with segmentation algorithm can be useful for obtaining good results. This method returns both in the case of uniform and variable velocity an accuracy with sensors of one meter is between the 70% and 80% with multiple users. This result can be a little improved changing several details: increasing the size of the dataset and so the number of the days the simulation. Another improvement can concern the implementation of the algorithm of the segmentation. Data structures more performing could be used for the storage of the segments and so to obtain a best execution time of the algorithm.

Acknowledgments

Vorrei ringraziare la mia famiglia per avermi dato l'opportunità di studiare e intraprendere questa esperienza formativa, supportandomi e motivandomi. Ringrazio tutti i miei amici e i colleghi incontrati durante il corso di Laurea. Infine ringrazio il relatore Prof. Massimo Mecella e il correlatore Ing. Francesco Leotta per la disponibilità e il supporto ricevuto durante il lavoro di tesi. Dedico questo importante traguardo a tutte le persone che mi sono state sempre vicine.
Dario

Bibliography

[1] Pykalman 0.9.2 documentation. Available from: https://pykalman.github.io/.

[2] Github repository project (2019). Available from: https://github.com/dariolitardi/projecthabitmining.

[3] Novelty and outlier detection (2019). Available from: https://scikit-learn.org/stable/modules/outlier_detection.html#outlier-detection.

[4] BZARG. How a kalman filter works, in pictures (2015). Available from: http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/.

[5] Fukumizu, K. Kernel methods for statistical learning (2012). Available from: http://yosinski.com/mlss12/MLSS-2012-Fukumizu-Kernel-Methods-for-Statistical-Learning/.

[6] Leotta, F., Mecella, M., and Sora, D. Visual analysis of sensor logs in smart spaces: Activities vs. situations. *IEEE*. Available from: https://ieeexplore.ieee.org/document/8405699.

[7] Leotta, F., Mecella, M., Sora, D., and Catarci, T. Surveying human habit modeling and mining techniques in smart spaces. *futureinternet*, (2019). Available from: https://www.mdpi.com/journal/futureinternet.

[8] Leotta, F., Mecella, M., Sora, D., and Spinelli, G. Pipelining user trajectory analysis and visual process maps for habit mining. *IEEE*, (2017). Available from: https://ieeexplore.ieee.org/document/8397509.

[9] Leslie, C. S., Eskin, E., and Noble, W. S. The spectrum kernel: A string kernel for svm protein classification. *PSB 2002, Lihue, Hawaii, USA*, pp. 566–575. Available from: https://www.ics.uci.edu/~welling/teatimetalks/kernelclub04/spectrum.pdf.

[10] Magid, E., Lavrenov, R., and Afanasyev, I. Voronoi-based trajectory optimization for ugv path planning. *2017 International Conference on Mechanical, System and Control Engineering (ICMSC)*. Available from: https://www.researchgate.net/publication/318036048_Voronoi-based_trajectory_optimization_for_UGV_path_planning.

[11] Pedregosa, F., et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*.

[12] Ravankar, A., Ravankar, A. A., Kobayashi, Y., Hoshino, Y., and Peng, C.-C. Path smoothing techniques in robot navigation: State-of-the-art, current and future challenges. *Sensors*. Available from: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6165411/`.

[13] Schon, T. B. and Lindsten, F. Manipulating the multivariate gaussian density. Available from: `http://user.it.uu.se/~thosc112/pubpdf/schonl2011.pdf`.

[14] Vlasveld, R. Introduction to one-class support vector machines (2013). Available from: `http://rvlasveld.github.io/blog/2013/07/12/introduction-to-one-class-support-vector-machines/`.

[15] Walicki, M. and Ferreira, D. R. Sequence partitioning for process mining with unlabeled event logs. *Data Knowl. Eng.*, pp. 821–841.