UNIVERSITÉ LIBRE DE BRUXELLES
Faculty of Science
Computer Science

Master Thesis

Machine Learning for Probabilistic Robotics with Webots

Joan Sebastian Gerard Salinas

Supervisor: Gianluca Bontempi

Academic year 2019-2020

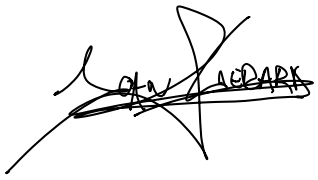
ULB UNIVERSITÉ LIBRE DE BRUXELLES

ULB UNIVERSITÉ LIBRE DE BRUXELLES

Exemplaire à apposer sur le mémoire ou travail de fin d'études,
au verso de la première page de couverture.

Fait en deux exemplaires, **Bruxelles, le** 17/08/2020

Signature



Réservé au secrétariat :	**Mémoire réussi***	**OUI**
		NON

CONSULTATION DU MEMOIRE/TRAVAIL DE FIN D'ETUDES

Je soussigné

NOM : Gerard Salinas

PRENOM : Joan Sebastian

TITRE du travail : Machine Learning for Probabilistic Robotics with Webots

AUTORISE*

REFUSE* (X)

la consultation du présent mémoire/travail de fin d'études par les utilisateurs des bibliothèques de l'Université libre de Bruxelles.

Si la consultation est autorisée, le soussigné concède par la présente à l'Université libre de Bruxelles, pour toute la durée légale de protection de l'œuvre, une licence gratuite et non exclusive de reproduction et de communication au public de son œuvre précisée ci-dessus, sur supports graphiques ou électroniques, afin d'en permettre la consultation par les utilisateurs des bibliothèques de l'ULB et d'autres institutions dans les limites du prêt inter-bibliothèques.

Formulaire version 04/2015

** Biffer la mention inutile*

** Biffer la mention inutile*

I dedicate this work to my parents: Gloria and Arnaud. Their love, endless support and encouragement kept me motivated to cross this long cobblestone road.

Disclaimer

I hereby declare that the work presented in this document is my own. All the sources and aids used are cited in the bibliography section. All texts either quoted directly or paraphrased are indicated by in-text citations. This work has not been submitted before to any institution for assessment purposes.

Acknowledgments

My deepest gratitude to Professor Gianluca Bontempi who offered me his assistance, guidance, and highly valuable ideas throughout the development of this project. I also would like to acknowledge PhD. student Théo Verhelst for proposing me improvement ideas and comments.

My acknowledgments to the Webots support team to help me troubleshooting. Special thanks to David Mansolino to assist me even during out-of-work hours.

I would like to thank the Simón I. Patiño Foundation for funding my studies and letting me expand my knowledge.

My sincere thanks to Camila, for his patience and endless support during my stressful journeys.

Finally, I would like to express my gratitude to my family and friends who encouraged me all along with this enriching and extraordinary experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Where am I?, *Where am I going?*, and *How should I get there?* are questions that a robot should answer to navigate successfully according to Leonard et al. [1]. These questions are about *localization*, defining a *goal*, and *planning* what actions a robot executes to accomplish such a goal. Answering to the first question with a trustworthy and robust solution is essential to answer the other two questions.

The current work focuses on the first question; more concretely, on mobile robot localization.

There is no direct mapping from sensor data to robot localization. Instead, the robot should infer its location from previously acquired knowledge and noisy sensor data. What increments the difficulty of such a task is uncertainty. Robot actions, sensor's measurements, and the environment itself present, all of them, certain degree of uncertainty. Many techniques that deal with uncertainty to localize a robot have been developed lately using probabilistic theory. These techniques have been largely compared, reproduced, justified, and explained using robotics simulator tools first, before grounding them to real robots.

The use of machine learning has increased in the last decade with the development of a high variety of applications in different domains including robotics. The aim of introducing machine learning into the robotics area was to have more intelligent robots. This idea was first mentioned in the 1990s by Brooks [2] presenting a robot architecture that lets the robot decide what to do given the perceived environment.

The thesis is divided into two parts. The first part is oriented to present the important basic concepts in probability, machine learning, probabilistic robotics and the useful components of Webots together with the related work that has been made on the field recently. The second part is dedicated to showing the contributions of the master thesis. It is divided into four chapters, the first chapter presents the architecture used to perform robot localization with Webots and machine learning, the second chapter explains in detail the particle filter algorithm implementation, the third and fourth chapters show the experiments, results, and conclusions made.

1.1 Aim of this work

Robot localization has become popular in the research area due to the high demand of autonomous robots able to navigate precisely in factories, fulfilment centers, houses, hospitals, etc.

This master thesis is oriented to empower existent probabilistic theory techniques used to deal with robot localization by making use of machine learning. These techniques are intended to be implemented, tested, and assessed in simulation.

1.2 Agent and Environment

In order to understand what mobile robot localization is, first, two basic concepts need to be developed: the agent and the environment.

The learner is called *agent* whereas the *environment* is everything the agent interacts with, outside the agent itself [3]. The environment includes the *world* where the agent is put into. The agent *perceives* or *senses* the environment through *sensors* and interacts with it using *actuators*.

A *percept* is the agent's perceptual input at any given time. All the agent's perceptual inputs that the agent has ever perceived is called a *percept sequence*. In other words, the agent *senses* the environment. The actuators, on the other hand, perform *actions* that modify the environment.

The agent's behaviour can be seen as a black box whose inputs are the agent's percepts using sensors and whose outputs are the agent's actions through its actuators as illustrated in figure 1.1. A concrete implementation of this black box is called the *agent program* or *controller*.

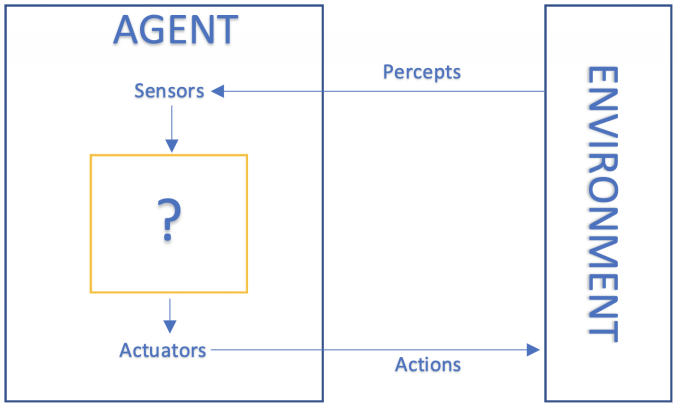


Figure 1.1: The probability density of the robot estimated position, at different steps of the particle filter algorithm.

Source: Extracted from Russell et al. [4] Artificial Intelligence A Modern Approach

The environment has a *state* associated with it that expresses a given configuration at a certain point in time. When the agent performs an action the environment changes its state.

As an example, an agent can be a car, the environment can be the roads, the actuators can be the steering, the accelerator, and the brake, the sensors can be the speedometer, and the cameras.

Environments can be classified, according to Russell et al. [4], using different dimensions as follows:

- *Fully observable* vs. *partially observable*: when the entire environment state is perceived by the agent at any time it is known to be fully observable. An environment is partially observable if only partial information about the environment state is perceived by an agent.
- *Single-agent* vs. *multi-agent*: an agent solving a problem by its own is in a single-agent environment whereas an agent cooperating or competing with other agents is in a multiagent environment.
- *Deterministic* vs. *stochastic*: an environment is deterministic if the next state is determined fully by the current state and the actions performed by the agent; otherwise, it is stochastic.
- *Episodic* vs. *sequential*: on the one hand, an environment is episodic if the agent's experience is divided in episodes such that at every episode the agent executes only one action based on the percept and the next episode does not depend on the action taken in previous episodes. On the other hand, in sequential environments the previously taken actions can affect the future actions to take.
- *Static* vs. *dynamic*: the environment is static when it does not change while the agent is performing a task; otherwise, it is dynamic. For instance, roads are dynamic environments since cars are moving as time goes on, making the environment change.
- *Discrete* vs. *Continuous*: this distinction depends on the state of the environment, the time, the percepts, and actions of the agent.
- *Know* vs. *Unknown*: an environment is known when the agent knows the physical rules that govern the environment before hand; otherwise, the agent needs to learn about them and the environment is said to be unknown.

The environments can also be categorized as *in-door* and *out-door* environments. In-door environments are closed spaces such as the inside of a package-delivery factory, the interior of a house, etc. Out-door environments are open spaces such as the Mars surface, a road from Brussels to Paris, etc.

Russell et al. [4] classify the agents into four categories:

- *Simple reflex agents*: they execute an action depending on the current percept only. They are simple but limited in terms of intelligence.

- *Model-based reflex agents*: they keep an *internal state* with information about previously perceived aspects of the world. This can be useful for the agents when dealing with partially observed environments to keep track of the part of the world that was seen before but it is not being seen now. The internal agent representation of the world is called a *model* of the world.

- *Goal-based agents*: the goal-based agents execute their actions based on the current state, the acquired knowledge so far and a *goal* information that describes the desired environment state.

- *Utility-based agents*: they maximizes their *expected utility* when choosing an action using a *utility function* that measures the agents preferences among states in the world.

- *Learning agents*: a learning agent can be divided into four conceptual components: a *learning element* that makes improvements, a *performance element* responsible for selecting the optimal action, the *critic* that given the percepts it offers some feedback to the learning element. Finally, the *problem generator* that is responsible to suggest whether executing the optimal action (exploiting the current knowledge) or executing a suboptimal action (exploring) that can be bad for the agent in the short term but resulting in a good choice in the long term.

1.3 Robot and Environment

A *robot* is a physical agent that interacts with the environment.

According to Russell et al. [4] there are three robot categories:

- *Manipulators*: they are robotic arms that are often used in industrial factories to automatize the production line. They are physically attached to their workplace.

- *Mobile robots*: unlike manipulators, mobile robots can move through the environment using wheels or robotic legs.

- *Humanoid robots*: they are a mix between mobile robot and manipulators. They are recognized by their physical similarity to humans.

The current work uses mobile robots.

1.4 Mobile Robot Localization

Navigation is about controlling and operating the course of a robot and it is one of the most challenging skills that a mobile robot needs to master to successfully move through the environment. According to Siegwart et al. [5] what is important for a robot to succeed in navigation is to succeed likewise in these four blocks of navigation:

- *Sense:* the robot perceives the environment through its sensors and extracts meaningful data to process and obtain information.
- *Localization:* the robot knows where it is in the environment.
- *Cognition:* the robot creates an execution plan on how to act to reach its goals.
- *Control Action:* the robot executes the execution plan by modulating its output motors.

This project focus on the localization block which is the first block that needs to be addressed to navigate successfully.

Localization in robotics refers to finding where things are in the environment including the robot itself [4]. The *mobile robot localization*, also called *positioning estimation* or *positioning tracking* [6], is a subcategory of robot localization oriented to find out a mobile robot's pose which is placed somewhere in the environment.

As it was mentioned by Thrun et al. [6], most of the robots do not have a noise-free sensor to determine the robot's pose and therefore, this information needs to be inferred from sensor observations.

- *Local vs. global localization*: according to Ferreira et al. [7], in *local localization* (also called position tracking [6]), the robot knows its initial position. The robot keeps track of its position at any time while it is moving. In *global localization* (see also [8]), the robot does not have any information about its initial position and it should be able to infer its pose using the available sensor observations. This technique includes the *kidnapped robot problem* in which a robot that knows its position with respect to the environment is kidnapped and taken into another location. The robot's task is to recover its position. According to Thrun et al. [6] the kidnapped robot's problem should be categorized as a third localization technique given its difficulty.
- *Passive vs. active localization*: in *passive localization* the robot actions are not aimed at facilitating the localization process whereas in *active localization* the actions are selected carefully by the robot to minimize the localization cost, time or error.

- *Localization in static vs. dynamic environments*: the localization process needs to consider if the robot is in a static or a dynamic environment. A task is more difficult in dynamic environments because the dynamic entities need to be considered in the environment state.
- *Single-robot vs. multi-robot localization*: *single-robot localization* deals with one robot positioned in the environment whereas in *multi-robot localization* a group of robots are put into the environment. The former can be used individually in each robot to handle the latter. The advantage of using multiple robots is that a robot r_1 that sees another robot r_2 can help to disambiguate the position of robot r_2 and therefore it can do better; however, robots need to communicate which can cause latency issues when using a lot of robots.
- *Visual robot localization*: a robot can have a camera as a sensor receiving images of the environment as input. The objective of visual robot localization is that the robot is able to localize itself using these images instead of sensor measurement data.

Global localization is more difficult than local localization because it deals with *global uncertainty*. That is, the robot does not have any idea where it is initially. Consequently, a robot can recover its position even if the sensor observations are not very precise using filtering techniques that are studied in section 2.4.4.

1.5 Subsumption Architecture

For moving a mobile robot to a goal position, it is told to execute a certain type of force into the wheels' motor for a period of time rather than going directly to the goal position coordinates. The robot movement is programmed using a *controller* which is a piece of software that tells the robot what to do after sensing the environment until a *control objective* is achieved.

Reactive controls are oriented to deal with simple reflex agents. Most of the actions a robot can execute are separable in subcomponents [2]. These subcomponents can be modeled as finite state machines that react to sensor feedback [4]. They are model-free and feedback-driven controllers. The subcomponents are called *behaviors* that can emerge from the interaction between a simple controller and a complex environment, in this case they are called *emergent behaviors* [4]. They adapt well to a changing world and they are appropriate for making low-level decisions in real-time [4].

Subsumption architecture emerged in the 1990s due to the need of having more intelligent robot controllers which model robot behaviors that are reactive to environment changes and sensor measurement errors.

The subsumption architecture was developed by Rodney Brooks [2]. The implementation consists of networks made of reactive controllers. A *node* or *state* often

contains sensor conditions that determine to stay in or leave that state. It is possible to go from one state (the *transmitter*) to another (the *receiver*) through *arcs* containing messages that are the output of the recently left state. Sometimes these arcs represent physical wiring and the messages are the signals sent to the robot's motor. These augmented finite state machines (AFSM) have internal clocks that control the time it takes to traverse an arc, the word *augmented* refers to such internal clocks. Two or more simple networks representing simple behaviors can be connected one to another to form more complex networks and thus to represent more complex behaviors. This process is called *layering*. For instance, a layer that will make the robot explore the environment, called *explore*, can be composed of *avoid* and *move* layers which will be in charge of avoiding obstacles and moving the robot randomly, respectively.

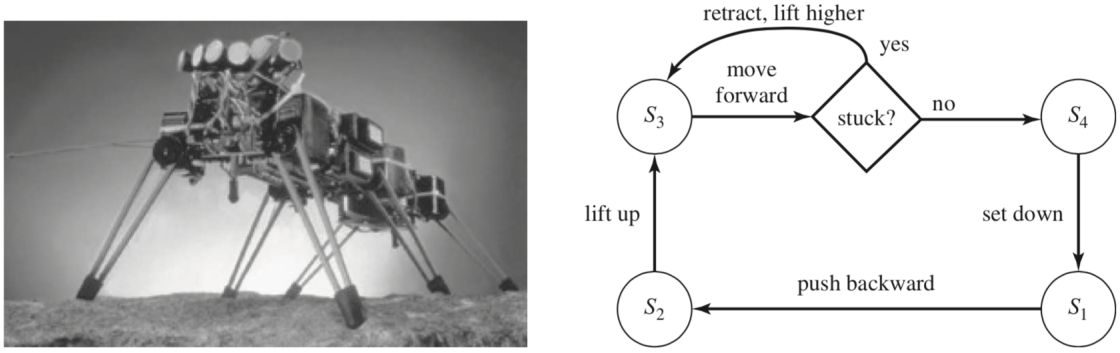


Figure 1.2: A hexapod robot called Genghis (left) was created by Rodney Brooks [2] using 57 augmented finite states machines (AFSM). An AFSM for controlling only one leg is illustrated on the right side. The AFSM receives feedback of a proximity sensor in order to see if the leg is stuck or not and to lift it higher next time.

Source: Russell et al. [4] *Artificial Intelligence, A Modern Approach.*

There is no central unit that orchestrates the network and there is no hierarchical arrangement [9].

1.6 Machine Learning in Robotics

Machine learning is a research field that involves statistics, artificial intelligence and computer science. It provides the necessary tools to derive knowledge and make predictions given unstructured or structured data [10].

According to Brooks [9], a robot can be seen as a system that interacts with the world, and uses learning as a method of improving its performance.

These are some of the aspects that are present in physically instantiated autonomous robots:

- They may receive hundreds (nowadays thousands) of input bits.

- These input bits cannot be directly mapped to a robot state.
- There are many actions that the robots may execute at any given time step.
- There is certain knowledge about the domain already available.

The available knowledge can be used directly by the robot instead of learning from scratch such as the design features regarding its engineering.

The role of machine learning in robotics should overcome the following challenges regarding a physically instantiated autonomous robot that possesses the aforementioned aspects:

- It should learn *representations* of the world that help to achieve whatever purpose the robot is programmed to accomplish.
- It should learn aspects concerning sensors and actuators that can help them to be more precise. This process is called *calibration*.
- A robot can behave differently regarding the perceived environment and its internal state. The *interaction* between these behaviors should be learned.
- A robot should be able to *create* new behaviors.

1.7 Uncertainty in Robotics

Robots need to deal with uncertainty all the time because the environments they are placed in, are themselves unpredictable. Likewise, their sensors and actuators are subject to noise affecting perception and action, respectively. For instance, if a robot is told to go two meters ahead, it can end up some centimeters beyond the goal position because it could have slipped due to the environment's ground material, its sensors could have reported that the two meters were already covered or its actuators could have executed more wheel turns than they should. This makes it so that the robot position cannot be tracked precisely after a while. Here is where *probabilistic robotics* techniques enter into action to deal with uncertainty.

Probabilistic robotics uses probability theory to deal with uncertainty in robotics. For instance, in global localization, the robot's internal knowledge about the state of the environment is represented by a probability density function (pdf)[1] over the space of locations [11].

An example of global localization can be found in [11] and [12]. The robot is equipped with a sensor that detects the presence of doors, it is placed somewhere in the environment, and its task is to localize itself. The robot estimate, also called

[1]This probability concept is further defined, see section 2.1.6.

belief ($bel(x)$), is represented as the probability density function over the space of locations. In figure 1.3.a the robot has total uncertainty and thus its belief is represented as a uniform distribution. When the robot senses the first door (figure 1.3.b), this belief puts high probability in all the places where there are doors, at this point the robot knows it is in front of a door but it does not know yet which of the three it is. Moving the robot forward makes its belief shift in the same direction (figure 1.3.c). Finally, the robot senses another door and the belief puts most of the probability on a location near a door, the robot now is confident about where it is in the environment (figure 1.3.d). This example illustrates some important properties of the *Bayes Filter* explained in more detail in section 2.4.4.

Bayes filter is considered the base of many probabilistic algorithms applied to robot localization. For instance, *particle filter* uses Bayes filter as a main component. It approximates the robot location using a set of particles where each particle has its own estimation with an associated value representing how good the particle's estimation is. More details are provided about particle filter later on section 2.4.4.

Most of the particle filter techniques that deal with robot localization in partially observable environments are complex to implement. The less-complex implementations deal with fully observable environments (for instance, robots with non-realistic sensors that measure the Euclidean distance to the closest landmark in the environment). Additionally, there is not a concrete "classical" particle filter implementation[2] that performs better than the rest in all the cases. Some particle filter implementations perform well using some specific sensors, some other perform better in different type of environments, etc. Therefore, instead of orienting the work to see if machine learning helps particle filter to perform better than all the classical approaches, the motivation will seek to answer to the question: *can machine learning help to decrease the complexity of particle filter implementation for robot localization while dealing with partially observable environments and a real set of sensors?*.

1.8 Simulation in Robotics

According to Ẑlajpah [13], simulation is the process of designing a model of an actual or theoretical physical system, executing the model, and examining the execution output.

Simulation has been widely used for autonomous robots in recent years (see [14]). It can help researchers to investigate, visualize, and test scenarios [13] with modeled robots in virtual environments.

There are many simulation tools available for robotics; however, fewer meet the requirements to be used in academia. These requirements are mentioned by Hughes et al. [15] such as being:

[2]The term *classical* here refers to the nonuse of machine learning.

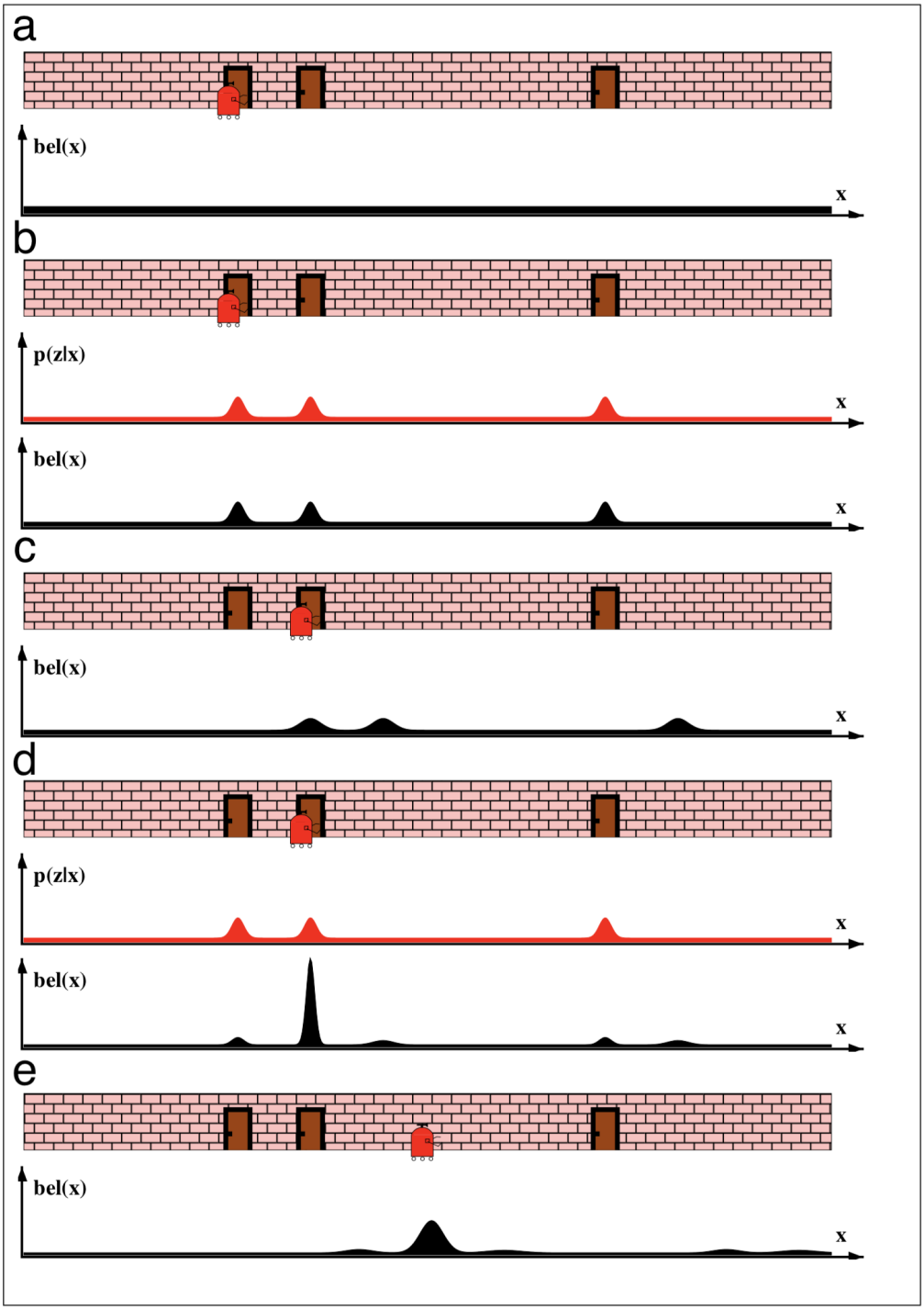


Figure 1.3: A robot is put somewhere in the environment with total uncertainty about its initial pose. Using a sensor that notices the presence of doors, and a map of the environment, it uses probabilistic robotics to disambiguate its pose and to be certain about where it is.

Source: Extracted from Thrun et al. [6] *Probabilistic Robotics*

- Open source and therefore freely accessible for students.
- Easy to install and intuitive to use.

- Comprehensive for all student levels from beginners (fundamentals) to advanced students.

- A bridge that allows students to pass from education to research.

Webots and Gazebo are two robotics simulation tools that meet the aforementioned requirements. Hughes et al. [15] compare both and their main results are here mentioned.

Webots[3] is an open source simulator that is used for simulation, research and industry. It allows the transfer of the robot controller from a simulated to a real robot. It can be installed in Linux, Windows and MacOS. The tool comes with a guide through beginner, novice, intermediate, and advanced level in robotics called the WikiBook Cyberbotics' Robot Curriculum[4]. Different interfaces can be used depending on the students' age and knowledge. For the younger ones, *Blockly*[5] provides an easy-to-understand and intuitive interface to program the robots. *BotStudio*, a tool to graphically program a robot using a finite automaton, is also used at a beginner, novice, and intermediate level. For advanced level students, instead, an interface is available to program the controller directly using C++, Python, Java, Matlab, etc.

Gazebo[6] is a robotic simulation tool that works on top of the Robot Operating System (ROS) which allows use of state-of-the-art ROS modules directly on simulated and real robots. It is maintained by the Open Source Robotics Foundation[7]. In consequence, a variety of robots and environments are ready to be used. Scratch[8], a graphical-interface programming language used to introduce young people to programming, can be used as an interface to communicate with ROS[9] for beginner to intermediate level students.

Some simulators have been popularized lately among researchers to deal with visual robot localization exclusively such as *House3D* [16], a virtual 3D environment containing 45000 indoor 3D scenes with any kind of 3D labeled objects and *AI Habitat* [17], a simulation platform for embodied AI agents training in a highly photorealistic and efficient 3D simulator.

The current work will make use of Webots as a robotics simulator tool given the aforementioned benefits compared to other robotic simulators.

[3] `https://www.cyberbotics.com`
[4] `https://en.wikibooks.org/wiki/Cyberbotics\OT1\textquoteright_Robot_Curriculum`
[5] `https://developers.google.com/blockly/`
[6] `http://gazebosim.org`
[7] `https://www.openrobotics.org/`
[8] `https://scratch.mit.edu`
[9] `http://wiki.ros.org/scratch`

	Gazebo	Webots
Physics engine	ODE, Bullet, DART, and Somebody	Open Dynamics Engine
Realism	Advanced 3D Graphics	3D Graphics
Sensor modelling	Sensors and noise	Some sensors, lower controllability
Operative system	Linux	Linux, Windows, MacOS
Open-Source	Open Source	Free, Open Source
Existing Education Applications	Limited	Extensive, many platforms
Long Term Support	Well-established Online Support	Some Support & many examples

Table 1.1: Comparison between Gazebo and Webots.

Source: Adapted from Hughes et al. [15] *Good Experimental Methodologies and Simulation in Autonomous Mobile Robotics*

1.8.1 Advantages and Disadvantages

The advantages that present the use of simulation in robotics according to Amigoni et al. [18] are as follows:

- *Comparison*: simulation makes possible the comparison of the performance of a variety of algorithms that deal with the same problem in a precise manner. For instance, two robot localization algorithms can be fairly compared using the same simulated environment and configuration. This is much harder in real environments since getting exactly the same setting is almost impossible due to systematic and non-systematic errors.
- *Reproducibility and repeatability*: reproducibility is the possibility to verify the outcome of an experiment and repeatability is to obtain the same outputs when the experiment is repeated more than one time under the same circumstances. Therefore, an experiment is successful when the goal of the experimenter is achieved and the experiment can be reproduced and repeated.
- *Justification and explanation*: experiments should be made under different settings (different initial position of the robot, different environments, different parameter configurations, etc.) in order to derive well-justified conclusions. These settings can be easily changed in simulation providing robust results that can be later verified in real robots.

"All models are wrong" is a phrase mentioned by George Box [19] in 1976 referring to the fact that models are a simplified representation of the complex reality. In consequence, there is a gap between simulation and reality that may affect the outcome of the experiments. Thus, a major disadvantage of using simulation in robotics is the fact that what works well in a simulated environment may not perform as expected in reality.

1.9 Master Thesis Contributions

The main thesis contributions of this project are cited below:

- An architecture for the robot positioning problem is presented. It illustrates how machine learning can be used together with probabilistic algorithms for robot localization while taking advantage of the use of simulation. The architecture is composed of 5 components interacting with each other. A first component that shows how to create a virtual world in simulation, a second component that is in charge of collecting synthetic data in simulation using a robot controller that implements a subsumption architecture with reactive control, a third component for training the prediction models using the collected synthetic data, a fourth component focused on localizing the robot using the pretrained models together with an adapted version of the particle filter, and finally, a visualization component that illustrates the prediction accuracy.
- A variant of the particle filter algorithm is introduced. It uses machine learning models to approximate the belief to passively localize a single robot under local and global uncertainty in a partially observable, deterministic, episodic, static, continuous, and known environment. The algorithm is explained, and analyzed in detail. Experiments are made to show its efficacy.
- An open-source plugin for Webots is created. It illustrates the particle filter algorithm and allows modification of the algorithm parameters in real-time while simulating to see how these changes may affect the algorithm accuracy.

Notations

General

$t \in \mathbb{N}$	Time
$D \in \mathbb{N}$	Space dimension
$P \in \mathbb{N}$	Number of variables: univariate, multivariate
$S \in \mathbb{N}$	Number of sensors
$M \in \mathbb{N}$	Number of Particles used in the particle filter
$NN_s : \mathbb{R}^P \to \mathbb{R}$	Neural Network model for sensor number s
$a \approx b$	a is approximately equal to b
$a \sim b$	a is similar to b
$a \propto b$	a is proportional to b

Odometry

p	Initial robot position
p'	Current robot position
$[x, y, \theta]^T$	Transpose of the initial position matrix
$[\Delta x, \Delta y, \Delta\theta]^T$	Transpose of the increment in translation and orientation matrix
D	Wheel diameter
R	Encoder resolution
n	Gear ratio
Δs_l, Δs_r	Incremental travel distance for the left and right wheels
N_l, N_r	Pulse increments for the left and right wheel encoders
ΔS	Incremental linear displacement of the robot's centerpoint
$\Delta\theta$	Increment change in orientation
b	Wheelbase
E_D	Error due to different wheel diameters
D_l, D_r	Left and right wheel diameter
E_b	Wheelbase error
cw	Clockwise
ccw	Counterclockwise
$(x_{c.g.,cw}, y_{c.g.,cw},)$	Center of gravity of the clockwise experiment
$(x_{c.g.,ccw}, y_{c.g.,ccw},)$	Center of gravity of the counterclockwise experiment
$r_{c.g.,cw}, r_{c.g.,ccw}$	Absolute offset of the center of gravity with respect to the initial position
E_{syst}	Odometric accuracy for systematic errors

Probability

Ω	Sample Space
$\mathcal{E}$	Event
$Prob(\mathcal{E})$	Probability that event $\mathcal{E}$ occurs
$Prob(\mathcal{E}_1 \vert \mathcal{E}_2)$	Conditional probability of event $\mathcal{E}_1$ given $\mathcal{E}_2$
$p(x)$	Probability distribution of continuous random variable x. Also noted as $p_{\mathbf{X}}(x)$
$p(x\vert y)$	Conditional probability distribution of continuous random variable x given y
μ	Mean
σ	Standard deviation
$\mathcal{N}(\mu, \sigma)$	Normal distribution with mean μ and standard deviation σ
$\mathcal{U}(a, b)$	Uniform distribution with boundaries a and b
$a \sim \mathcal{N}(\mu, \sigma)$	a is drawn from a normal distribution $\mathcal{N}$
$\mathbf{X}, \mathbf{Z}$	Random variables
$p(z\vert x)$	Conditional probability distribution of $\mathbf{Z} = z$ given $\mathbf{X} = x$

Machine Learning

n	Number of features
N	Number of samples
D_N	Data set composed of N samples
$X \in \mathcal{R}^{n \times N}$	Input data
$Y \in \mathcal{R}^N$	Output desired data
$\hat{Y} \in \mathcal{R}^N$	Output predicted data
$f : \mathcal{R}^{n \times N} \to \mathcal{R}^N$	Unknown function
$\hat{f} : \mathcal{R}^{n \times N} \to \mathcal{R}^N$	Model or hypothesis
a_i	Activation value of unit i
w_{ij}	Weight from unit i to unit j
$g : \mathcal{R} \to \mathcal{R}$	Activation function
L	Loss
W	Neural network weights

Probabilistic Robotics

$x_t \in \mathbb{R}^{D \times P}$	Matrix representing state x at time t
$z_t \in \mathbb{R}^S$	Vector of measurement z at time t
$u_t \in \mathbb{R}^D$	Action u at time t
$\hat{z} : \mathbb{R}^{D \times P} \to \mathbb{R}^S$	Function that predicts sensor measurements given a state
$\hat{x}_t \in \mathbb{R}^{D \times P}$	Matrix representing predicted state $\hat{x}$ at time t
$\hat{x}_t^* \in \mathbb{R}^{D \times P}$	Matrix representing best-predicted state $\hat{x}$ at time t
$\theta_t \in \mathbb{R}$	Angle of orientation at time t
$x_{1:t}$	Sequence containing $\{x_1, x_2, ..., x_t\}$
$x_t^{[m]}$	State x at time t of particle m
$bel(x_t)$	Belief of state x at time t
$\overline{bel}(x_t)$	Prediction belief of state x at time t
σ_{xy}	Standard deviation for the translation
σ_θ	Standard deviation for the rotation
w_t^m	Weight of the particle m at time t

Abbreviations

AFSM	augmented finite state machines
ROS	Robot Operating System
EPFL	Swiss Federal Institute of Technology in Lausanne
ROS	Robot Operative System
3D	Three-Dimensional
DOF	Degrees Of Freedom
pdf	Probability Density Function
MAP	Maximum a Posteriori probability
KF	Kalman Filter
PF	Particle Filter
EKF	Extended Kalman Filter
MCKF	Maximum Correntropy Kalman Filter
MMSE	Minimum Mean Square Error
MSE	Mean Square Error
MAE	Mean Average Error
MCC	Maximum Correntropy Criterion
IEKF	Invariant Extended Kalman Filter
PCC	Pearson Correlation Coefficient
PPF	Pearson Particle Filter
GPU	Graphical Processing Unit
GPGPU	General Purpose Graphical Processing Unit
MCL	Monte Carlo localization
SLAM	Simultaneous Localization And Mapping Problem
DP	Differentiable Programming
DPFs	Differentiable Particle Filters
PF-net	Particle Filter Network
HF	Histogram Filter
DMN	Differentiable Mapping Network
DPFRL	Discriminative Particle Filter Reinforcement Learning
DVRL	Deep Variational Reinforcement Learning
GRU	Gated Recurrent Unit
RNN	Recurrent Neural Networks
PF-RNN	Particle Filter Recurrent Neural Networks
CTE	Cumulative Translation Error
CRE	Cumulative Rotation Error
RMSD	Root Mean Square Deviation
RMSDT	Root Mean Square Deviation for Translation
RMSDR	Root Mean Square Deviation for Rotation

Part I

State of the Art

Chapter 2

Basic Notions

2.1 Probability

This section presents the basic notions in probability theory that are considered useful for the general understanding of the current work.

2.1.1 Experiments, Sample Spaces, Events, and the Probability Law

Bertsekas et al. [20] define an *experiment* (also called *random experiment* [21]) as an elemental process involved in every probabilistic model. It produces one or multiple outcomes that are not deterministic. The set of all possible outcomes is called *sample space* of the experiment, denoted as Ω. An *event* $\mathcal{E}$ is a subset of the sample space, it could also be the entire sample space, its complement, or the empty set.

The *probability law* associates to every event $\mathcal{E}$ a number $Prob(\mathcal{E})$, called the probability of $\mathcal{E}$. It satisfies the following three axioms: *nonnegativity, additivity,* and *normalization.*

- **Nonnegativity**: $Prob(\mathcal{E}) > 0$, for every event $\mathcal{E}$.
- **Additivity**: if $\mathcal{E}_1$ and $\mathcal{E}_2$ are two disjoint events, then the probability of their union satisfies: $Prob(\mathcal{E}_1 \cup \mathcal{E}_2) = Prob(\mathcal{E}_1) \cup Prob(\mathcal{E}_2)$.
- **Normalization**: $Prob(\Omega) = 1$ where Ω is the entire sample space.

Thus, an experiment is defined by the tuple $(\Omega, \{\mathcal{E}\}, Prob(.))$.

2.1.2 Conditional Probability and Independence

The *conditional probability* of an event $\mathcal{E}_1$ given event $\mathcal{E}_2$, denoted as $Prob(\mathcal{E}_1|\mathcal{E}_2)$ is defined by equation 2.1.

$$Prob(\mathcal{E}_1|\mathcal{E}_2) = \frac{Prob(\mathcal{E}_1 \cap \mathcal{E}_2)}{Prob(\mathcal{E}_2)}, Prob(\mathcal{E}_2) \neq 0 \tag{2.1}$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are two events in the same sample space. In more general terms, it is the probability that some event happens as the outcome of an experiment based on partial information.

For a fixed event $\mathcal{E}_3$, it can be shown that conditional probabilities specify a probability law that satisfies the aforementioned axioms.

- Nonnegativity and normalization:

$$Prob(\Omega|\mathcal{E}_3) = \frac{Prob(\Omega \cap \mathcal{E}_3)}{Prob(\mathcal{E}_3)} = \frac{Prob(\mathcal{E}_3)}{Prob(\mathcal{E}_3)} = 1$$

- Additivity for two disjoint events $\mathcal{E}_1$ and $\mathcal{E}_2$:

$$\begin{aligned} Prob(\mathcal{E}_1 \cup \mathcal{E}_2|\mathcal{E}_3) &= \frac{Prob((\mathcal{E}_1 \cup \mathcal{E}_2) \cap \mathcal{E}_3)}{Prob(\mathcal{E}_3)} \\ &= \frac{Prob((\mathcal{E}_1 \cap \mathcal{E}_3) \cup (\mathcal{E}_2 \cap \mathcal{E}_3))}{Prob(\mathcal{E}_3)} \\ &= \frac{Prob(\mathcal{E}_1 \cap \mathcal{E}_3) + Prob(\mathcal{E}_2 \cap \mathcal{E}_3)}{Prob(\mathcal{E}_3)} \\ &= \frac{Prob(\mathcal{E}_1 \cap \mathcal{E}_3)}{Prob(\mathcal{E}_3)} + \frac{Prob(\mathcal{E}_2 \cap \mathcal{E}_3)}{Prob(\mathcal{E}_3)} \\ &= Prob(\mathcal{E}_1|\mathcal{E}_3) + Prob(\mathcal{E}_2|\mathcal{E}_3) \end{aligned}$$

$\mathcal{E}_1$ is *independent* of $\mathcal{E}_2$ when the occurrence of event $\mathcal{E}_2$ provides no information about the occurrence of event $\mathcal{E}_1$. It is defined by equation 2.2.

$$Prob(\mathcal{E}_1 \cap \mathcal{E}_2) = Prob(\mathcal{E}_1)Prob(\mathcal{E}_2) \tag{2.2}$$

$Prob(\mathcal{E}_1 \cap \mathcal{E}_2)$ is also called the *joint probability* [22] denoted as $Prob(\mathcal{E}_1, \mathcal{E}_2)$. In addition, if $Prob(\mathcal{E}_2) > 0$, then equation 2.3 holds.

$$\begin{aligned} Prob(\mathcal{E}_1|\mathcal{E}_2) &= \frac{Prob(\mathcal{E}_1 \cap \mathcal{E}_2)}{Prob(\mathcal{E}_2)} \\ &= \frac{Prob(\mathcal{E}_1)Prob(\mathcal{E}_2)}{Prob(\mathcal{E}_2)} \\ &= Prob(\mathcal{E}_1) \end{aligned} \tag{2.3}$$

2.1.3 Conditional Independence

Two events, $\mathcal{E}_1$ and $\mathcal{E}_2$ are *conditional independent* given event $\mathcal{E}_3$ for $Prob(\mathcal{E}_3) > 0$ if equation 2.4 holds.

$$Prob(\mathcal{E}_1 \cap \mathcal{E}_2|\mathcal{E}_3) = Prob(\mathcal{E}_1|\mathcal{E}_3)Prob(\mathcal{E}_2|\mathcal{E}_3) \tag{2.4}$$

The notion of independence does not imply conditional independence, and vice versa [20].

2.1.4 Bayes' rule

Bayes' rule was defined by Thomas Bayes in the 1700s [21]. From the definition of conditional probability (see equation 2.1), the following relation can be obtained:

$$Prob(\mathcal{E}_1 \cap \mathcal{E}_2) = Prob(\mathcal{E}_1|\mathcal{E}_2)Prob(\mathcal{E}_2) = Prob(\mathcal{E}_2 \cap \mathcal{E}_1) = Prob(\mathcal{E}_2|\mathcal{E}_1)Prob(\mathcal{E}_1) \tag{2.5}$$

From the second and last term of the aforementioned equation, the following equation can be obtained:

$$Prob(\mathcal{E}_1|\mathcal{E}_2) = \frac{Prob(\mathcal{E}_2|\mathcal{E}_1)Prob(\mathcal{E}_1)}{Prob(\mathcal{E}_2)}, Prob(\mathcal{E}_2) > 0 \tag{2.6}$$

This equation is useful to solve $Prob(\mathcal{E}_1|\mathcal{E}_2)$ in terms of $Prob(\mathcal{E}_2|\mathcal{E}_1)$.

2.1.5 Random Variables

A *random variable* is a function that associates a real number to each outcome in the sample space of an experiment. Bold uppercase letters are used to represent a random variable such as $\mathbf{X}, \mathbf{Z}$. Additionally, a random variable can be conditioned on an event or another random variable [20].

A *discrete random variable* is a random variable that can take a finite or countable infinite number of values [22].

A *continuous random variable* is a random variable that can take as a value any of the infinite values within a range of real numbers [21].

2.1.6 Probability Density Function

The *probability distribution* of a random variable $\mathbf{X}$ is a description of the probabilities associated with the possible values of $\mathbf{X}$ [21].

A *probability density function* (pdf) is a nonnegative function $p(x)$ that describes the probability distribution of a continuous random variable such that for any subset B of the real line:

$$Prob(\mathbf{X} \in B) = \int_B p(x)dx \tag{2.7}$$

Therefore, the probability of a continuous random variable is defined within an interval of values.

$$Prob(a \leq x \leq b) = \int_a^b p(x)dx, \quad \int_{-\infty}^{\infty} p(x)dx = 1 \tag{2.8}$$

Normal Distribution

Also called *Gaussian distribution*. Its probability density function is defined as follows:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Where σ and μ are the standard deviation and mean of the distribution respectively. It is noted as $\mathcal{N}(\mu, \sigma^2)$.

Uniform Distribution

Its probability density function is defined as follows:

$$p(x) = \begin{cases} \frac{1}{b-a} & for\, a \leq x \leq b \\ 0 & for\, x < a \ or \ x > b \end{cases}$$

where a and b are two boundary numbers. It is noted as $\mathcal{U}(a, b)$.

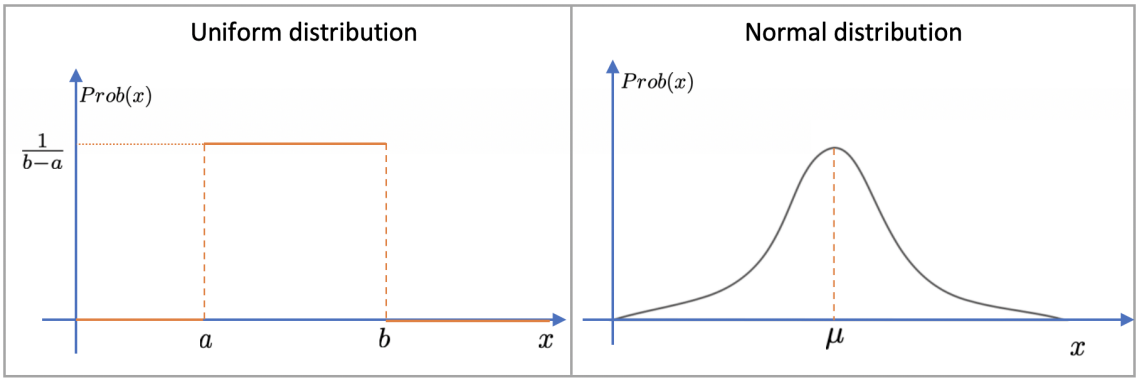


Figure 2.1: Uniform vs. Normal distribution.

2.1.7 Expectation, Variance, and Standard Deviation

The *expected value, expectation* or *mean* of a random variable $\mathbf{X}$ with probability density $p_{\mathbf{X}}$ is defined by equation 2.9.

$$E[\mathbf{X}] = \int x\, p_{\mathbf{X}}(x)\, dx \tag{2.9}$$

The expected value summarizes the probabilities of all the possible values of $\mathbf{X}$ in a single representative number [20].

The *variance* of a random variable $\mathbf{X}$ is a nonnegative quantity defined by equation 2.10.

$$Var[\mathbf{X}] = E[(\mathbf{X} - E[\mathbf{X}])^2] \tag{2.10}$$

It provides a measure of dispersion of $\mathbf{X}$ around its mean [20].

The *standard deviation* of $\mathbf{X}$ is another measure of dispersion. It is easier to interpret because it is in the same units of $\mathbf{X}$ and it is defined by equation 2.11.

$$\sigma_{\mathbf{X}} = \sqrt{Var[\mathbf{X}]} \tag{2.11}$$

2.1.8 Bayesian Statistical Inference

In the Bayesian view the unknown parameters are random variables with known distributions whereas in the classical view the unknown parameters are deterministic quantities [20].

The *unknown parameter* is denoted by X which is modeled as a single or as a

sequence of random variables denoted as $\mathbf{X}$. The information about $\mathbf{X}$ can be obtained given a *collection of observations* $\mathbf{Z} = \{\mathbf{Z_1}, ..., \mathbf{Z_n}\}$ (being z any possible value of $\mathbf{Z}$), and assuming the following information is known:

- $\mathbf{Z}$ has a probability distribution that is a function of x denoted as $p(z|x)$ implying that the exact form of the distribution of $\mathbf{Z}$ is conditional on the value assigned to x. Thus the joint distribution of the collection of observations is $p(z_1, ..., z_n|x)$.

- The *prior distribution* for x is known. It refers to additional information about x that can be summarized in the form of a probability distribution for x, denoted as $p(x)$ with mean μ and variance σ^2. The probabilities related to the prior distribution shows the analyst's belief regarding the true value of x before observing $\mathbf{Z}$ [21].

Once some particular values $\{z_1, ..., z_n\}$ of $\mathbf{Z} = \{\mathbf{Z_1}, ..., \mathbf{Z_n}\}$ have been observed, the prior distribution $p(x)$ and the joint distribution $p(z_1, ..., z_n|x)$ are used to find the *posterior distribution* for x, defined as $p(x|z_1, ..., z_n)$. The posterior distribution represents the analyst's degree of belief concerning the true value of x after observing $\mathbf{Z}$.

Therefore, the posterior distribution is calculated using Bayes' rule as shown in equation 2.12.

$$p(x|z_1, ..., z_n) = \frac{p(z_1, ..., z_n|x)p(x)}{p(z_1, ..., z_n)} \tag{2.12}$$

Uninformative Prior

A *uninformative prior* is a form of prior distribution intended to have as little influence on the posterior as possible [23] in order to present a more objective analysis and thus the summarized conclusions by the posterior density are based on an initially unprejudiced probability distribution [24]. According to Bishop [25] is also referred to as *"letting the data speak for themselves"*.

2.1.9 Point Estimation

A *point estimate* is a single numerical value that represents the best guess for X, denoted as $\hat{x}$.

The value of $\hat{x}$ can be obtained applying a function g to the observed value z. This

is called the *estimate*.

$$\hat{x} = g(z) \tag{2.13}$$

The random variable $\hat{\mathbf{X}} = g(\mathbf{Z})$ is called an *estimator*. $\hat{\mathbf{X}}$ is a random variable because the outcome of the estimation depends on the random value of the observation [20].

An estimator $\hat{\mathbf{X}}$ is *unbiased* if and only if $E[\hat{\mathbf{X}}] = X$; otherwise it is *biased* with bias $Bias[\hat{\mathbf{X}}] = E[\hat{\mathbf{X}}] - X$. The variance of an estimator $\hat{\mathbf{X}}$ is defined by equation 2.14.

$$Var[\hat{\mathbf{X}}] = E[(\hat{\mathbf{X}} - E[\hat{\mathbf{X}}])^2] \tag{2.14}$$

2.1.10 Combining two estimators

According to Bontempi [22], given two unbiased estimators $\hat{\mathbf{X}}_\mathbf{1}$ and $\hat{\mathbf{X}}_\mathbf{2}$ of the parameter X.

$$E[\hat{\mathbf{X}}_\mathbf{1}] = E[\hat{\mathbf{X}}_\mathbf{2}] = X$$

Both estimators have equal non-zero variance and they are uncorrelated.

$$Var[\hat{\mathbf{X}}_\mathbf{1}] = Var[\hat{\mathbf{X}}_\mathbf{2}] = v$$

Let $\hat{\mathbf{X}}$ be the combined estimator such that:

$$\hat{\mathbf{X}} = \frac{\hat{\mathbf{X}}_\mathbf{1} + \hat{\mathbf{X}}_\mathbf{2}}{2}$$

The estimator $\hat{\mathbf{X}}$ will be unbiased and with smaller variance:

$$E[\hat{\mathbf{X}}] = \frac{E[\hat{\mathbf{X}}_\mathbf{1}] + E[\hat{\mathbf{X}}_\mathbf{2}]}{2} = X$$

$$Var[\hat{\mathbf{X}}] = \frac{1}{4} Var[\hat{\mathbf{X}}_\mathbf{1} + \hat{\mathbf{X}}_\mathbf{2}] = \frac{Var[\hat{\mathbf{X}}_\mathbf{1}] + Var[\hat{\mathbf{X}}_\mathbf{2}]}{4} = \frac{v}{2}$$

2.1.11 Maximum a Posteriori Probability

An example of Bayesian estimator is the Maximum a Posteriori probability (MAP). Given an observed value z of $\mathbf{Z}$, a value for x is selected that maximizes the posterior distribution $p(x|z)$ as in equation 2.15.

$$\hat{x} = \arg\max_{x} p(x|z) \tag{2.15}$$

Figure 2.2 illustrates this method.

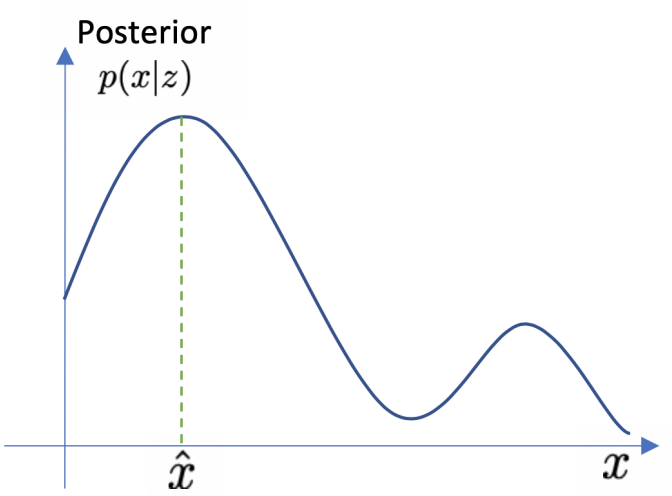


Figure 2.2: The selected estimated value $\hat{x}$ is selected such that the posterior distribution is maximized.

Source: Adapted from Bertsekas et al. [20] *Introduction to Probability*

2.1.12 Nonparametric Estimation

The *nonparametric estimation* is a set of statistical methods developed to estimate a value for which the set of parameters nor the probability distribution are fixed. They are also called *distribution-free* methods [22].

Given a collection of observations z of the random variable $\mathbf{Z}$, in order to estimate a parameter x, using the estimate $\hat{x} = g(z)$. The accuracy of the estimator $\hat{\mathbf{X}}$, in most of the cases, can be obtained using the associated distribution [22]. What if such distribution is not known? Here is when nonparametric estimation methods enters into action.

Resampling is an important component of most of the nonparmetric estimation methods. It requires a large number of repeated computations of the data. The

computational power nowadays can deal with such large computations which has led to the popularization of nonparametric estimation techniques [22].

2.1.13 Monte Carlo Simulation

Bontempi et al. [26] define the Monte Carlo (MC) simulation as a technique based on sampling the distributions of uncertain quantities. It can be seen as a black box called *the simulator* that receives the uncertain quantities as input called *parameters* and returns one or more values called *measurements* representing the analysis of such quantities. The simulator is a deterministic function.

The MC simulation consists of three steps:

- The simulator receives as input a number of pseudo-random numbers $\mathbf{Z} = \{z_1, ..., z_n\}$.
- For each number received as input $z_i \in \mathbf{Z}$, where $1 \leq i \leq n$, it calculates the output applying the deterministic function $x_i = s(z_i)$.
- The outputs $\{x_1, ..., x_n\}$ are aggregated and they can be combined to estimate the desired quantity $\mathbf{X}$.

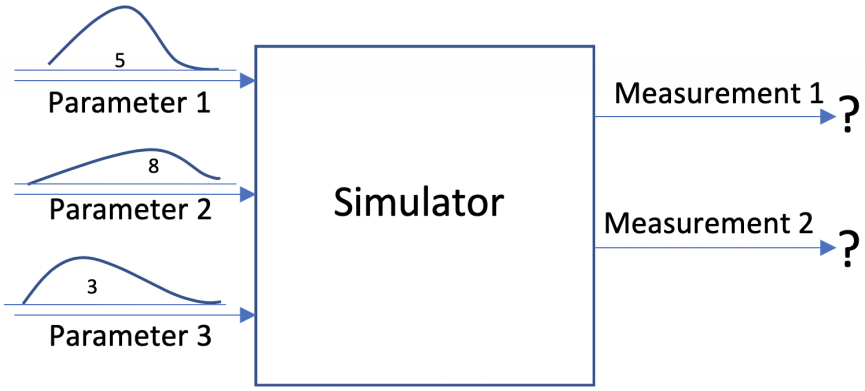


Figure 2.3: Scheme of Monte Carlo Simulation.

Source: Adapted from Bontempi et al. [26] *Cours de Modélisation et Simulation*

More formally, let $\mathbf{Z} \in \mathbb{R}^D$ and $\mathbf{X} \in \mathbb{R}$ be random variables with space dimension of D and 1, respectively. $\mathbf{Z}$ represents the input and $\mathbf{X}$ represents the output of an MC simulator. There exists a deterministic function by definition $s : \mathbb{R}^D \to \mathbb{R}$ such that $\mathbf{Z} = s(\mathbf{Y})$.

The MC simulation generates a set of independent variables $\{z_1, ..., z_n\}$ drawn from the same probability density $p_{\mathbf{Z}}(z)$, this process is called *sampling*.

Given the random variable $\mathbf{Z}$ and the function $s(.)$, some properties of the distribution of $\mathbf{X}$ can be estimated such the expected value.

$$x = E[s(\mathbf{Z})] = \int s(z)p(z)dz$$

Using the previously generated set of independent variables $\{z_1, ..., z_n\}$.

$$\hat{x} = \frac{1}{n}\sum_{i=1}^{n} s(z_i)$$

This estimator is unbiased.

$$E[\hat{\mathbf{X}}] = x$$

Its variance is defined below.

$$Var[\hat{\mathbf{X}}] = \frac{\sigma^2}{n}$$

where σ^2 is the variance of $s(\mathbf{Z})$.

The variance of $\hat{\mathbf{X}}$ represents the precision of the estimation of x and it is independent of the dimension D of the input $\mathbf{Z}$ [26]. The precision will depend on the number of samples n generated.

2.2 Machine Learning

Machine learning, predictive analytics or *statistical learning* is a research field at the intersection of statistics, artificial intelligence and computer science [27]. It is in charge of providing the necessary tools to derive knowledge and make predictions given unstructured or structured data [10].

In the early days of artificial intelligence, a system that was considered *"intelligent"* was made based on hand-coded *if* and *else* statements [27]. For instance, a spam classifier can be made using a black list of words and filtering those emails out that contain any of the words in the list. Many applications use hand-crafted rules, specially those in which humans have a good understanding on how the application works; however, this is not always the case. For example, a face recognition application can not be made out of hand-crafted rules because the way a computer perceives an image is different from the way how humans perceive it and therefore humans are not able to come up with a set of hand-coded rules that deals with the problem. This is where machine learning can be used, given a large set of face images can be enough for feeding an algorithm that will be able to recognize a face.

2.2.1 Supervised and Unsupervised Learning

The algorithms that learn from input-output pairs are called *supervised learning* [27]. For instance, determining whether an email is spam, detecting fraud in credit card transactions, and recognizing damaged components in manufactured chips.

Unsupervised learning includes those algorithms that learn using input data only. In other words, the output is not known. For instance, segmenting customers into groups, detecting whether it is a bot or a human using the web system.

The data can be thought to be a table [27] containing rows and columns. Every row, called a *sample* is a data point (e.g. an image, a transaction, a manufactured chip) and every column, called *feature* is a representation of some information associated with a data point (e.g. a pixel, the amount of the transaction, the type of manufactured chip). In the case of supervised learning, every output data point is often called *target* or *label*. The available data before the learning process is called *data set*.

More formally, according to Bontempi [22], a supervised learning problem can be described statistically using the following components:

- An input vector of n random variables $\mathbf{X} \in \mathbb{R}^n$ following an unknown probability distribution $p_{\mathbf{X}}(.)$.
- A *target* operator, transforming the inputs into outputs $\mathbf{Y}$ according to an unknown probability conditional distribution $p_{\mathbf{Y}}(y|\mathbf{X} = x)$.
- A training data set collection D_N of N input/output pairs drawn from the joint density $p_{\mathbf{X},\mathbf{Y}}(x, y)$.
- A *learning machine* returns an estimation of the target value y given a training data set D_N and a new input value x. The function used by this machine to make predictions is called *model*.

In order to understand further about what a learning machine is, an example is taken from James et al. [28] *"An Introduction to Statistical Learning"*.

Given a set of features $X = (X_1, ..., X_n)$ and the corresponding observed response Y, assume there is a relationship between X and Y which can be expressed by equation 2.16.

$$Y = f(X) + \epsilon \tag{2.16}$$

where f is a fixed but unknown function representing the systematic information that X provides about Y and ϵ is a random *error term*, independent of X with mean zero.

The job of a learning machine is to return an estimator $\hat{f}$, also called a model, for the function f such that given a set of inputs X, the estimator will return a prediction of their corresponding outputs as illustrated in equation 2.17.

$$\hat{Y} = \hat{f}(X) \tag{2.17}$$

where $\hat{Y}$ is an estimator of Y. The accuracy of $\hat{Y}$ is dependent on the *reducible error* and the *irreducible error*. The former can be minimized and it depends on the model chosen whereas the latter depends on ϵ and cannot be predicted using X because they are independent by definition and thus it cannot be reduced. In other words, $\hat{f}$ will not be as perfect as f.

Therefore, given a estimate $\hat{f}$ and a set of inputs X such that their predicted values are $\hat{Y} = \hat{f}(X)$ and assuming that $\hat{f}$ and X are fixed. Then, the reducible and irreducible errors are obtained as follows:

$$\begin{aligned} E[Y - \hat{Y}]^2 &= E[f(X) + \epsilon - \hat{f}(X)]^2 \\ &= \underbrace{[f(X) - \hat{f}(X)]^2}_{Reducible} + \underbrace{Var(\epsilon)}_{Irreducible} \end{aligned}$$

2.2.2 Fitting a Model

The selection of a model that can be suitable to represent the data set in the best way possible is called *fitting*. In other terms, the learning machine selects a function $\hat{f}$ that fits the data such that the reducible error is minimized among every reducible error produced by the other model candidates.

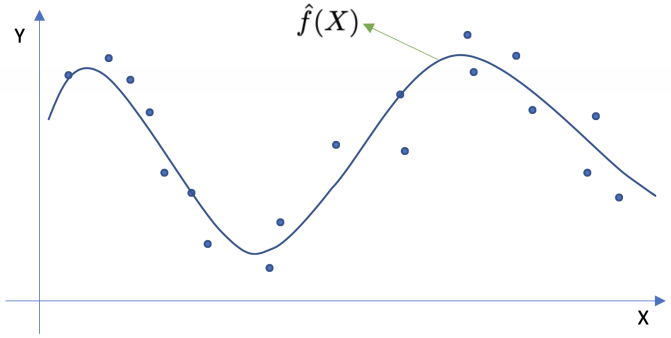


Figure 2.4: A function $\hat{Y} = \hat{f}(X)$ that fits the data.

Artificial Neural Networks

The concept of *artificial neural networks*, also called *neural networks* or *neural nets*, is inspired by networks of brain cells called *neurons*.

An artificial neural network is made of a bunch of *units* connected by *links*. A link going from unit i to unit j has a numerical value associated with it that represents the strength of the link called *weight*, denoted as w_{ij}. The link has the purpose of transmitting the *activation* value a_i from unit i to j. Each unit has a dummy input $a_0 = 1$ with weight w_{0j} called the *bias unit*.

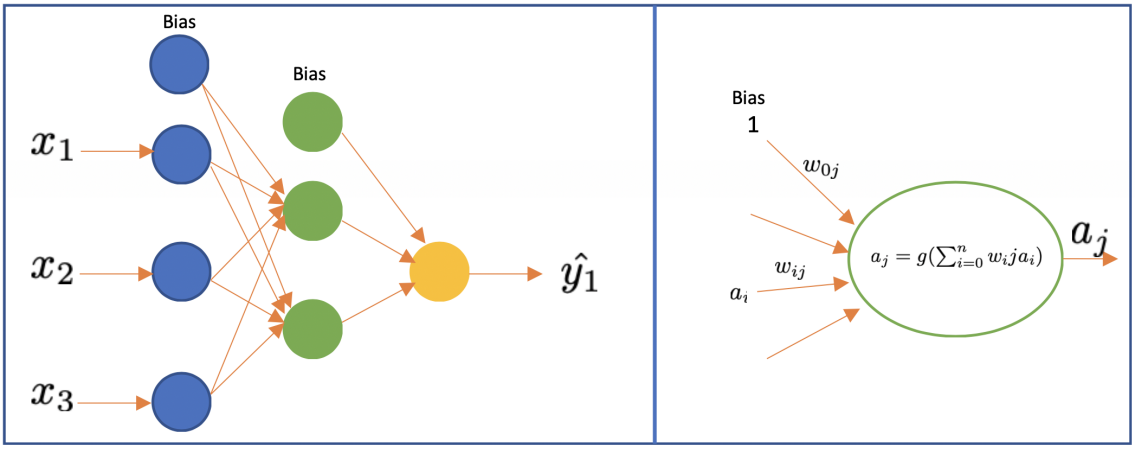


Figure 2.5: A neural network with one hidden layer in green, an input layer in blue and an output layer in yellow (left). A unit with its associated activation value (right).

Each unit j first computes the weighted sum of the activation values received as input.

$$x_j = \sum_{i=0}^{n} w_{ij} a_i$$

Then an *activation function* g is applied to obtain the activation value of unit j.

$$a_j = g(x_j)$$

The activation function determines whether the neuron will be activated or not. For instance, a *Sigmoid function* and a *Rectifier Linear Unit* (ReLU) can be used as illustrated in figure 2.6.

Now that the mathematical model for a single unit is defined, a *feed-forward network* connects the units using directed links forming a directed acyclic graph where every node represents a unit. Feed-forward networks are arranged in *layers* such that each unit receives as input the activation values of the preceding layer. There are three kind of layers:

- *Input* layer: it does not have any preceding layer, it receives as input direct values.

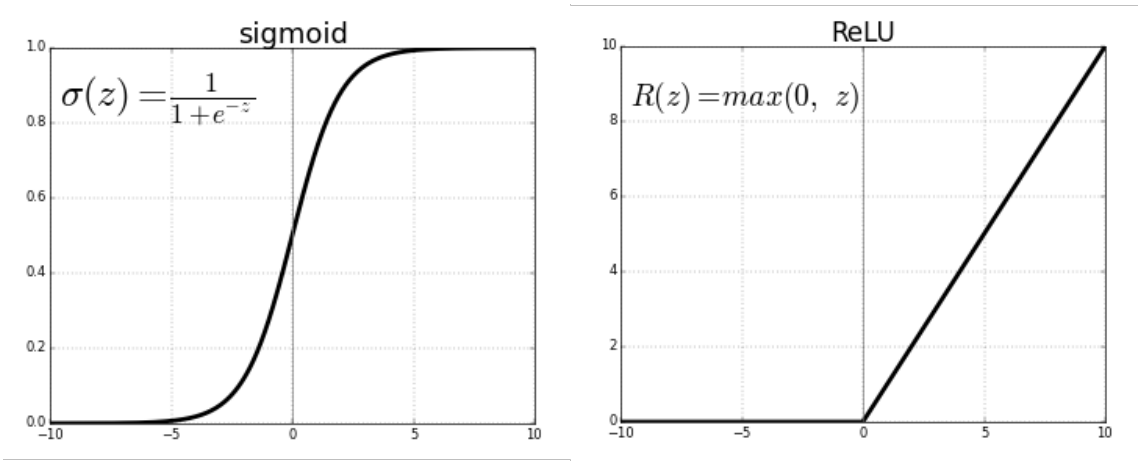


Figure 2.6: Sigmoid function (left) and ReLU function (right).

- *Hidden* layer: it is the layer that is between the input and output layers. It can connect other hidden layers.

- *Output* layer: it provides the output of the neural network.

A neural network has an input, an output and one or many hidden layers. An example illustrates how the different layers are connected in figure 2.5. A neural network can be seen as a parameterized function $\hat{f}(X, W)$ where X and W represent the input data and the weights respectively. During training process, a neural network adjusts the numerical value of the weights based on a two-step process: a *forward propagation* and a *backward propagation*.

The forward propagation step is in charge of obtaining predicted outputs $\hat{Y}$ based on inputs X, forward propagating them through the hidden layers until the output layer applying the nonlinear function of each unit. The desired output is denoted as Y.

During the back propagation step the error between the desired output and the predicted output is minimized modifying the weights. Using a loss function such that $Loss = (Y - \hat{Y})^2 = (Y - \hat{f}(X, W))^2$, it calculates the error and back propagates it from the output layer through the hidden layers until the input layer while modifying the corresponding weights. The weights are then updated using the partial derivative of the loss function with respect to the weights.

$$W \leftarrow W - (\eta \frac{\delta L}{\delta W})$$

Where η is the *learning rate* that controls how much the weights are adjusted with respect to the loss gradient.

2.2.3 Assessment of a Model

A model $\hat{f}$ is a *good fit* for f if the predicted output values $\hat{Y} = \hat{f}(X)$ are close to the desired output values Y. That is, it is necessary to quantify the range to which the predicted response value for an observation is close to the true response value for that observation [28]. A commonly-used measure is the *mean squared error* (MSE) given by equation 2.18.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{f}(x_i))^2 \tag{2.18}$$

The MSE is computed using the training data, this value is called *training MSE*. The *test MSE* is the value that is calculated with previously unseen observations not used to train the model [28]. There is no guarantee that the model that gave the lowest training MSE will also give the lowest test MSE.

Bias-Variance Trade-Off

The expected test MSE, for a previously unseen sample $\{x_0, y_0\}$ can be decomposed into the sum of the variance of $\hat{f}(x_0)$, the squared bias of $\hat{f}(x_0)$ and the variance of the error ϵ as in equation 2.19.

$$E[y_0 - \hat{f}(x_0)]^2 = Var[\hat{f}(x_0)] + (Bias[\hat{f}(x_0)])^2 + Var[\epsilon] \tag{2.19}$$

where $E[y_0 - \hat{f}(x_0)]^2$ is the *expected test MSE*. In order to minimize it, the model needs to achieve a low bias and a low variance at the same time, this is known as the *bias-variance trade-off* and it depends on the flexibility of the model. Simple models tend to have a high bias and a low variance whereas complex models usually have a high bias and a low variance. The former is said to *underfit* the data whereas the latter is said to *overfit* the data. Figure 2.7 illustrates this trade-off problem for three different data sets.

Cross-Validation

A training and a test data set should be used to estimate the training and the test error respectively. However, a test data set is not always available and thus there exist some methods to estimate the test error by holding out a subset of the observations set.

The *validation set approach* splits the available data set into two: a *training data set* and a *validation data set*. It fits a model using the training data set and then it

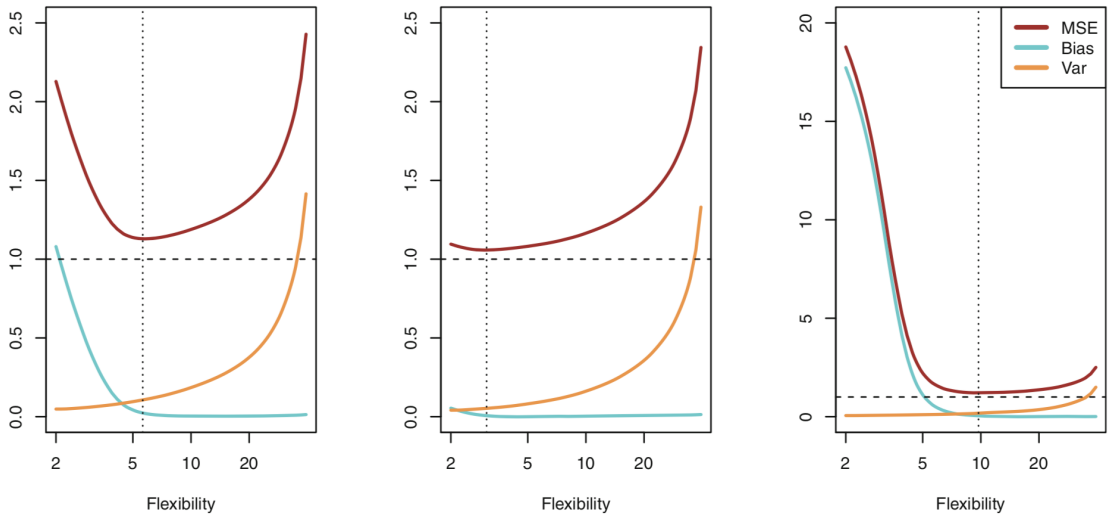


Figure 2.7: Squared bias (in blue), variance (orange), the test MSE (red), $Var[\epsilon]$ (horizontal line), and the flexibility level corresponding to the smallest test MSE (vertical line) for three data sets.

Source: Obtained from James et al. [28] *An Introduction to Statistical Learning with Applications in R.*

evaluates such model making predictions using the validation data set and comparing them with the desired outputs to obtain an estimation of the test error.

K-Fold Cross-Validation

As indicated in [28], the data set is randomly divided into k groups or folds of approximately equal size. The first fold will be used as the validation data set and the rest $k-1$ folds will be used as the training data set. The test MSE_1 is computed, after fitting the model using the training data set. The process is repeated k times, every time a different fold is selected as the validation data set. The final estimated test error is computed averaging the k test errors as in equation 2.20.

$$CV_k = \frac{1}{k}\sum_{i=1}^{k} MSE_i \tag{2.20}$$

Usually $k = 5$ or $k = 10$ is used. CV_k is an estimation of the true test MSE.

2.3 Odometry

Odometry is a way to estimate the robot positioning using the knowledge of incremental wheel motion over time. It gives good results in the short-term and it is

inexpensive; however, it presents an accumulation of errors over time.

2.3.1 Optical Encoders

In order to understand how odometry works, first the concept of an *encoder* is developed.

Borenstein et al. [8] defines an encoder to be a beam of light that points to a light sensor, called *photodetector*. The beam of light is constantly interrupted by a rotating disk, called *channel*, containing a coded opaque/transparent pattern, each called a *line*, attached to the shaft of interest. The light passes through the channel to the photodetector producing square pulses when the channel rotates. Therefore, it suffices to count how many pulses there were produced in a determined amount of time to estimate the relative position which is called *pulse increment*. One channel is not enough to decide the direction of rotation and thus two 90-degree-displaced channels are often used. The amount of lines will define the precision (also called *resolution*) of the encoder.

There are two types of optical encoders: *incremental* and *absolute*. The incremental optical encoders estimate the relative position based on the rotational velocity whereas the absolute optical encoders estimate the velocity based on the angular position.

Encoders can be integrated into the robot's wheel to measure the wheels' rotation.

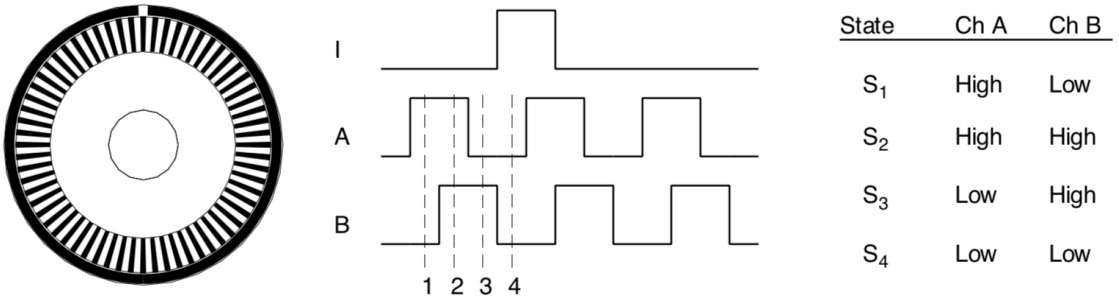


Figure 2.8: An incremental optical encoder with two channels: A and B. Both pulses can be used to determine the direction of rotation. The unique S_1, S_2, S_3, S_4 states increase the encoder resolution.

Source: Borenstein et al. [8] *Where am I?*

2.3.2 Odometry Equations

Odometry calculation depends on the type of physical mobile robot used. Here, odometry equations are defined for a differential drive mobile robot which will be used in this project.

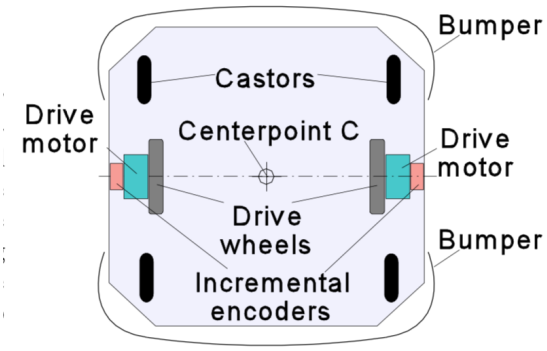


Figure 2.9: A differential drive mobile robot view from the top

Source: Borenstein et al. [8] *Where am I?*

A *differential drive* is a sub category of a mobile robot. It contains two wheels, each connected to a motor and an incremental encoder as illustrated in figure 2.9.

The initial position of the robot is defined to be $p = [x, y, \theta]^T$. Where x and y are the robot positioning in the environment and θ is the robot rotation.

The current position of the robot is $p' = p + [\Delta x, \Delta y, \Delta\theta]^T$.

The conversion factor that translates pulse increments into linear distance traveled is defined by:

$$c = \frac{\pi D}{gR}$$

where D is the wheel diameter in mm (πD is the wheel perimeter), R is the encoder resolution in pulses per revolution and g is the gear ratio of the reduction gear between the motor and the drive wheel.

The incremental travel distance is defined as follows:

$$\Delta s_{l/r} = cN_{l/r}$$

where Δs_l and Δs_r are the incremental travel distance for the left and right wheels, respectively. N_l and N_r are the pulse increments of the left and right wheel encoders.

Therefore, the incremental linear displacement of the robot's centerpoint is defined according to:

$$\Delta s = \frac{\Delta s_l + \Delta s_r}{2}$$

The robot increment change in orientation is:

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b}$$

where b is the distance between the two contact points of the wheels with the floor, called the *wheelbase*.

Next, the robot increment change in translation is:

$$\Delta x = \Delta s \cdot \cos(\theta + \Delta\theta)$$
$$\Delta y = \Delta s \cdot \sin(\theta + \Delta\theta)$$

Thus, the current robot position is defined as in equation 2.21.

$$p' = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \Delta s \cdot \cos(\theta + \frac{\Delta s_r - \Delta s_l}{b}) \\ \Delta s \cdot \sin(\theta + \frac{\Delta s_r - \Delta s_l}{b}) \\ \frac{\Delta s_r - \Delta s_l}{b} \end{bmatrix} \tag{2.21}$$

2.3.3 Systematic and Non-systematic Odometry Errors

Odometry is based on the assumption that wheel revolutions can be translated into linear displacement relative to the floor which can be the source of systematic and non-systematic errors [8]. For instance, if a differential mobile robot is placed on a surface with oil, it is probable that the robot will slip. Thus the robot position provided by odometry will differ from the true robot position as time goes on.

The errors to which this technique is subject can be categorized in two based on [8]: *systematic* and *non-systematic* errors.

- *Systematic errors*: they are constant and predictable. For instance, they happen when the wheels' diameter is not equal, the wheels are disaligned, or by definition the encoders present a finite encoder resolution which produce systematic errors.

- *Non-systematic errors*: they are unpredictable or uncertain. For instance, they happen when the floor where the robot navigates is not flat, or the wheels slip due to slippery surfaces, overacceleration, fast turning, etc.

As time goes on the uncertainty of the robot positioning increases and it can be marked as an ellipse representing the uncertainty zone where the robot is positioned as shown in figure 2.10. The ellipses increments in size as the robot moves.

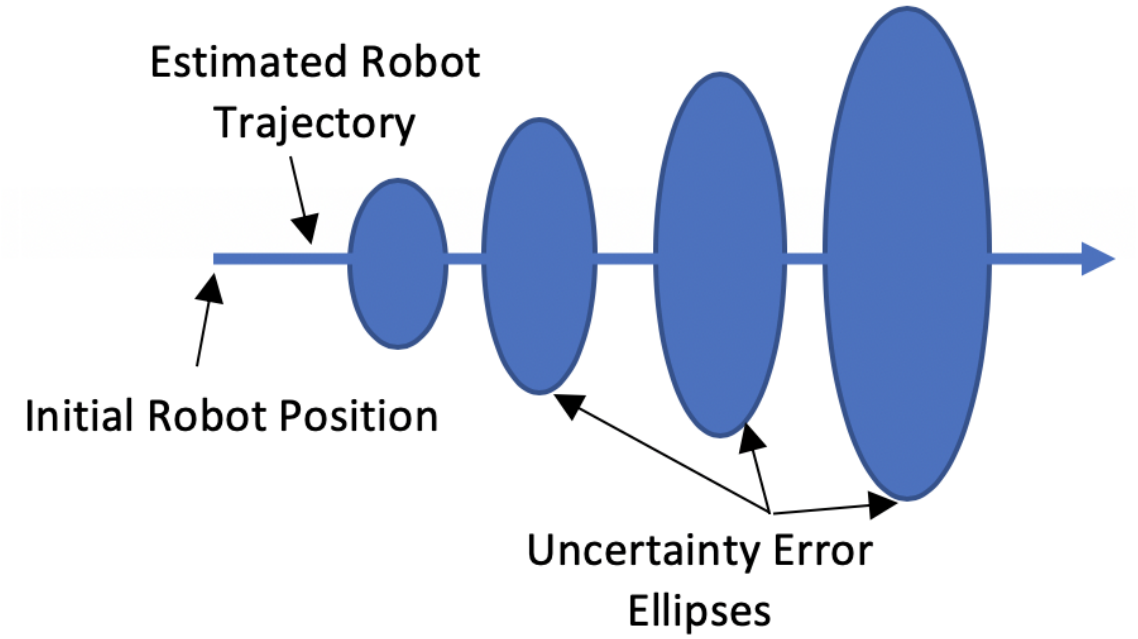


Figure 2.10: Growing ellipses indicating the growing uncertain robot positioning region.

Source: Adapted from Borenstein et al. [8] *Where am I?*

2.3.4 Measurement of Odometry Errors

Two of the predominant systematic errors can be measured using simple equations.

The error due to *different wheel diameters* is defined as:

$$E_D = 1 - \frac{D_{min}}{D_{max}}$$

where $D_{min} = min(D_l, D_r)$ and $D_{max} = max(D_l, D_r)$. D_l and D_r are the left and right wheel diameter, respectively.

The error due to uncertainty about the effective wheelbase:

$$E_b = 1 - \frac{b_{min}}{b_{max}}$$

where $b_{min} = min(b_{actual}, b_{nominal})$ and $b_{max} = max(b_{actual}, b_{nominal})$. b_{actual} is the measured wheelbase of the robot and $b_{nominal}$ is the wheelbase according to the design.

E_D and E_b will be close to 1 if the error is big; otherwise, these values will be close to 0.

An experiment to measure the systematic odometry errors consists of a robot traveling a square 4×4m path. Due to the errors, the robot will not end up at the same position it initially started in. This is called the *unidirectional square-path test* and is illustrated in figure 2.11.

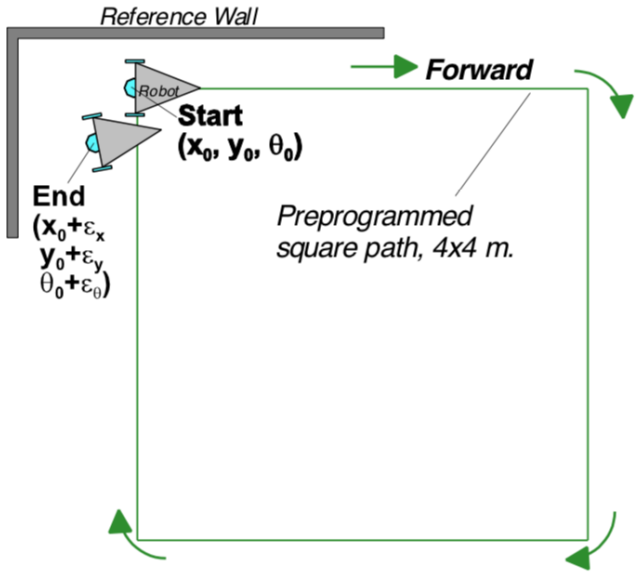


Figure 2.11: Robot traversing a square path. Systematic errors such as uncertainty about the effective wheelbase or differences in the wheel diameters cause the robot to end up at a slightly different position from where it started.

Source: Borenstein et al. [8] *Where am I?*

The initial robot pose is defined as (x_0, y_0, θ_0). After the experiment the robot pose is $(x_0 + \epsilon_x, y_0 + \epsilon_y, \theta_0 + \epsilon_\theta)$. Where $\epsilon_x, \epsilon_y, \epsilon_\theta$ are the position and orientation errors due to odometry, defined as follows:

$$\epsilon_x = x_{abs} - x_{calc} \tag{2.22}$$
$$\epsilon_y = y_{abs} - y_{calc} \tag{2.23}$$
$$\epsilon_\theta = \theta_{abs} - \theta_{calc} \tag{2.24}$$
$$\tag{2.25}$$

Where $x_{abs}, y_{abs}, \theta_{abs}$ are the absolute position and orientation of the robot and $x_{calc}, y_{calc}, \theta_{calc}$ are the calculated position and orientation by odometry.

The problem with the unidirectional square-path test is that the robot can end up at the same place it started even if there are systematic errors, they can be compensated during the trajectory. The *bidirectional square-path experiment* does not have this problem and it has demonstrated to be the correct way to measure odometric accuracy due to systematic errors [8].

The bidirectional square-path experiment, called also the UMBmark (University of Michigan Benchmark), consists on executing the unidirectional square-path test

in clockwise (cw) n times and registering the position where the robot ends up after each experiment $\{(x_{1,cw}, y_{1,cw}), \dots (x_{n,cw}, y_{n,cw}))\}$ with respect to the initial robot position (x_0, y_0). These form a cluster whose center of gravity is defined by $(x_{c.g.,cw}, y_{c.g.,cw})$, where:

$$x_{c.g.,cw} = \frac{1}{n} \sum_{i=1}^{n} x_{i,cw}$$
$$y_{c.g.,cw} = \frac{1}{n} \sum_{i=1}^{n} y_{i,cw}$$

Respectively, the experiment is then repeated in counterclockwise (ccw) n times, each time the robot ends up in a different position $\{(x_{1,ccw}, y_{1,ccw}), \dots (x_{n,ccw}, y_{n,ccw}))\}$ forming a cluster with the center of gravity $(x_{c.g.,ccw}, y_{c.g.,ccw})$, where:

$$x_{c.g.,ccw} = \frac{1}{n} \sum_{i=1}^{n} x_{i,ccw}$$
$$y_{c.g.,ccw} = \frac{1}{n} \sum_{i=1}^{n} y_{i,ccw}$$

Next, the absolute offsets of the centers of gravity with respect to the initial robot position is calculated as follows:

$$r_{c.g.,cw} = \sqrt{(x_{c.g.,cw})^2 + (y_{c.g.,ccw})^2}$$
$$r_{c.g.,ccw} = \sqrt{(x_{c.g.,ccw})^2 + (y_{c.g.,ccw})^2}$$

Finally, the measure of odometric accuracy for systematic errors is defined as follows:

$$E_{syst} = max(r_{c.g.,cw}, r_{c.g.,ccw}) \tag{2.26}$$

2.4 Probabilistic Robotics

Robotics has being evolved for years. From robotic systems designed to automatize highly repetitive and physically demanding human tasks in the early-to-mid-1940s [7], passing through researchers making the strong assumption of having exact

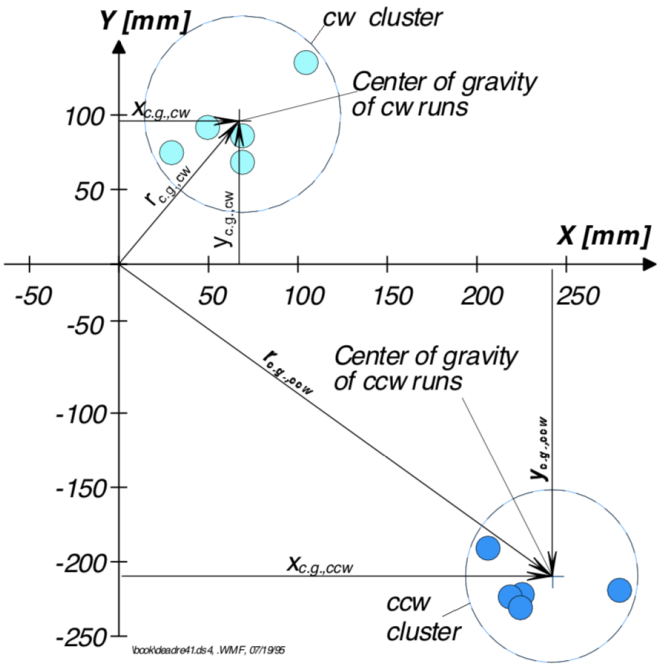


Figure 2.12: Bidirectional square-path experiment running with $n = 5$. The turquoise points indicate the end position of the robot going in clockwise sense (cw). The blue points are the end position of the robot going in counterclockwise direction (ccw). The figure shows the cw and ccw center of gravity.

Source: Borenstein et al. [8] *Where am I?*

models of robots and environments in the 1970s, to probabilistic robotics in the mid-1990s [11]. Thus a robot needs to deal with uncertainty most of the time. As an example, a robot ordered to walk 1 meter forward from its current position may not end up at the desired location due to some systematic and non-systematic errors that can be accumulated over time. In consequence, the robot might be found at a totally different position from where it was supposed to be. That is why a robot needs to be capable to deal with this uncertainty, predicting, and preserving its current position and orientation within the environment [29] regardless of the errors.

The notion of *probabilistic robotics* refers to probabilistic theory applied to robotics to deal with uncertainty. This section is oriented to explain the important and relevant concepts from probabilistic robotics applied to the current project.

2.4.1 State

The notion of *State*, according to Thrun et al. [6], refers to the set of all aspects of the robot and its environment that can influence the future. Two types of state can be defined:

- **Dynamic state** changes its position over time. For instance, people or other

robot locations.

- **Static state** does not change its position over time. For example, the walls or other moveless object locations.

Some instances of state are:

- The position of physical static entities in the environment as walls, boxes, doors, etc.
- The location and speed of mobile entities in the environment as people, other robots, etc.
- The robot pose is usually defined by its *position* (*translation*) and its orientation (*rotation*) relative to a global reference frame. A robot moves on a *fixed frame* that is attached to the ground and does not move. This frame is called the *global reference frame* (GRF). Additionally, a robot is linked within a frame that moves along with the robot. This frame is referred to as a *local reference frame* (LRF). Communication between the coordinate frames is known as the *transformation of frames* and it is a key concept when modeling and programming a robot [30]. Figure 2.14 illustrates the difference between GRF and LRF where the X_R axis points to the right side of the robot, the Y_R axis is aligned to its longitudinal axis and the Z_R axis points upward. Besides these Cartesian coordinates, the robot's angle orientation is defined by the Euler angles: Yaw, Pitch, and Roll (see [31]).

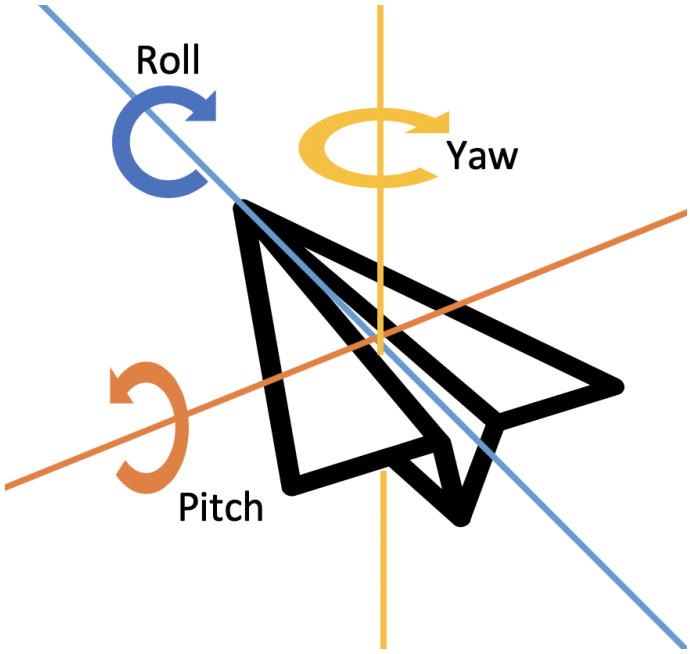


Figure 2.13: Euler angles: Yaw, Pitch, and Roll

The notation used to represent a state at time t will be denoted as x_t.

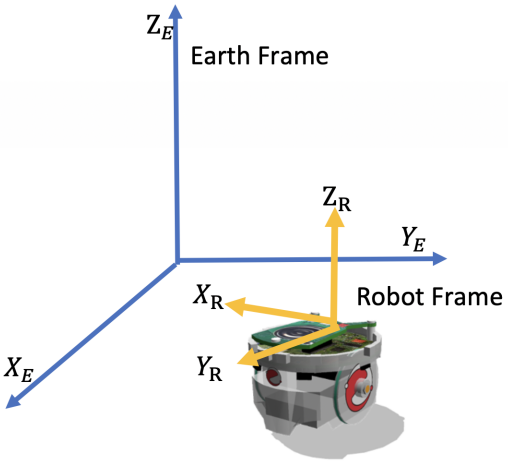


Figure 2.14: Local vs Global reference frame

The *control data* is information about the change of state in the environment [6]. It can be, for instance, the velocity or the acceleration of the robot at a given time t and thus it will be represented as u_t. The *sensor data* provided at time t is denoted as z_t and therefore a collection of percepts, called here a sequence of sensor observations, will be denoted as $z_{1:t} = z_1, z_2, ..., z_t$.

2.4.2 Belief Representation

A *belief* is a representation of the internal state of the robot about its position in the environment whereas the true state is where the robot is in the environment. In other words it is the probability that a robot at time t is at location x given previous sensor data $z_1, z_2, ..., z_t$ and previous control actions $u_1, u_2, ..., u_t$. A belief can be expressed as the conditional probability function of state x_t given sensor data $z_{1:t}$ and control actions $u_{1:t}$. Following Thrun et al. notation [6], a belief over a state variable x_t will be denoted as $bel(x_t)$ which is the posterior density conditioned on all past measurements and control actions.

$$bel(x_t) = p(x_t|z_{1:t}, u_{1:t}) \tag{2.27}$$

Equation 2.28 shows the prior distribution which is called *prediction*.

$$\overline{bel}(x_t) = p(x_t|z_{1:t-1}, u_{1:t}) \tag{2.28}$$

The posterior $bel(x_t)$ can be calculated using the prior $\overline{bel}(x_t)$. This is known as *correction* [11].

The relation between equations 2.27 and 2.28 is determined by equation 2.29 where

the posterior depends on the prior distribution.

$$bel(x_t) = \eta\, p(z_t|x_t)\, \overline{bel}(x_t) \tag{2.29}$$

Where $p(z_t|x_t)$ is the likelihood model for the measurements, a causal, but noisy relationship [32] and η is the *normalization constant* defined as $p(z_t|z_{1:t-1}, u_{1:t})$. More details are provided in appendix A.1.

The notion of belief can be categorized in two important concepts [5]:

- *Single-hypothesis belief*: the posterior is represented by a Gaussian curve. In other words, the robot belief is at one location in the environment with some marginal error.
- *Multiple-hypothesis belief*: the posterior is represented by a mixture of Gaussian curves. In other words, the robot belief is at several locations in the environment with some marginal error.

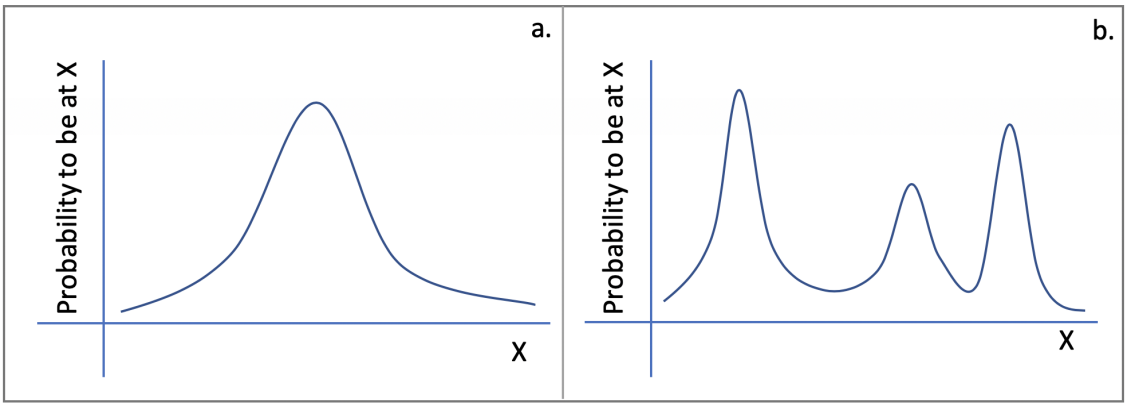


Figure 2.15: a) Single-hypothesis belief: the robot belief is represented by one location over all the possible locations in x. b) Multiple-hypothesis belief: the robot belief is represented by three locations over all the possible locations in x.

Single-hypothesis beliefs are used for local localization in order to track the robot position whereas multiple-hypothesis beliefs are used for global localization due to the powerful representation over all the set of states[1].

2.4.3 The Markov Property

The *MarkovProperty* claims that a future state depends on the present state only and thus it resumes all the past states' information [6].

[1]See section 1.4 to see the difference between local and global localization.

Formally, let x_t be a stochastic discrete process that can take on a discrete set of values, where $t \in \mathbb{N}$. A value in process x_t has the *Markov Property* if for all t such that $t \geq 1$, the probability distribution of x_{t+1} is conditionally independent, given the past values $\{x_t | t = 0, 1, ..., t-1\}$ [33]. That is:

$$p(x_{t+1}|x_t, x_{t-1}, ..., x_0) = p(x_{t+1}|x_t) \tag{2.30}$$

The Markov Property is a convenient assumption, commonly used in mobile robotics since it simplifies tracking, reasoning, and planning and hence it is very robust for such applications [5].

2.4.4 Robot Localization Methods

Sensors and actuators take part in determining the robot's localization but both are subjected to noise and thus the problem of localization becomes difficult. Another problem with sensors is that they usually do not provide enough information content to determine the position of the robot directly and therefore the robot might be in an ambiguous location. This problem is known as *sensor aliasingSiegwart:intro-autonumous-robots* and along with sensors and actuators noise turns the localization problem into a difficult task.

Here, some known techniques to deal with robot localization are mentioned.

Bayes Filter

As claimed by Borriello et al. [12] *Bayes Filter* is a statistical estimator of a dynamic system's state based on noisy observations. In other words, it is an algorithm that calculates the belief's distribution based on measurements and control data [6]. This is generally done in two steps as was mentioned in section 2.4.2: the *prediction* and the *correction* step and thus each time a robot receives the sensor's measurement data, the robot controller software needs to compute the posterior densities shown in equation 2.28 but notice that such a task is computationally heavy. Consequently, its time complexity grows exponentially because the amount of sensor's measurements increases over time. A solution to this problem is to assume that such a dynamic system has the Markovian Property. That is the future state x_{t+1} depends on the present state x_t because of the assumption that x_t carries all the needed information.

Bayes filter algorithm is recursive. It needs the previous posterior distribution to calculate the current posterior distribution. The algorithm is described below and its correctness is shown in appendix A.1.

Algorithm 1: Bayes Filtering Algorithm [6]

input : $bel(x_{t-1})$, u_t, z_t
output: $bel(x_t)$

foreach x_t **do**
  $\overline{bel}(x_t) \leftarrow \int p(x_t|u_t, x_{t-1})\, bel(x_{t-1}) dx_{t-1}$
  $bel(x_t) \leftarrow \eta\, p(z_t|x_t)\, \overline{bel}(x_t)$
end
return $bel(x_t)$

Kalman Filter

The *Kalman Filter* (KF) is a Gaussian technique invented in 1950 by Rudolph Emil Kalman [34] and was first implemented by NASA in the Apollo program to estimate the trajectory of the space capsule in real time [35]. It takes noisy data and takes the noise out to get information with less uncertainty [36]. It works with continuous states and it represents the beliefs by the first and second moments [12] from multivariate normal distributions. Chen et al. [37] have proposed another version of KF called *Maximum Correntropy Kalman Filter* (MCKF) which is an effective and robust algorithm for non-Gaussian signals and heavy-tailed impulsive noise. Instead of using the Minimum Mean Square Error (MMSE) as KF, its optimality criterion is the Maximum Correntropy Criterion (MCC).

Extended Kalman Filter

The *Extended Kalman Filter* (EKF) does not assume any linearity as KF does and therefore the next state and the measurement probabilities are nonlinear functions [38] [6]. Some variants of this method include *Unscented Kalman Filter*, where the state distribution is approximated by a Gaussian random variable as in EKF, but now a minimal set of sample points is cautiously chosen for representing the state distribution [39] [40]. Another improvement of EKF is the *Invariant Extended Kalman Filter* (IEKF) used for continuous-time systems with discrete observations [41].

Grid-based Filters

The *Grid-based Filters* split the environment into small grids, each containing a belief of the true position state [12]. For instance, Burgard et al. [42] proposed a Bayesian approach based on certainty grids where each cell has associated a probability of the robot to be on that cell. Thus they successfully predicted the robot absolute position and orientation in a global localization problem using standard sensors in complex environments.

Topological Approaches

Topological approaches represent the environment using graphs where each node is a representation of a location and the edges are the connectivity between two locations [12].

Particle Filter

Particle filter is a nonparametric implementation of the Bayes Filter algorithm where the posterior is approximated by a set of M samples called *particles* where M is usually a large number (e.g. 2000). Under the context of localization, it is also known as *Monte Carlo localization* (MCL) [43]. Each particle has associated a *weight*, also called *importance factor*, that represents the contribution to the overall estimate of the posterior [44]. Thus, equation 2.31 shows the belief at time t approximated by a set of particles and weights of M particles.

$$bel(x_t) \approx S_t = \{\langle x_t^{[m]}, w_t^{[m]} \rangle | 1 \leq m \leq M\} \tag{2.31}$$

where x denotes the state of particle m, and w represents the weight of such a particle.

According to Thrun et al. [6] the probability of including a particle $x_t^{[m]}$ will be proportional to its posterior belief (see equation 2.32) and thus more particles in a region of the environment means that the true state is more likely to be in that region.

$$x_t^{[m]} \sim p(x_t | z_{1:t}, u_{1:t}) \tag{2.32}$$

Algorithm 2 illustrates the particle filter algorithm. It has two loops, representing the prediction and resampling steps respectively. The *prediction step* obtains M particles. Every particle is associated with a state and a weight. This set of particles corresponds to $\overline{bel}(x_t)$ in the Bayes filter algorithm. Furthermore, the weight for each particle is calculated as a conditional probability of sensing z_t given the state of that specific particle representing the posterior $bel(x_t)$ in the Bayes filter algorithm. The *resampling step* consists of selecting with replacement M particles from the temporal data set $\bar{S}_t$ with probability proportional to each particles' weight. Thus, the particles with lower weights have less probability to be selected [6]. Figure 2.16 illustrates the use of MCL for robot localization.

As claimed by Thrun et al. [6], one drawback of PF is that the performance of the algorithm highly depends on the environment dimension, that is, the higher the dimension, the worst its performance is.

Sanjeev et al. [46] describe several variants of the particle filter:

Algorithm 2: Particles Filtering Algorithm. Adapted from [6].

input : S_{t-1}, u_t, z_t
output: S_t

$\bar{S}_t \leftarrow \emptyset$
$S_t \leftarrow \emptyset$
for $m = 1$ *to* M **do**
  sample $x_t^{[m]} \backsim p(x_t|u_t, x_{t-1}^{[m]})$
  $w_t^{[m]} \leftarrow p(z_t|x_t^{[m]})$
  $\bar{S}_t \leftarrow \bar{S}_t \cup \{\langle x_t^{[m]}, w_t^{[m]}\rangle\}$
end
for $m = 1$ *to* M **do**
  $\langle x_t, w_t\rangle \leftarrow$ draw i from $\bar{S}_t$ with probability $\propto w_t^{[i]}$
  $S_t \leftarrow S_t \cup \{\langle x_t, w_t\rangle\}$
end
return S_t

- *Sampling Importance Resampling Filter [47] (SIR)*: it assumes that a model describing the evolution of the system in time and a model representing the measurement noise are both known. Additionally, realizations can be sampled from the noise distribution and from the prior distribution. Thus the particle's weights are normalized before the resampling step and obtained from $w_t^{[m]} = p(z_t|x_t^{[m]})$.

- *Auxiliary Sampling Importance Resampling Filter [48] (ASIR)*: it uses a characterization of x_t given by $x_t^{[m]}$, noted by $\mu_k^{[m]}$, which can be the mean. The weights are assigned according to:

$$w_t^{[m]} = \frac{p(z_t|x_t^{[m]})}{p(z_t|\mu_t^{[m]})}$$

- *Regularized Particle Filter (RPF)*: it is the same as the SIR filter except that RPF resamples from a continuous approximation of the posterior density $p(x_t|z_{1:t})$ such as:

$$p(x_t|z_{1:t}) \approx \sum_{i=1}^{m} w_t^i K_h(x_t - x_t^{[m]})$$

where

$$K_h(x) = \frac{1}{h^D} K\left(\frac{x}{h}\right)$$

where $h > 0$ is a scalar parameter called the *Kernel bandwidth*, D is the space dimension of x, and w_t^i are normalized weights. The *Kernel density* $K(.)$ is a symmetric probability density function that satisfies:

$$\int xK(x)dx = 0, \quad \int ||x||^2 K(x)dx < \infty$$

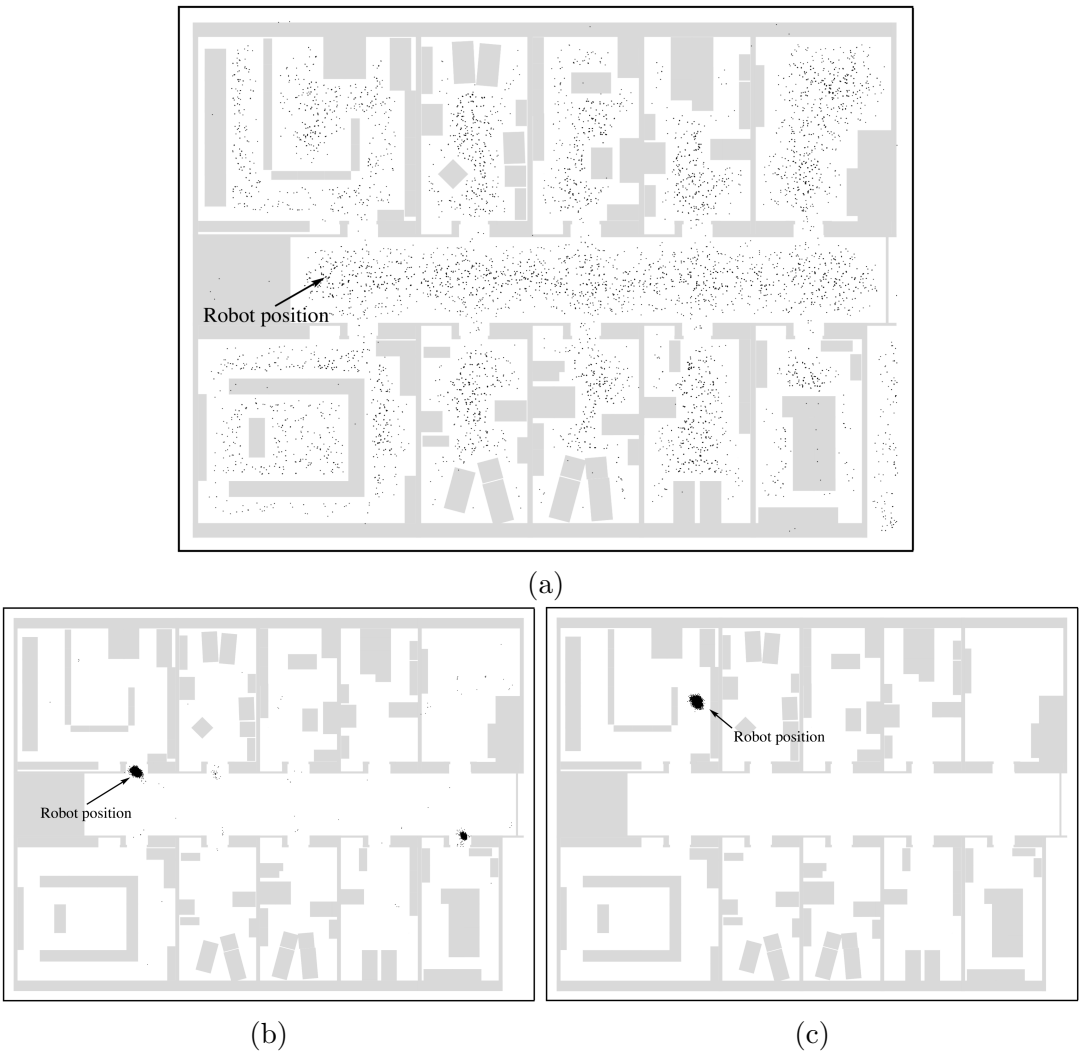


Figure 2.16: A robot localization example using PF. (a) The robot has a global uncertainty about its position in the map, (b) it handles some hypothesis about where it is after executing certain control actions, (c) it is quite sure where it is in the environment.

Source: Particle Filters in Robotics [45].

- *Likelihood Particle Filter*: instead of using the priors, it uses an approximation based on the likelihood.

Sanjeev et al. [46] also compare the performance of such algorithms for a specific problem in which the system dynamics and the measurement noise models are known (see [49]). They use the RMS error as a metric and finally, they show that the likelihood particle filter outperforms the rest closely followed by the ASIR filter.

Thrun et al. [6] also proposed a modified version of the particles filter called *Augmented Particle Filter* which differs from previous versions in the way that it keeps track of the sample's overall weighting over time in order to control the amount of selected samples during the resampling step. In consequence, the algorithm depends on two more parameters that regularize the sensitiveness of the particles selection.

These parameters can be optimized.

The optimal values assigned to the particle filter parameters depend on the nature of the algorithm and the problem which it is oriented to deal with. For instance, Musso et al. [50] found an optimal choice for the Kernel density and bandwidth for RPF. Burchardt et al. [51] used the Particle Swarm Optimization algorithm [52] to find the optimal parameters for the particle filter for robot self-localization using a swarm of Nao robots for the RoboCup Soccer league[2].

2.5 Webots

Webots was created by Cyberbotics Ltd. a spin-off company from the EPFL. It has been developing since 1998. The company currently[3] employs 6 people in Lausanne, Switzerland to continuously develop Webots according to customers' needs. Cyberbotics provides consulting on both industrial and academic research projects and delivers open-source software solutions to its customers. It also provides user support and training. The source code and binary packages are available for free, their team provides support to the users through a Discord channel[4]. The program is available for MacOS, Ubuntu Linux, and Windows operative systems [53].

Webots website has a reference manual that describes nodes and API functions. It has a complete user guide as well endowed with examples of simulations that show the use of actuators, the creation of different environments, geometries primitives, complex behaviors, functionalities, and advanced 3D rendering capabilities. This section is oriented to describe the basics of Webots and is focused on what will be useful to develop the project. For a detailed description please refer to the user guide[5] or the reference manual[6].

2.5.1 Graphic Interface

Webots supports the following programming languages: C, C++, Python, Java, MATLAB, and ROS. Additionally, it offers the possibility of creating a custom interface to third-party software such as Lisp™ or LabView™ using TCP/IP protocol.

Figure 2.17 shows the main graphic interface of Webots. It can be divided into 5 panels:

1. Simulation: graphic visualization of the simulated world objects.

[2] `https://www.robocup.org`
[3] 2019
[4] `https://discordapp.com/invite/nTWbN9m`
[5] `https://cyberbotics.com/doc/guide/index`
[6] `https://cyberbotics.com/doc/reference/index`

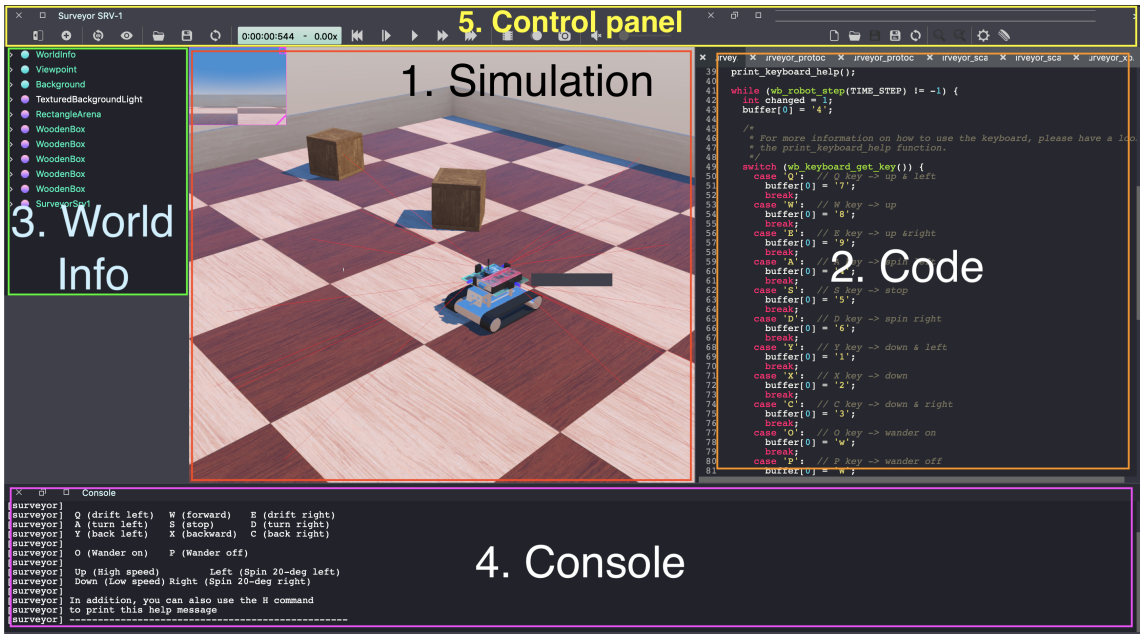


Figure 2.17: Webots graphic interface

2. Code visualization: robot controller code editor.
3. World Information: information represented in a tree structure about the simulated world including the robot components (sensors, actuators, physical characteristics), world objects, textures, etc.
4. Console: program execution output stream.
5. Control panel: the set of buttons that controls the simulation execution.

The program allows users to create highly personalized simulated environments, which are called *worlds*, from scratch using pre-built 3D object models such as robots, wood boxes, walls, arenas, etc. A robot needs to be associated with a *controller program* that contains the source code with the desired behavior. This controller can be easily edited in the code panel. Once it is saved it is automatically reloaded into all the robots associated with it. For running the simulation, users can play, stop, or reset it, among other options, using the control panel set of buttons. The output stream of the controller execution will be displayed in the console panel.

2.5.2 Robots and World Creation

Webots allows creating large simulated worlds. The world description and content is presented as a tree structure where each node represents an object in the world. Those objects have themselves nodes and sub-nodes with a name and a value indicating different physical characteristics or components.

For adding a node into the world we can press the `Add object` button that will open the window shown in figure 2.18. The base nodes are the basic objects that can be

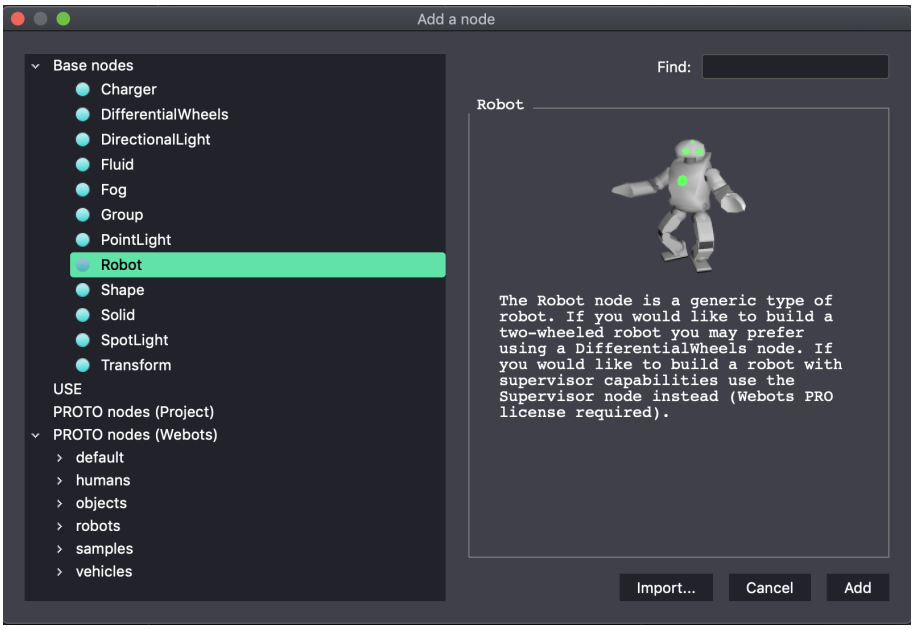
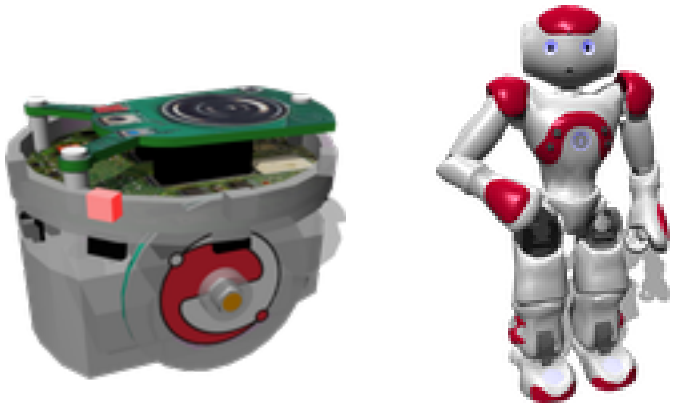


Figure 2.18: World structure

part of the world. Moreover, they are simpler models than others. The USE nodes are objects that were already created in the world and can be reused. For instance, if a wood box was added into the world with some specific physical characteristics, a name can be assigned to its USE attribute, and then it can be reused instead of creating a new one with the same characteristics. The PROTO nodes contain a set of 3D models as robots, objects, vehicles, etc. Fruits, toys, drinks, plants, chairs, stairs, and buildings are only a small part of the big set of modeled objects. The robots node offers as well a large diversity of models. From simple robots as the e-puck (figure 2.19a) model which is used very often in research, to more complex robots as the very well known humanoid Nao (figure 2.19b), these are some examples of a total of 44 3D modeled robots that are available to use within the simulator tool.

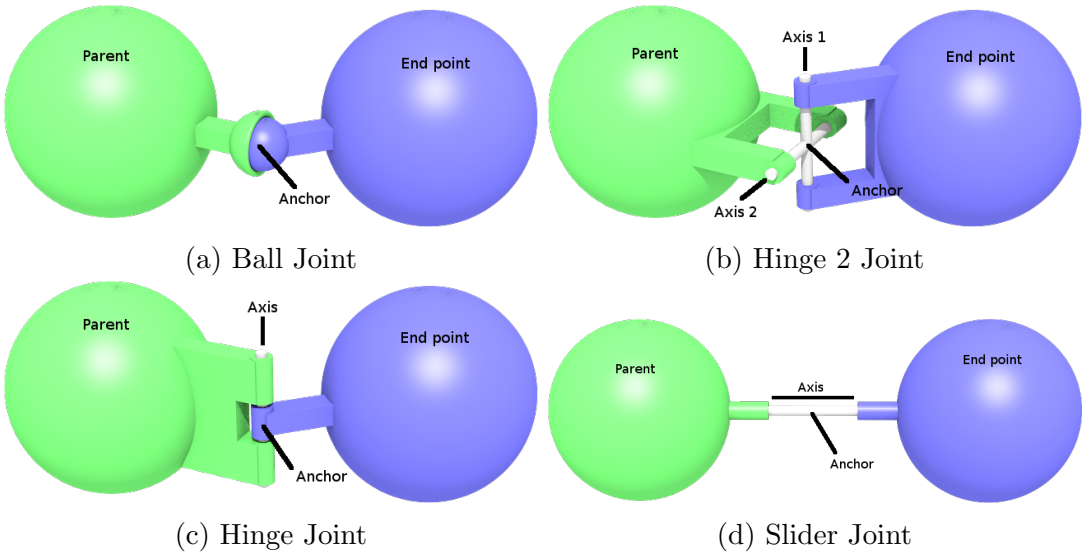


(a) e-puck (b) Nao

Figure 2.19: Robots

Another powerful feature of Webots is to create a custom robot model from scratch using a tree-based structure of solid nodes, which are virtual objects with physical

properties, and put them together using joint nodes, which are abstract nodes that model mechanical joints. Figure 2.20 shows the different types of joints that can be used.



(a) Ball Joint (b) Hinge 2 Joint

(c) Hinge Joint (d) Slider Joint

Figure 2.20: Different types of joint nodes

Source: Webots documentation

A position sensor can be added into the device's property of a joint node to monitor the device positioning. In like manner a rotational motor node can be added for actuating it. Thus, a simplistic version of a custom robot can be created based on the tree information and properties given to the simulator.

2.5.3 Sensors

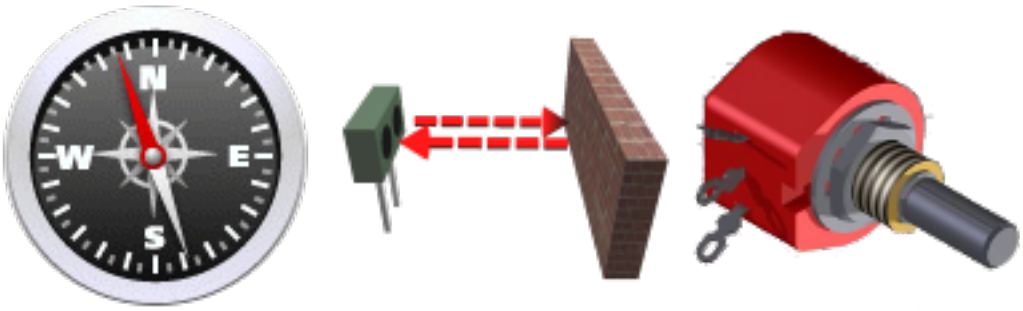
Sensors allow the robot to perceive the environment. Webots has a wide range of generic and commercially available sensors that can be plugged into a custom robot. On one side, the compass, distance sensor, and position sensor are some examples of generic sensors. On the other side, camera, lidar, radar, and range finder sensors built by different manufacturers as Hokuyo, Velodyne, and Microsoft are offered as subcategories of commercially available sensors.

- The **compass sensor** can be used to express the angle orientation of the robot over the position of the virtual north as a 1, 2, or 3-axis vector (roll, pitch, and yaw angles). The virtual north can be specified in the WorldInfo node by the `northDirection` field. The major fields that can be customized are the xAxis, yAxis, zAxis, lookup table, and resolution field.

- The **distance sensor** measures the distance between the sensor, and an object throughout the rays collision with objects. Webots models different types of distance sensors as generic, infra-red, sonar, and laser. All of these have a

lookup table, number of rays cast by the sensor, aperture angle, Gaussian width, and resolution field that can be personalized according to needs.

- A **position sensor** can be inserted into the device field of a Joint or a Track. It measures the mechanical joint position. Moreover, the noise and resolution of the sensor can be customized. This sensor is useful for estimating the position of a robot given the current position of a mechanical joint. This technique is known as Odometry.



(a) Compass sensor (b) Distance sensor (c) Position sensor

Figure 2.21: Sensors

Source: Webots documentation

Sensors have some fields in common like, for instance, the resolution field that indicates the sensitivity of the measurement. When it is set to -1 this means that it will measure the infinitesimal smallest perceived change. As another example, the lookup table is a field that maps a value from the sensor read data to another custom value. Additionally, the standard deviation noise introduced by the sensor can be specified for a given measure.

Table 2.1 shows the possible values that can be associated with the lookup table field. The first column represents the data measured by the sensor, the second column is the mapped value and the third column represents the noise as a gaussian random number whose range is calculated as a percentage of the response value [53]. For instance, for the row $[0.3, 50, 0.1]$ when the sensor senses an object to 0.3m of distance from the robot, it will return a value of 50 ± 5.

Value	Mapped value	Noise
0	1000	0
0.1	1000	0.1
0.2	400	0.1
0.3	50	0.1
0.37	30	0

Table 2.1: Lookup table of a distance sensor

Source: Webots documentation

Figure 2.22 shows the relation between the current sensor values and the mapped values within the associated noise.

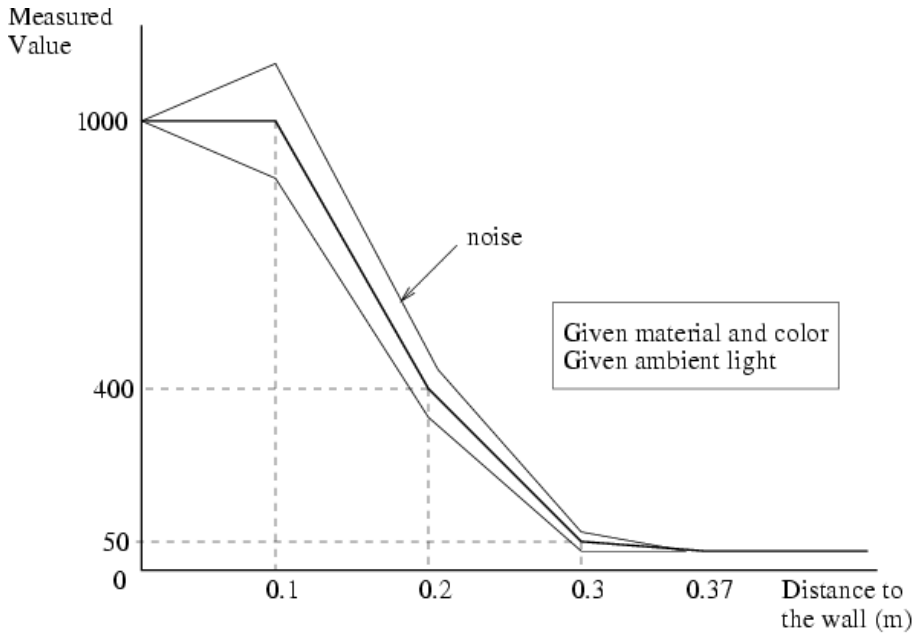


Figure 2.22: Sensor response versus obstacle distance

Source: Webots documentation

2.5.4 Actuators

Actuators allow the robot to modify the environment. Webots can simulate rotational motors, linear motors, speakers, LEDs, etc.

2.5.5 The Supervisor Functions

When the robot supervisor node is set to `TRUE` a new set of functions is available. The supervisor functions are one more level of abstraction and they can be used to replace the human interaction when making experiments. For instance, the robot can be rotated and translated to a random position and the experiment can be reset. In the example below the robot is rotated 1.48 radians and translated to the position $(0.5, 0, 0.5)$ in the arena. One important thing to notice here is that after this change the robot physics needs to be reset; otherwise, an error in the robot wheels' distance to the robot is accumulated over time making them far away from the robot body.

2.5.6 Robot Windows

Webots allows the creation of custom user-implemented windows interfaces for the robots which are integrated into Webots using a controller plugin.

A robot window is a custom window that can be made using HTML and JavaScript. It is associated with a specific robot using its `window` property. The robot windows need to be created under the `plugin/robot_windows/<window_name}>` directory. The window's name should be the same as the controller's name; otherwise, Webots does not recognize it and therefore it does not display it under the windows list.

2.6 Related Work

This section is oriented to show state-of-the-art algorithms using probabilistic theory applied to robot localization.

2.6.1 Differentiable Programming

In order to understand what *differentiable programming* (DP) is, Christopher Olah's explanation [54] is here restated.

Deep learning is a subfield of machine learning that uses artificial neural networks with many hidden layers to learn *representations* from data [55].

Types in programming embed some kind of data in n bits whereas representations in deep learning embed mixed data in n dimensions. Therefore, representations are equivalent to types.

Two functions can be composed together if their types agree. Similarly, two neural networks layers can be composed together if their representations agree. For instance, functions f_1 and f_2 can be composed together if the output type of f_1 is equal to the input type of f_2.

A piece of code that is repeated along a program can be placed into a function and the repeated code can be replaced by function calls, reducing development time. Likewise, many copies of a neuron can be assembled to an artificial neural network to reduce training time. Neurons form blocks following a structure (such as *recursive neural networks* or *convolutional neural networks* [55]) that can be assembled from building blocks. They are equivalent, in functional programming, to a function that takes another function as input.

Conditional branching has also its equivalent in deep learning: a neuron with a Sigmoid function activation. The equivalence of loops is recursive neural networks

which are neural networks whose output is used as input of the same neural networks [56].

DP refers to a programming model that is constructed using neural network blocks with data-dependent branches and recursion, being trainable using backpropagation and gradient descent [57].

Christopher Olah [54] also mentions that DP makes writing programs possible using these flexible, learnable blocks. The correct behaviour of the program can be defined using data, gradient descent can be applied, and as a result, a program is created, capable of doing things that cannot be done by functional programming such as generating a caption that describes an image.

Many variants of particle filter have been popularized lately using this DP approach for addressing robot localization.

2.6.2 Particle Filter Applied to Robot Localization

Zhou et al. [58] proposed an improved PF algorithm called the *Pearson Particles Filter* (PPF) because it is based on the Pearson Correlation Coefficient (PCC). PCC is a statistical technique to determine the linear dependence between two random variables and thus it is used to decide how close the hypothetical particle state is to the true state value. Even though this technique solves the degeneracy and sample impoverishment problem present in PF, it is computationally complex.

Differentiable Particle Filter

Jonschkowski et al. [59] presented the *Differentiable Particle Filters* (DPFs), a differentiable version of the classic particle filter algorithm with end-to-end learnable models. The recursive state prediction and correction steps are encoded in the DPFs structure which is made of a recurrent network representing the filtering loop. Even though their experiments reduce the error rate by 80% compared to algorithmic priors, it presents the limitation of resampling as a non-differentiable operation which stops the gradient computation after a single loop iteration limiting the scope of the implementation to supervised learning.

The popularity of DP has been extended through the field of Probabilistic Robotics. Karkus et al. [60] proposed the *Particle Filter Network* (PF-net), a fully differentiable system model that encodes the particle filter algorithm trained end-to-end from data. Unlike DPFs, PF-net solves the limitation of resampling as a non-differentiable operation presenting a differentiable approximation. PF-Net is used for visual localization with different kinds of cameras: RGB, depth, RGB-D, simulated 2D Lidar, and a Lidar-W. Their observation model receives a spatial transformation of a part of the 2D floor map, and the output of the transition model and the images of the camera. It feeds a neural network architecture composed

by convolutional and fully connected neural networks obtaining the particle likelihood. It also includes semantic information on the map as labels for doors and room types to improve the localization task. They conduct experiments at various levels of uncertainty: local, semi-global and global localization using the House3D data set [16] that contains a large collection of realistic buildings, obtaining better results compared to alternative network architectures such as Histogram Filter (HF) networks [61], Long Short-term Memory networks (LSTM) [62], PF, and odometry.

PF-net is used to develop further works on the domain of Probabilistic Robotics. For instance, *Differentiable Mapping Networks* (DMNs) [63] go one step further targeting the construction of a spatially structured view-embedding map which combined with PF-nets can be used for subsequent visual localization. DMN generates a structured latent map representation based on a set of image-pose pairs. The map representation is made up of pairs of viewpoint coordinates and learned image embeddings. This end-to-end differentiable model is evaluated in 3D simulated environments [64][65] and in more real-world environments using the Street View dataset [66], demonstrating strong performance in both.

Discriminative Particle Filter Reinforcement Learning

Ma et al. [67] introduced a reinforcement learning framework for complex partial observations called *Discriminative Particle Filter Reinforcement Learning* (DPFRL). It uses a DPF in the neural network policy for reasoning with partial observations over time. It is composed of two main components: a discriminatively trained particle filter that tracks a latent belief, and an actor-network that learns a policy given the belief. DPFRL is benchmarked using different problems in different domains. First, the Mountain Hike Task [68], consisting of an agent that goes from a start to a goal position in a map that contains an irregular terrain with partial visibility showing that DPFRL learns faster compared to other models such as *Deep Variational Reinforcement Learning* (DVRL) [68] and *Gated Recurrent Unit* (GRU) [69]. Second, Atari games using an introduced domain called *Natural Flickering Atari Games* that combine two challenges: the *Flickering Atari Game*, the partially observable version of the Atari games, where the observations are completely black frames half of the time and the *Natural Atari Games* where the black background is replaced by a sampled video stream. Results show that DPFRL outperformed in 7 out of 10 games. Finally, visual navigation using an RGB-D camera in the Habitat Environment [17] with the Gibson dataset [70] in which the robot navigates to a goal in previously unseen environments significantly outperforms both DVRL and GRU.

Particles Filter Recurrent Neural Networks

Ma et al. [71] extended Recurrent Neural Networks (RNN) to use particle filters. This is different from RNNs that approximate the belief as a long latent vector, updating it using a deterministic nonlinear function, *Particles Filter Recurrent Neural*

Networks (PF-RNNs) approximate the belief using a set of weighted particles and the stochastic particle filter algorithm to update them. PF-RNNs are applied to LSTM and GRU RNNs architectures, called PF-LSTM and PF-GRU respectively. They are evaluated for the robot localization task in three custom 2D synthetic symmetric maps. The robot uses the distance to a set of landmarks to simulate sensor measurements. PF-LSTM can converge to the true pose even though it has previously converged towards a wrong distant pose which is unusual while using the particle filter algorithm by itself. The results show that PF-RNNs perform better than traditional RNNs architectures on sequence prediction tasks.

Part II

Thesis Contributions

Chapter 3

Webots and Machine Learning

3.1 Introduction

Robot localization can be simplified as a query-answer problem, where a robot is situated into a rectangle closed box, the robot is equipped with a set of sensors pointing to different directions that measure the distance from the robot to the closest wall, the query is the set of robot sensor measurements z_t at time t and the answer should be the robot state prediction $\hat{x}_t$ represented by the robot position and orientation $\{\hat{X}, \hat{Y}, \hat{\theta}\}$ as shown in figure 3.1.

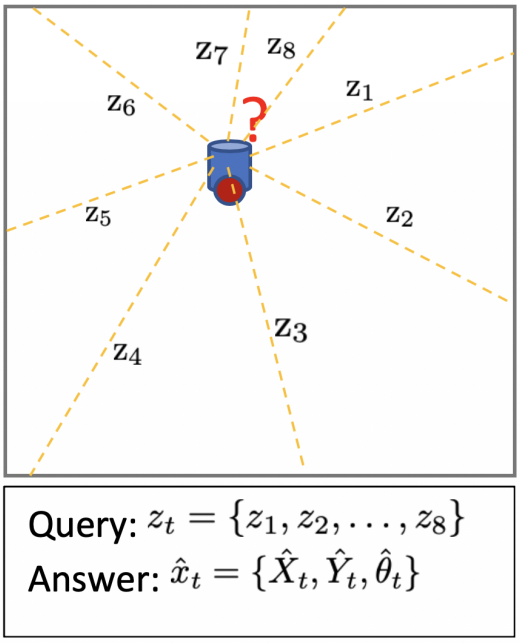


Figure 3.1: Robot Positioning Task

This chapter presents an architecture illustrating how Webots might be used together with machine learning techniques to deal with robot localization.

3.2 Architecture

The architecture presented here is oriented to illustrate how machine learning is used together with probability theory to deal with robot localization. The architecture is composed of *individual components* that interact with each other. Every component is to a methodology and a customize implementation.

Note that this architecture does not refer to the robot's controller software architecture which forms part of one or more components. The five components and their interaction is illustrated in figure 3.2.

The five components are:

- *Virtual world creation:* a virtual world to run experiments is created.
- *Data collection:* true robot state together with sensor measurements are captured using the virtual world.
- *Train models:* a set of machine learning models are trained using the captured data.
- *Predict robot state:* using probabilistic methods together with the previously trained models .
- *Robot window:* a visual tool that shows the odometry, estimated, and true robot positioning together with the set of particles.

The architecture presented should be considered as a reference whose internal components may be changed and adapted by the *program maker* who is in charge of implementing the components. For instance, instead of using neural networks for training, convolutional neural networks can be used. The description of each component will be oriented to the experiments run and explained in more detail in chapter 5.

3.2.1 Virtual World Creation

Methodology

This component is in charge of defining what kind of robot will be used, what sensors the robot should have, how it will interact within its environment, and how the world will be: complex, simple, large, or small, etc.

Webots comes with many world examples from basic worlds for running simple experiments to realistic worlds that contain many real-world objects. It also comes with a wide variety of robots.

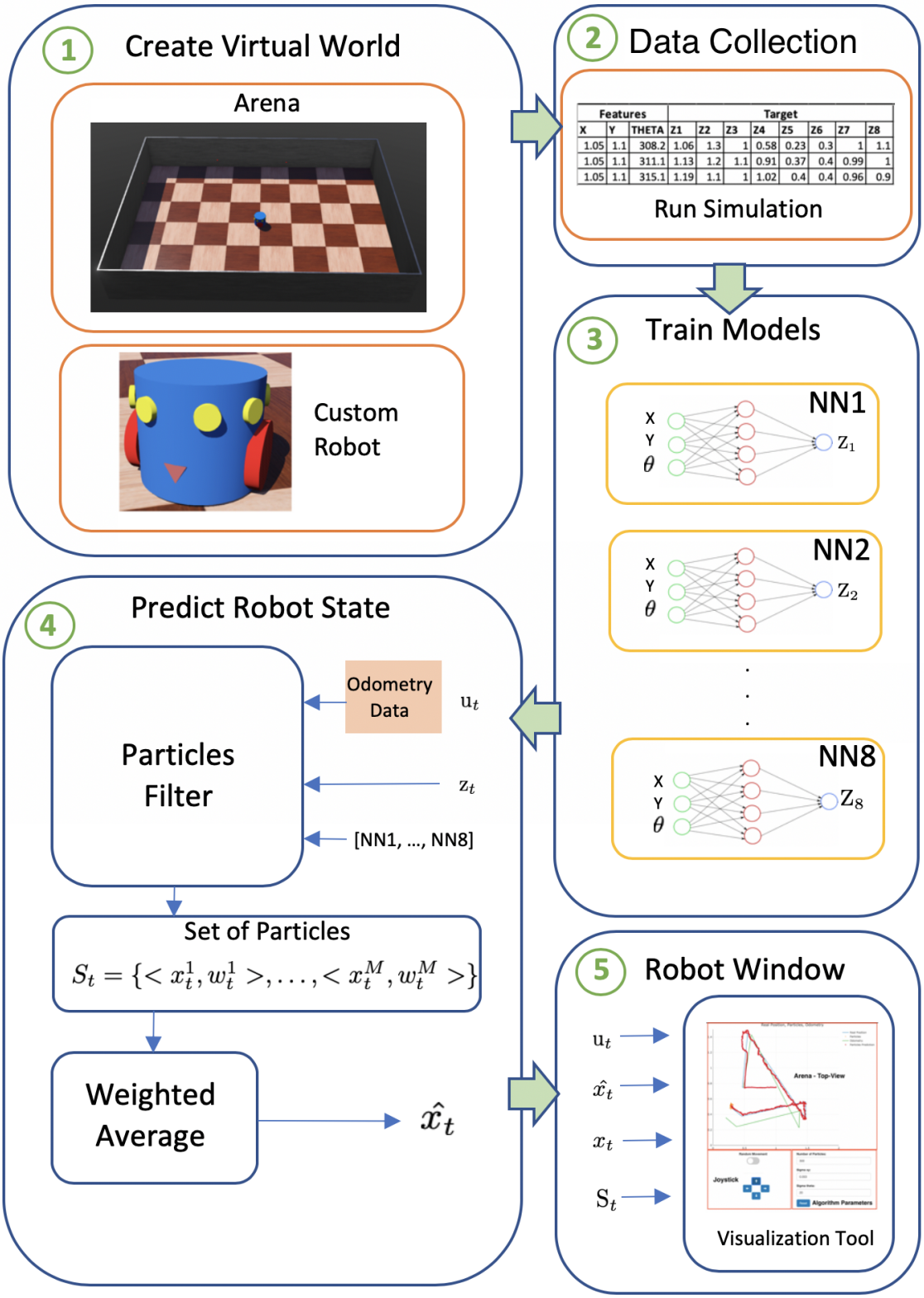


Figure 3.2: Architecture

Implementation

To run experiments with Webots it is important to install all the Python libraries dependencies in an isolated environment, tell Webots to use that environment, and

configure an external IDE such as IntelliJ IDEA [72] to execute and debug that controller. Even though Webots lets the user run the controllers from the tool itself, it is hard to debug when building more robust and complex controllers. A custom local environment configuration is recommended to run and debug a controller in Webots from IntelliJ IDEA using Python 3.7 and a virtual Python environment. A detailed guide on how to make this configuration is found in appendix A.3.

The virtual world consists of a rectangular arena of dimensions 250×150 cm. It is surrounded by four walls of 30 cm of height.

The robot used is a custom-made robot based on the examples provided by Webots.

Webots allows the creation of a robot from scratch. The main benefit of doing so is the high control and customization of the robot model when running different experiments.

A robot is made of solid nodes attached using joint nodes together with device nodes as actuators and sensors[1]. A custom robot will be created based on the Webots examples provided in the `{WEBOTS_HOME}/projects/samples` directory. It is slightly personalized, mainly on the type of distance sensors used. It has the following characteristics:

- A *blue cylindrical body* in which all the components will be attached.
- Two *red cylindrical wheels* with hinge joints, each with a position sensor and a rotational motor. The former serves to obtain the position of the wheel at a given time step, the latter controls the velocity and direction of the wheel. Both have a diameter of 5cm
- Eight *yellow laser-type distance sensors* around the body of the robot that measure the distance between the robot and the closest wall in meters.
- A *compass* sensor for measuring the robot orientation over the virtual north.

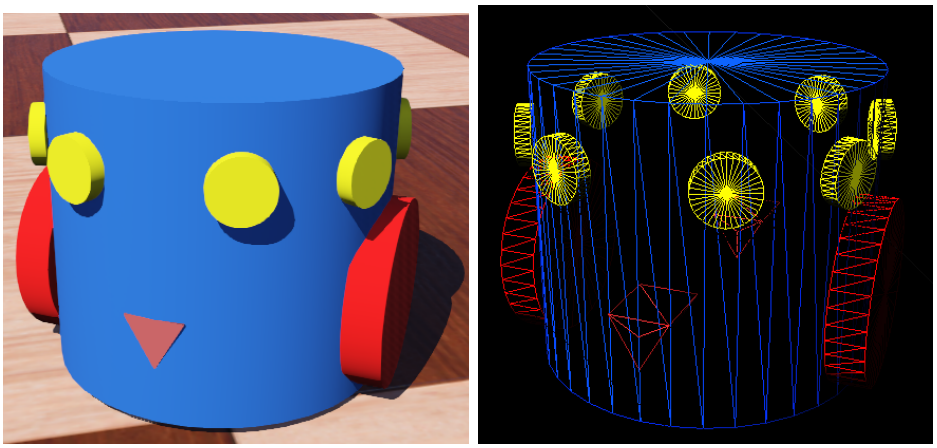
Figure 3.3 illustrates the created custom robot model under two different renderings.

3.2.2 Data Collection

Methodology

After creating the virtual world and the robot the next step consists of capturing data using the sensor measurements and the `Supervisor` class to obtain the true robot position and orientation.

[1]This link contains more detailed information about how the nodes hierarchy are: `https://cyberbotics.com/doc/guide/tutorial-6-4-wheels-robot`



(a) Plain rendering (b) Wireframe rendering

Figure 3.3: Custom robot view

The objective is to have synthetic data that can be used for the component in charge of training the models. The robot should make random walks avoiding hitting walls while capturing data with its sensors and moving to a random position every t time-steps for increasing the variety of samples that can be obtained.

Implementation

A subsumption architecture is used to deal with a simple reflex agent in order to implement a controller that allows the robot to make random walks while avoiding crashing with walls. This robot controller is used in this component only.

When any of the front sensors report that the robot is near 15 centimeters close to a wall the robot will make a turn to avoid hitting that wall. If the robot does not detect any obstacle close around, it will turn θ_{turn} degrees with probability $p_{turn} = 0.2$ where $0 < \theta_{turn} \leq 30$ is selected randomly. The robot chooses to turn to the left or the right randomly.

The robot is placed into a new random position and orientation every 200 time-steps using the `Supervisor` class. This process is done to increase the chances of visiting not-already-seen robot states.

Noise is introduced in the sensor measurements using the concept of Lookup Tables provided by Webots tool. The exact values are found in section 5.1.1 together with the experiments made. Sensors will not report any value with a probability of 0.001.

The simulation runs for thirty minutes in fast mode and data is saved into a `.csv` file.

3.2.3 Train Models

Methodology

The data collected in the *Data Collection* component should be used to train, test, validate, assess, and select the models that will be used for the *Prediction Robot State* component. The machine learning techniques used for such purpose are dependent on the kind of algorithm used for predicting the robot state and the kind of sensors applied. For instance, if the robot has an RGB-D camera, probably the trained models will use convolutional neural networks in conjunction with fully connected networks.

Implementation

The prepossessing actions made to the previously created data set are minimum. The rows that present a missing sensor measurement are deleted. Even though normalizing the data can improve the accuracy of the models, they are fed with raw data while training because those models will be used by each particle in the particle filter algorithm. Thus, normalizing the particles state and denormalizing the sensor measurement prediction for each particle would be extremely costly and thus avoiding this step makes the simulation to run smoothly.

The models are created and trained using Keras and TensorFlow Python libraries.

A neural network is trained per each sensor and thus 8 models are obtained $\{NN1, ..., NN8\}$. A model is fed with a robot state $x = \{X, Y, \theta\}$ and it predicts a sensor measurement $\hat{z}$. The detailed information about the neural network architecture and the accuracy of the model is detailed in section 5.1.

3.2.4 Predict Robot State

Methodology

In this component the program maker should specify what probabilistic algorithm will be used to predict the robot state and how the trained models can help in such a task.

Implementation

The probabilistic algorithm chosen for this component is particle filter, supported by odometry data, sensor measurements, and the trained models can predict the

robot state.

Cyberbotics' Robot Curriculum website [73] provides a good explanation about how to implement odometry estimation and which parameters need to be calibrated to have a precise implementation. These parameters need to be found experimentally since they are not measurable directly with enough precision: the distance of increment for the left wheel, the right wheel, and the distance between both wheels. To have those values well configured four tests need to be performed:

- *Increments per tour:* the number of motor increments made per one complete wheel rotation. While the robot has gone forward, the wheel had made one complete tour.
- *Axis wheel ratio:* the diameter of the wheels divided by the distance between them. The robot turns around its axis and the goal is to end up in the same position as where it began.
- *Wheels diameters:* the robot follows a square trajectory and it should end up in the same position from where it started. If that is not the case, the diameter of one wheel can differ from the other as shown in figure 3.4.
- *Scaling factor:* it configures the scale of the trajectory.

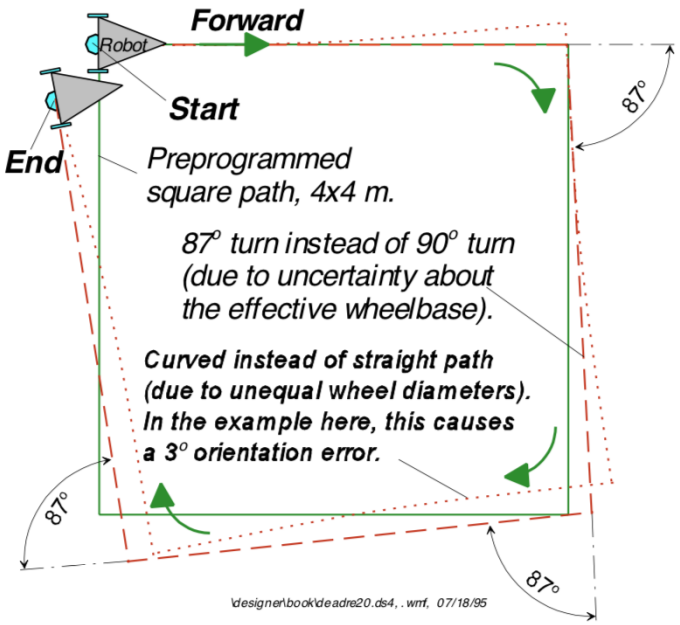
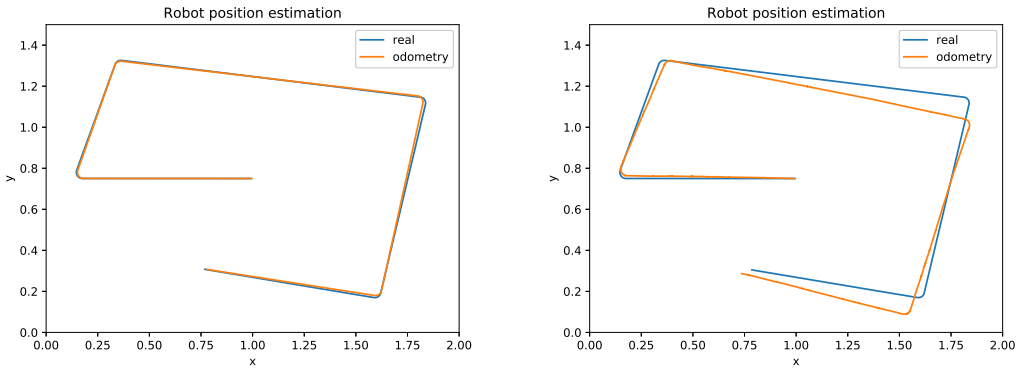


Figure 3.4: Wheel diameters test.

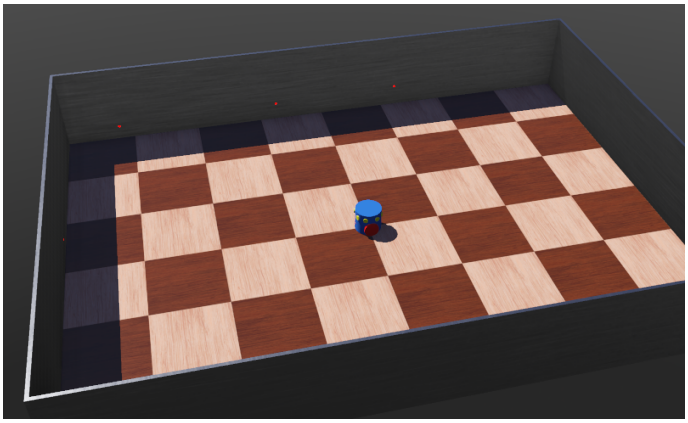
Source: Where am I? [8]

After performing the tests and having an estimation of the corresponding values the results are shown in figure 3.5. The plots were taken in an aerial-view way where the axis represents the width and height of the arena and the lines mark the trajectory of the robot. On the one hand the blue line represents the true state pose of the robot obtained by the supervisor and compass data, on the other hand the orange

line represents the predicted state pose of the robot using odometry techniques. The left-side image was obtained by running one simulation of the mobile robot with zero noise on its position sensors. Even though the odometry measurements approximate very well the true pose, they are still phased out because the parameters used by odometry were found experimentally, and thus some systematic error was introduced. The right-side image shows how the predicted robot state quickly starts to diverge from the true state due to noise associated with the position sensors that affect the data returned by the odometry technique. Thus as time goes on the robot's predicted state will be far from the true state.



(a) Position Sensor Noise = 0 (b) Position Sensor Noise = 0.1



(c) Robot in the arena

Figure 3.5: Odometry results

The particle filter algorithm receives at time t the control actions defined by odometry estimation u_t, the sensor measurements z_t and the set of trained models $\{NN1, ..., NN8\}$. It generates as output a set of particles that contains information about one or more hypotheses of the robot state. Thus the weighted average of these particles' state gives an approximation about the true robot state. The particle filter algorithm is described in more detail in chapter 4.

3.2.5 Robot Window

Methodology

This component is in charge of displaying the results of the true robot state compared to the predicted robot state using a Robot Window plugin. Additionally, some robot-control tasks can be implemented such as moving the robot with a joystick or activating/deactivating some of the robot sensors. Finally, a form containing some specific inputs of the probabilistic algorithm chosen might be implemented to let the user have more control over it.

Implementation

The controller can send and receive messages to the robot window using the command `wwiSendText(text)` and `wwiReceiveText()` respectively. Similarly the window robot can communicate with the controller with the `webots.window(" <window name> ").receive` and `webots.window(" <window name> ").send` commands. The sent messages from and to the controller are in JSON format. This JSON object has an attribute called "code" that contains a unique sequence of characters that allows the entity to recognize what operation it should execute. For instance, if the user disables the random movement in the windows robot, it sends a message to the controller with the format `{"code": "stop_random_walk"}` and thus the controller knows that it should stop the robot from taking random walks.

A robot window is created to visualize data associated with the robot localization task such as:

- The true robot state
- The predicted robot state
- The odometry data
- The particles state

Figure 3.6 shows three components of the robot window. First, the top-view arena visualization where the robot's true state, the odometry estimation, the particle's prediction state, and the particle's positioning is traced while the simulation is running. This component was made using the Plotly open-source JavaScript library[2] to draw charts. Second, the random movement switch that allows disabling/enabling the robot random movement. If the switch is turned off, a joystick component will be displayed that will allow the user to control the robot movements. Finally, the algorithm parameters section allows the user to set some parameters while the simulation is running, like the number of particles, σ_{xy}, and σ_θ parameters which are explained in detail in chapter 4.

[2] `https://plotly.com/javascript/`

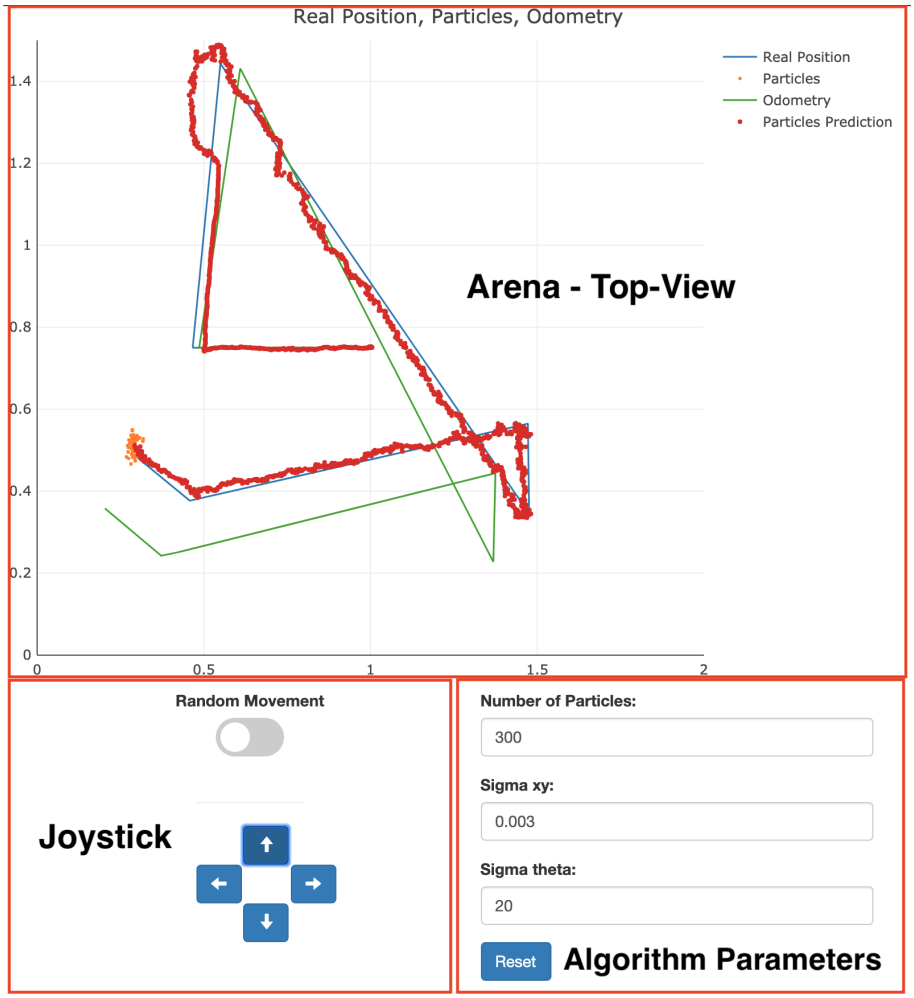


Figure 3.6: Custom robot window for visual robot localization

Figure 3.7 shows through a sequence diagram how the controller communicates with the robot window. First the controller sends an initialization command to the robot window together with some parameters such as the arena size and the value of the parameters. Once the simulation starts at each robot step[3], the controller sends the robot positioning coordinates to the robot window, which is in charge of plotting them.

The random robot movement is disabled when the user clicks on the switch component. This is made through the robot window which afterward sends a message to the robot controller to stop moving the robot. Additionally, a joystick is displayed to the user who has four possibilities: move the robot up, down, left, or right. Once one of these options is selected the robot window sends a message to the controller with the desired movement, then the latter translates this action to the robot wheels' actuators. This process is presented in figure 3.8. When the random robot movement is enabled again this action is communicated to the controller and starts moving the robot randomly.

[3]A *robot step* is referred to one tick on the robot's system clock. In simulation, it represents a turn in the main control cycle.

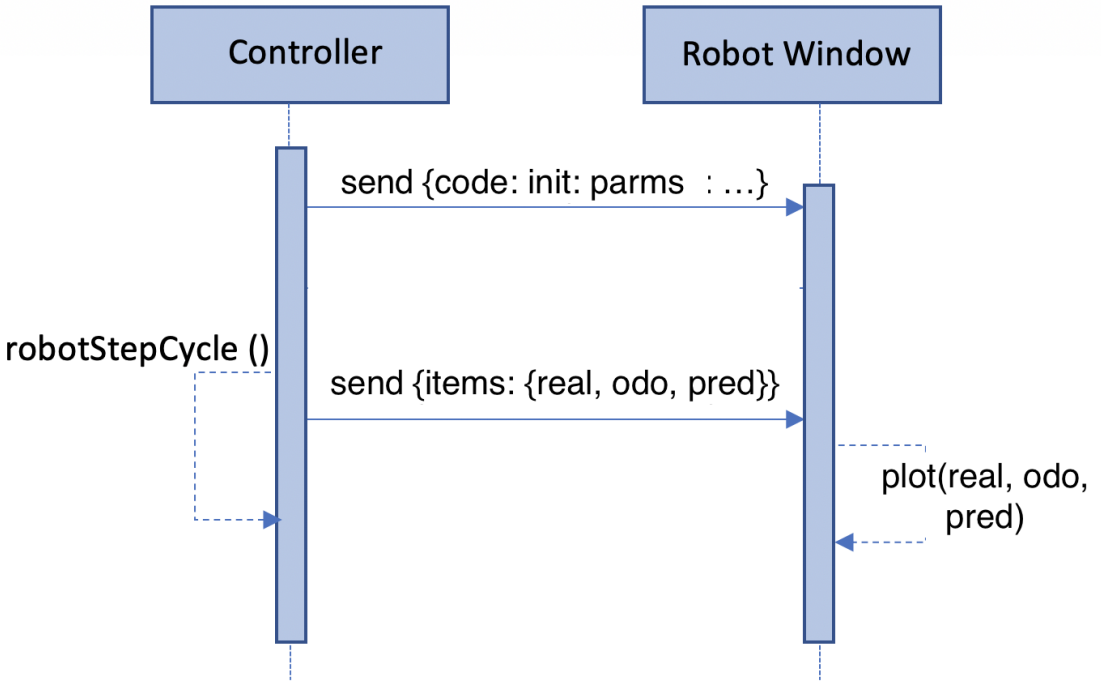


Figure 3.7: Sequence diagram for plotting robot positioning.

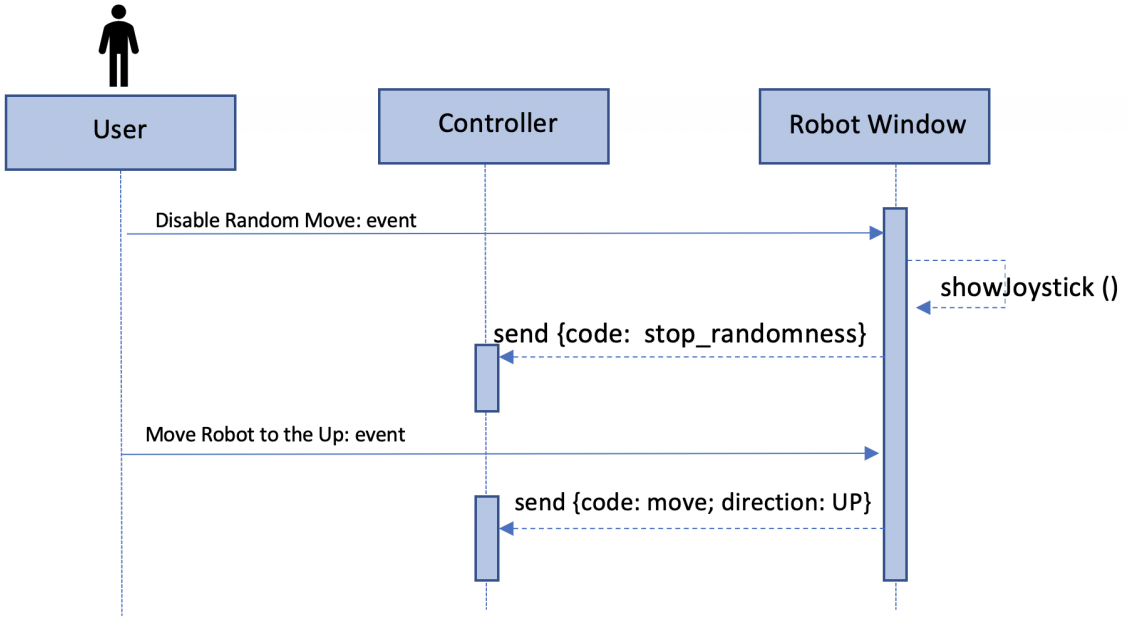


Figure 3.8: Sequence diagram for robot random movement deactivation.

Similarly the algorithm parameters setting sends all the parameters defined by the user from the robot window to the controller to reset the simulation settings.

This robot window offers some control over the robot to see the result of the experiments while running the simulation.

Chapter 4

Empowering Particle Filtering with Machine Learning

4.1 Introduction

This chapter is oriented to explain and illustrate how the classical particle filter algorithm is adapted to make use of machine learning to deal with robot localization. The proposed particle filter is here treated simply as *particle filter*.

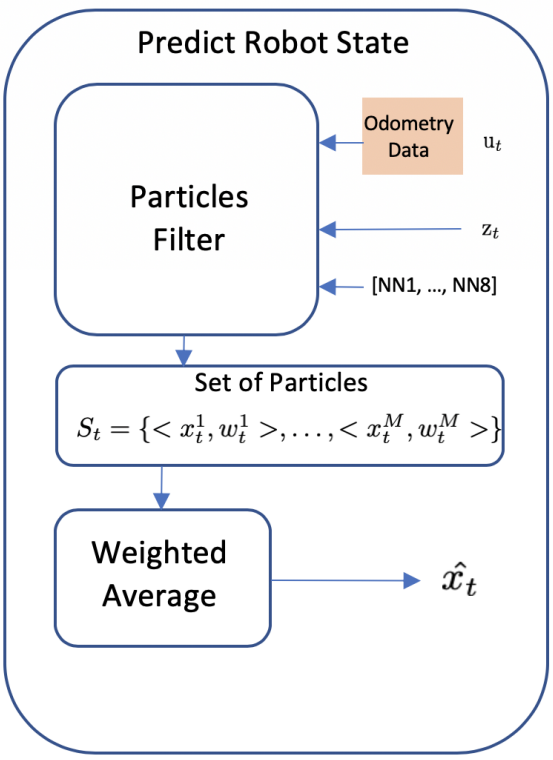


Figure 4.1: Proposed particle filter algorithm overview

The particle filter algorithm receives three elements as input:

- The *control action* u_t given by the odometry data estimation. It is represented by translation and rotation values.
- A *set of trained prediction models*. There is one model per sensor. Each of them predicts a sensor measurement given a particle state.
- The *robot current perception* z_t. The custom robot is integrated with eight laser sensors and thus the robot perception will be $z_t = \{z_t^i \mid 1 \leq i \leq 8\}$.

The output of the algorithm at time t is a set of M particles. Each particle m has information about the robot state belief, and an importance factor referred to the particle's weight denoted as $x_t^{[m]}$, and $w_t^{[m]}$ respectively. A weight $w_t^{[m]}$ represents how close the particle's state prediction $x_t^{[m]}$ is to the true state x_t. The greater the importance factor, the closer the particle's prediction is to the real state.

The predicted robot state depends on all the particle's beliefs. The particles with higher weight are closer to the true robot state. Therefore, the robot prediction state denoted as $\hat{x}_t$ is given by the weighted average of the individual belief of each particle as shown in equation 4.1.

$$\hat{x}_t = \frac{1}{W_t} \sum_{1}^{m=M} w_t^{[m]} x_t^{[m]} \tag{4.1}$$

where W_t is the sum of all the particle's weight $W_t = \sum_1^{m=M} w_t^{[m]}$.

Note that the particles can manage different hypotheses (multi-hypothesis belief) about where the true robot state is. Averaging between two or more hypotheses states can produce a bad prediction because one hypothesis can be closer to the real state and the others far away. As a result the prediction will be some state in the middle of both; however, at some point the particles are able to clarify this ambiguity and converge to the true state (single-hypothesis belief). Moreover, the utility of combining a set of averaging unbiased estimators with large variance is that the variance of the combined estimator is reduced as shown in section 2.1.10.

4.2 Inside the particle filter algorithm

In order to get a good understanding of the implementation details, the particle filter was split into four parts:

- Initialization of particles
- Update particles state
- Calculation particles weight

- Resampling

First, before executing any robot step the particles' initial belief and weight are initialized. Second, while executing a robot step at time t the controller transmits the set of particles S_t, the control action u_t, and the sensor measurements z_t to the particle filter class. Third, the particles state is updated, and their weight calculated using the predicted sensor measurements. Fourth, the particles are resampled according to their weights. This new set of particles will be used as input in the next time step. The process is illustrated in figure 4.2.

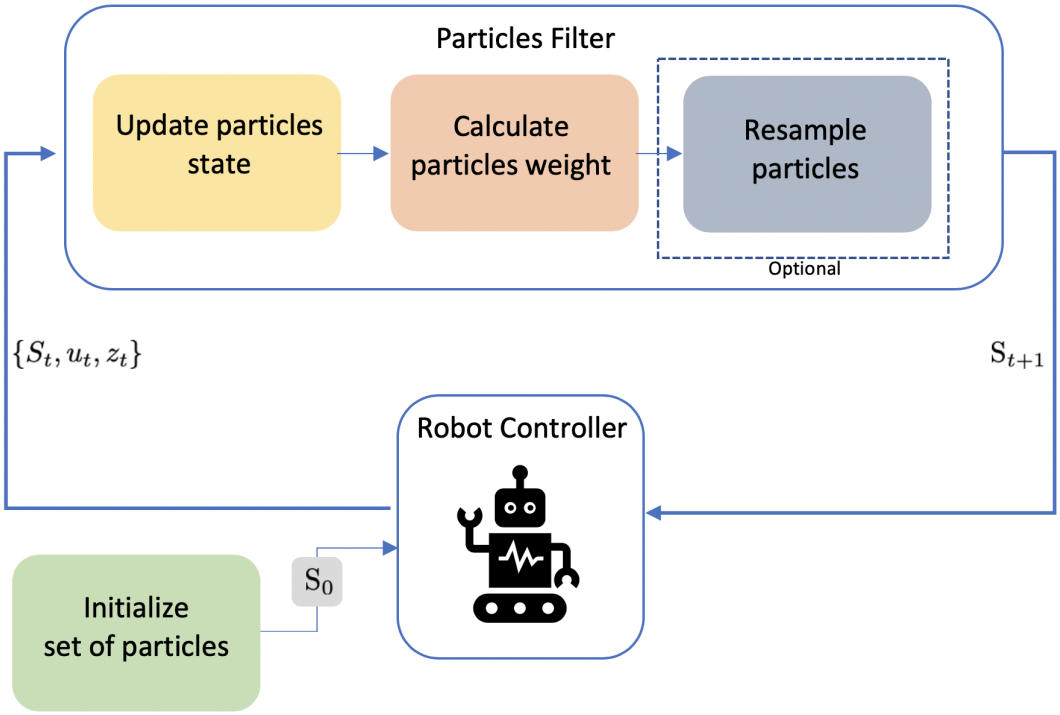


Figure 4.2: An overview inside the proposed particle filter

The Python implementation of the particle filter algorithm can be found on the GitHub repository[1].

4.2.1 Initialization of Particles

The initial belief of the particles is determined by the kind of problem solved. For the local robot positioning problem the particle filter knows the initial robot state and therefore each particle's belief should be closer to this initial state. Nonetheless for the global robot positioning problem the particle filter does not know the initial

[1] `https://github.com/joangerard/webots-thesis/blob/master/webots-project/controllers/pos-prediction/particles_filter/particles_filter.py`

robot state. Thus each particle's belief should be randomly distributed over all the sets of states.

The initial particles state for local localization is sampled from three normal distributions corresponding to the translation and rotation of the particles:

$$\Theta \sim \mathcal{N}(\theta_0, \sigma_\theta)$$
$$X \sim \mathcal{N}(x_0, \sigma_x)$$
$$Y \sim \mathcal{N}(y_0, \sigma_y)$$

where Θ, X, Y are random variables, $\{\theta_0, x_0, y_0\}$ is the initial robot state, σ_θ is the standard deviation parameter for the robot rotation, σ_x and σ_y are the standard deviation parameters for the robot translation in the (x, y) coordinates respectively.

Here, a single value is taken for both σ_x and σ_y parameters. This value is denoted as σ_{xy} where $\sigma_{xy} = \sigma_x = \sigma_y$.

The values taken by σ_θ and σ_{xy} are chosen empirically over a set of candidates as illustrated in section 5.3.

The benefit of using a normal distribution here is that there is one maximum point occurring at the mean which corresponds to the known initial position of the robot. Moreover, the sampling distribution of mean approaches normal as the number of particles increases. In consequence, when using a large number of particles, the normal distribution is a good approximation of the real value. One disadvantage, on the other hand, is that it can generate particles whose state possesses negative values which might cause a particle to lay outside of the arena's dimensions. This disadvantage is compensated by the aforementioned benefits.

The robot's initial state is known in local localization. Thus, the particles' state can be initialized near the true robot's state; however, in global localization the robot lies under global uncertainty and thus the particles' state should be uniformly distributed over the state space. Therefore, the initial particles' state for global localization is sampled from three uniform distributions corresponding to the translation and rotation of the particles:

$$\Theta \sim \mathcal{U}(0, 360)$$
$$X \sim \mathcal{U}(0, env_x)$$
$$Y \sim \mathcal{U}(0, env_y)$$

Where 360 is the maximum robot rotation angle, env_x and env_y are the arena dimensions in the (x, y) coordinates respectively.

Every particle's weight is initialized with value $\frac{1}{M}$ where M is the number of particles. At the very beginning the particles have not established their beliefs yet and they have equal probability to be at the true robot state. This way of initializing the weights produces the prior distribution to be an uninformative prior because it has little influence on the posterior distribution2.1.8.

4.2.2 Update Particles State

When some control action u_t is executed, the robot changes from state x_{t-1} to state x_t. Every particle state makes this transition plus some white noise. The noise is sampled from two different normal distributions corresponding to the translation and rotation of the robot with mean zero and standard deviation σ_{xy}, and σ_θ respectively.

To avoid the particles having an invalid state after the noise addition, such values are clipped to be not greater than the arena dimensions and to be between 0 and 360 degrees for the position and rotation values respectively. In the case of the rotation angles when they have exceeded the limits after the noise addition, the clipping process should recalculate their values instead of cutting it off to 0 or 360 as in the case of the positioning coordinates values. For instance, if the angle value of a particle state is 362 after the noise addition, the clipping process output will be 2 degrees instead of 360; however, when the positioning values of a particle state (x, y) is $(4, 4)$ in an arena whose dimensions are 2×2, the clipping process will cut off those values to be $(2, 2)$.

4.2.3 Calculate Particles Weight

Here is where the particles filter algorithm is empowered by machine learning and differs to the classic particle filter. The trained prediction models are used as a basis to obtain the posterior belief $bel(x_t)$. The models predict sensor measurements given a particle state. This prediction is compared to what the robot perceives, and based on how close the prediction is to the real sensor measurements, a weight is assigned to the particle. In consequence, the set of weighted particles represents an approximation of the posterior belief [6].

More formally, given a particle $\langle x_t^{[m]}, w_t^{[m]} \rangle$, the prediction models $\{NN1, ..., NN8\}$ are fed, each, with the particle state $x_t^{[m]}$, they predict the sensor measurements $\hat{z}_t^{[m]} = \{\hat{z}_t^i \,|\, 1 \leq i \leq 8\}$ corresponding to the eight sensors. These prediction values are compared to what the robot is truly perceiving z_t according to equation 4.2.

$$e_t^{[m]} = \sum_{1}^{i=8} |z_t^i - \hat{z}_t^i|^p \tag{4.2}$$

where $e_t^{[m]}$ is the *sensor prediction error* and p is a hyperparameter that emphasizes larger differences between the true and predicted value.

The error is inversely proportional to the particle weight. The greater the error, the further the particle state is from the true robot state. Moreover, the error is not normalized nor averaged because a normalization process occurs over all the particles' weight afterward. The weight of particle m is then recalculated as in equation 4.3.

$$w_t^{[m]} = \frac{1}{e_t^{[m]}} \tag{4.3}$$

On the one hand, if the value of p is large, only those particles that are really close to the true state will have a large weight. On the other hand, if the value of p is small, the difference in the weight of a particle that is far from the true state compared to a particle that is close from the true state will be minimal.

4.2.4 Resampling

Resampling is the technique in which the particles are drawn randomly with replacement according to their weights. Therefore the particles that have a greater weight have more chances to be chosen compared to the particles whose weight is smaller. This process is used for the algorithm to discard those particles that are not close to the true state and augment the exploration chances of those near the true state. This step is optional and can be executed every t time-steps.

4.3 Summary

The particles initialization for local positioning is described by algorithm 3 using pseudocode. The algorithm inputs are: the number of particles, the standard deviation for translation and rotation, and the initial robot position.

The particles initialization for global positioning is described by algorithm 4 using pseudocode. The algorithm inputs are: the number of particles, the standard deviation for translation and rotation, and the environment dimensions.

The pseudocode for the proposed particle state estimation is described by algorithm 5. The algorithm inputs are: the set of particles in the previous time step, the control action executed by the robot, the robot perception, the trained predictive models, the standard deviation for translation/rotation, the hyperparameter p, and

the number of sensors denoted h.

Algorithm 3: Particles initialization for *local* positioning

input : M, $\{\sigma_x, \sigma_y, \sigma_\theta\}$, $\{\theta_0, x_0, y_0\}$
output: S_1

$S_1 \leftarrow \emptyset$
$W \leftarrow \frac{1}{m}$
for $m = 1$ *to* M **do**
  $\theta \sim \mathcal{N}(\theta_0, \sigma_\theta)$
  $x \sim \mathcal{N}(x_0, \sigma_x)$
  $y \sim \mathcal{N}(y_0, \sigma_y)$
  $x_t^{[m]} \leftarrow \{x, y, \theta\}$
  $w_t^{[m]} \leftarrow W$
  $S_1 \leftarrow S_1 \cup \{\langle x_t^{[m]}, w_t^{[m]} \rangle\}$
end
return S_1

Algorithm 4: Particles initialization for *global* positioning

input : M, $\{\sigma_x, \sigma_y, \sigma_\theta\}$, $\{env_x, env_y\}$
output: S_1

$S_1 \leftarrow \emptyset$
$W \leftarrow \frac{1}{m}$
for $m = 1$ *to* M **do**
  $\Theta \sim \mathcal{U}(0, 360)$
  $X \sim \mathcal{U}(0, env_x)$
  $Y \sim \mathcal{U}(0, env_y)$
  $x_t^{[m]} \leftarrow \{x, y, \theta\}$
  $w_t^{[m]} \leftarrow W$
  $S_1 \leftarrow S_1 \cup \{\langle x_t^{[m]}, w_t^{[m]} \rangle\}$
end
return S_1

Algorithm 5: Predict robot state algorithm

input : S_{t-1}, u_t, $\{z_t^i \,|\, 1 \leq i \leq h\}$, $\{NNi \,|\, 1 \leq i \leq h\}$, $\{\sigma_{xy}, \sigma_\theta\}$, p
output: $\hat{x}_t$

$\bar{S}_t \leftarrow \emptyset$
$S_t \leftarrow \emptyset$
/* proposed particle filter */
for $m = 1$ *to* M **do**
  $x_t^{[m]} \leftarrow$ make transition from state $x_{t-1}^{[m]}$ according to u_t
  $x_t^{[m]} \backsim \mathcal{N}(x_t^{[m]}, \sigma_{xy}, \sigma_\theta)$
  $\{\hat{z}_t^1, \ldots, \hat{z}_t^h\} \leftarrow$ predict using models $\{NNi | 1 \leq i \leq h\}$ and $x_t^{[m]}$
  $e_t^{[m]} \leftarrow \sum_1^{i=h} |z_t^i - \hat{z}_t^i|^p$
  $w_t^{[m]} \leftarrow \frac{1}{e_t^{[m]}}$
  $\bar{S}_t \leftarrow \bar{S}_t \cup \{\langle x_t^{[m]}, w_t^{[m]} \rangle\}$
end
/* resampling */
for $m = 1$ *to* M **do**
  $\langle x_t, w_t \rangle \leftarrow$ draw i from $\bar{S}_t$ with probability $\propto w_t^{[i]}$
  $S_t \leftarrow S_t \cup \{\langle x_t, w_t \rangle\}$
end
/* posterior belief approximation */
$W_t = \sum_1^{m=M} w_t^{[m]}$
$\hat{x}_t = \frac{1}{W_t} \sum_1^{m=M} w_t^{[m]} x_t^{[m]}$
return S_t

Chapter 5

Experiments

The experiments are oriented to show how well the particle filter algorithm combined with machine learning models can perform when dealing with the robot localization problem. Additionally, the different parameters of the algorithm are modified and analyzed to check how they affect the algorithm.

This chapter provides more details about how the models were trained, what architecture was applied, and what parameters were used.

5.1 Train Models

5.1.1 Collecting Data

Before collecting data, some noise was introduced to every sensor defined by the lookup table 5.1. The input distances are mapped to the desired response values with a standard deviation of 0.05. For instance, when the sensor reports a value of $z = 10$, then $Prob(9.95 \leq z \leq 10.05) \approx 0.6827$ [1].

Input distances	Desired response values	Standard deviation
0	0	0.05
10	10	0.05

Table 5.1: Noise on laser sensors

During the data-collecting process the simulation was run in fast mode for 30 minutes collecting 1125965 samples. Figure 5.1 shows the first 5 values of the data set.

The data set was shuffled and split on 80% for training and 20% for testing. Eight models were trained using the true robot state values as input (*x, y, theta*) and the

[1] Refer to the *three-sigma rule* in [74].

	x	y	theta	sensor_1	sensor_2	sensor_3	sensor_4	sensor_5	sensor_6	sensor_7	sensor_8
0	1.050733	1.094226	308.240738	1.061660	1.289745	0.990510	0.577111	0.232157	0.279216	1.000384	1.083896
1	1.052836	1.093243	311.136141	1.129972	1.214009	1.135844	0.905048	0.365532	0.420028	0.993033	1.009313
2	1.054507	1.091473	315.074322	1.185616	1.131318	1.019394	1.015204	0.395438	0.405837	0.956272	0.944564
3	1.056316	1.089796	319.070072	1.098113	1.103999	1.094631	1.069772	0.391122	0.360150	0.881076	1.000029
4	1.058129	1.088026	323.456321	1.327904	1.104746	1.123384	1.032261	0.425896	0.358535	0.842176	0.892464

Figure 5.1: First five rows of the generated data set

corresponding sensor column as target (*sensor_1* for the first model, *sensor_2* for the second model, etc.).

5.1.2 Model Assessment

The optimizer selected is `rmsprop` (Root Mean Square Propagation), it specifies the exact way in which the gradient of the loss will be used to update the parameters [55]. The loss function uses the `MSE` (Mean Square Error) value and the `MAE` (Mean Average Error) as a metric. All the graphs that uses the word *"loss"* as the axis label make reference to the *MSE* value.

$$MAE = \frac{\sum_1^{i=N} |z_i - \hat{z}_i|}{N} \tag{5.1}$$

$$MSE = \frac{\sum_1^{i=N} (z_i - \hat{z}_i)^2}{N} \tag{5.2}$$

where N is the number of samples and z_i and $\hat{z}_i$ are the true and predicted sensor data.

5.1.3 Model Selection

Three neural network architecture candidates were considered. The models were tested using only the third sensor as target under the strong assumption that the model that gives good results for this sensor will also offer similar results for the rest of the sensors.

One Hidden Layer

The first architecture consists on a one-hidden-layer architecture with **n** hidden neurons where $\mathbf{n} \in \{2, 5, 8, 12, 16\}$:

- layers.Dense(**n**, activation='relu', input_shape=(3,)
- layers.Dense(1)

Figure 5.2 shows the MAE values corresponding to the 6 models with one hidden layer for 2, 5, 8, 12, and 16 hidden neurons after k-folding with $k = 4$ and 75 epochs.

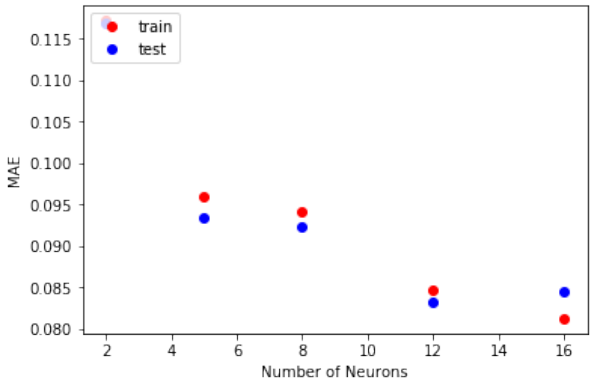


Figure 5.2: Tuning the number of hidden neurons for the one-hidden-layer architecture

The selected model among these candidates is the model with 12 neurons in the hidden layer with MAE value equal to 0.085. This model was here chosen because the MAE value reported for test and train is lower than the rest in both cases.

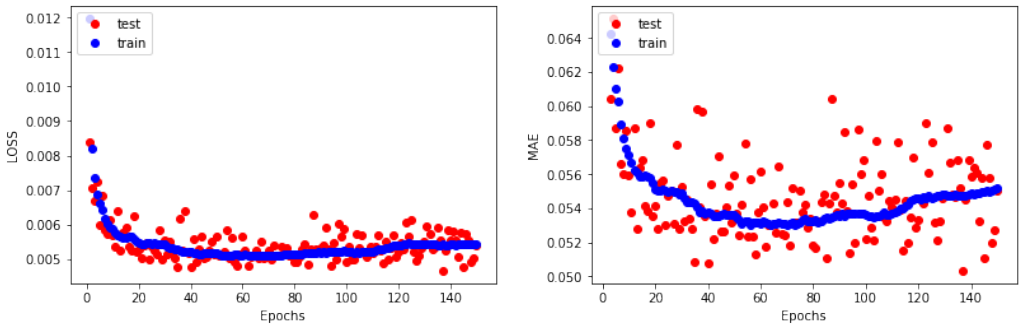
Three Hidden Layers

The second architecture consists of a three-hidden layer architecture with 10, 6, and 3 neurons on the first, second and third hidden layer respectively:

- layers.Dense(10, activation='relu', input_shape=(3,)
- layers.Dense(6, activation='relu')
- layers.Dense(3, activation='relu')
- layers.Dense(1)

Figure 5.3 shows the Loss and MAE values after k-folding with $k = 4$ and 150 epochs.

The MAE curve from epoch 80 starts increasing which demonstrates that the neural network is starting to overfit at this point. Additionally, the training loss curve does not always decrease which is a sign that the neural network has trouble with learning to predict the targeting value.



(a) Loss values for the three-hidden-layer architecture.

(b) MAE values for the three-hidden-layer architecture.

Figure 5.3: Loss and MAE values after 150 epochs and 4-fold.

Four Hidden Layers

Finally, the third architecture consists of a four-hidden layer architecture as follows:

- layers.Dense(16, activation='relu', input_shape=(3,)
- layers.Dense(32, activation='relu')
- layers.Dense(64, activation='relu')
- layers.Dense(16, activation='relu')
- layers.Dense(1)

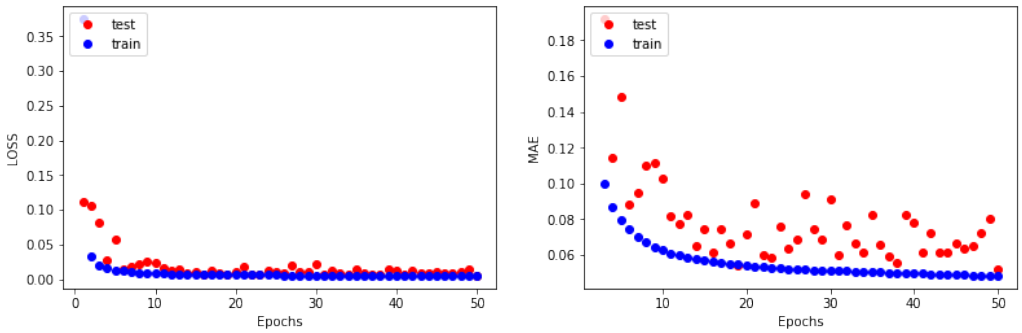
Note that the number of neurons per layer is a multiple of 2 because this increases the opportunity of CPU vectorization, which accelerates the training process.

Mini-batch mode with a batch size of 64 was preferred over Stochastic mode used in previous architecture candidates. The difference between both is that the first uses a batch size greater than one and less than the number of samples while the latter uses a batch size of exactly one. The decision was made because of two factors: velocity on training and MAE values obtained. Training a neural network with batches of 64 is faster than training it sample by sample.

The model was assessed using the k-fold technique where $k = 4$ and 50 epochs. Figure 5.4 shows that the MAE values starts at 0.1 and it decreases down to 0.0458 after 50 epochs. The first loss value on training is 0.35 and it decreases down to 0.0046 without increasing the testing loss function, which is a good sign of generalization.

Final Model Selection

Table 5.2 reports the MAE value for the three aforementioned candidate architectures. The selected model is the neural network with four hidden layers because it



(a) loss for Mini-batch mode. (b) MAE for Mini-batch mode.

Figure 5.4: Loss and MAE values reported for the four-hidden-layer architecture.

reports the lower MAE value among the rest.

Model	MAE
One hidden layer	0.085
Three hidden layers	0.053
Four hidden layers	0.045

Table 5.2: MAE values for the three different architectures.

The same architecture is used to train the eight models, corresponding to the eight sensors, for 50 epochs. The loss and MAE values obtained for each sensor are presented in table 5.3.

Sensor #	loss	MAE
1	0.0902	0.0838
2	0.0148	0.0503
3	0.0044	0.0458
4	0.0049	0.0462
5	0.0046	0.0458
6	0.0043	0.0454
7	0.0148	0.0496
8	0.0898	0.0830

Table 5.3: MAE and loss values obtained after 50 epochs.

Note that the sensors that are in the backside of the robot (3,4,5,6) present lower values compared to the sensors situated in the front side (1,2,7,8).

5.2 Particle Filter Accuracy Metrics

To measure the performance and the accuracy of the particle filter algorithm predictions, two metrics are used: the cumulative error and the Root Mean Square Deviation ($RMSD$).

The cumulative error is a non-decreasing function that depends on the prediction and the true value. It computes the error at time t as a sum of all the previous errors. To have a metric separately for translation and rotation two cumulative functions are used, the first depends on the positioning coordinates and the second depends on the rotational angle as indicated in equations 5.3 and 5.4.

$$CTE_t = \sum_{1}^{i=t} \sqrt{(X_i - \hat{X}_i)^2 + (Y_i - \hat{Y}_i)^2} \tag{5.3}$$

$$CRE_t = \sum_{1}^{i=t} \sqrt{(\Theta_i - \hat{\Theta}_i)^2} \tag{5.4}$$

where CTE_t and CRE_t are the Cumulative Translation Error and the Cumulative Rotation Error at time t, respectively. The true robot state at time step i is denoted as X_i, Y_i and Θ_i for translation and rotation whereas $\hat{X}_i, \hat{Y}_i$ and $\hat{\Theta}_i$ indicates the predicted robot state.

$RMSD$ measures the average difference between the predicted and true values. This metric is used separately for translation and rotation and it is defined by equations 5.5 and 5.6 respectively.

$$RMSDT_t = \sqrt{\frac{\sum_1^{i=t}(X_i - \hat{X}_i)^2 + (Y_i - \hat{Y}_i)^2}{t}} \tag{5.5}$$

$$RMSDR_t = \sqrt{\frac{\sum_1^{i=t}(\Theta_i - \hat{\Theta}_i)^2}{t}} \tag{5.6}$$

Lower values of $RMSDT$ and $RMSDR$ show that the predicted robot states are close to the true robot states.

5.3 Particle Filter Parameters

This section is oriented to study the performance of the algorithm for a set of parameters, such as the number of particles and the noise introduced in the positioning and rotation of the particles.

The simulation runs during 2000 steps, the robot follows a predefined path collecting at each time step the true robot state, the predicted state, and the odometry data.

The trajectory made for the robot is illustrated in figure 5.5. It shows the true (blue) and predicted robot state (red), together with odometry data (green) and particles' state (orange).

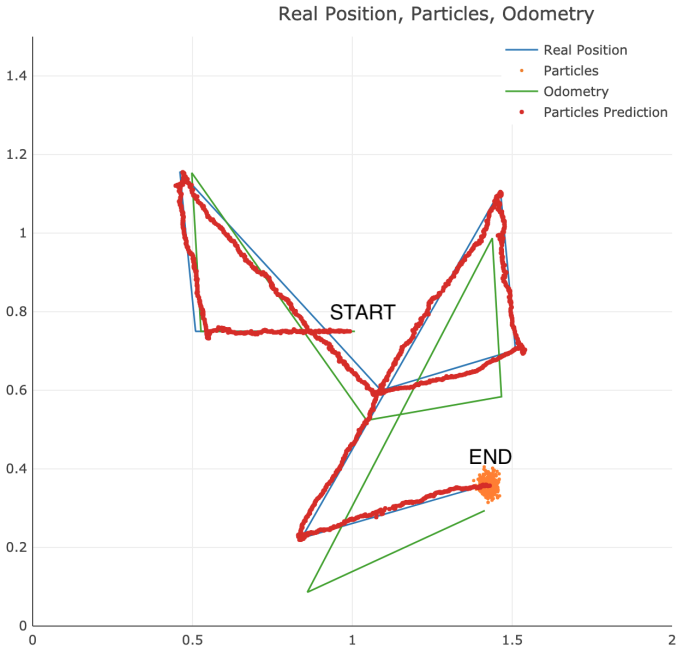
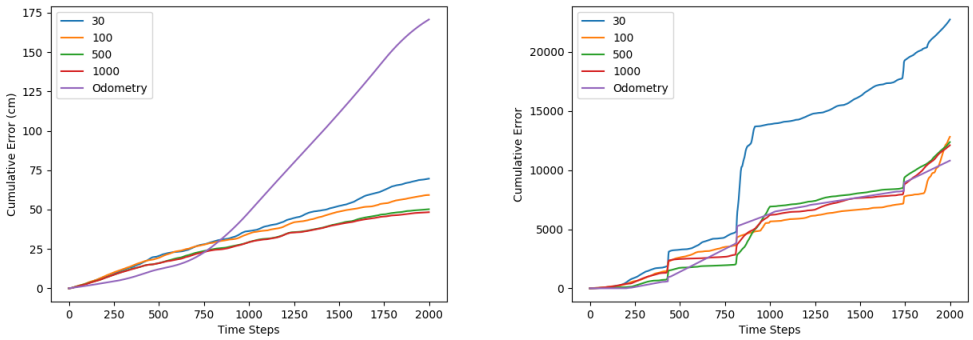


Figure 5.5: Robot Trajectory.

Note that the particles predict the state of the robot at time t and not where the robot will be at time $t+1$ nor what actions should be executed next. Therefore the prediction of the particles is represented as a point in the space of coordinates (representing the $\hat{X}, \hat{Y}$ predicted robot sate) instead of a trajectory line.

5.3.1 Number of Particles

The simulation was run with a different number of particles: 30, 100, 500, 1000. The rest of the parameters were fixed to $\sigma_{xy} = 0.005$, $\sigma_\theta = 20$, and $p = 1$. The cumulative error for translation and rotation is shown in figure 5.6.



(a) CTE (b) CRE

Figure 5.6: Cumulative error for 30, 100, 500, and 1000 particles.

Figure 5.6a shows how the odometry error starts being smaller than the particle's prediction error from time step 0 to 750. At this point the robot executes its

second turn and the odometry error starts to grow faster compared to the particles' prediction error. Therefore, the translation predictions are not affected by the robot turns; however, the rotation predictions are, especially when using a small number of particles as shown in figure 5.6b where the cumulative error grows significantly at time step 750 for 30 particles whereas the rest is closer but not always better than odometry.

Particles Number	**RMSDT (cm)**	**RMSDR (degrees)**
30	3.83	32.01
100	3.26	16.15
500	2.74	14.98
1000	2.66	14.30
Odometry	9.68	21.48

Table 5.4: RMSD metrics for a different number of particles.

Table 5.4 reports the RMSD values for a different number of particles. The best value obtained is highlighted.

Consequently, the more particles are used, the better; however, there is not a big improvement from 500 to 1000 particles where the cumulative error and the RMSD value are closer to each other for both translation and rotation.

5.3.2 Standard Deviation for Particles Translation

The particle's translation is affected by a noise that comes from a normal distribution with mean 0 and standard deviation of σ_{xy} as explained previously in 4.2.2. This experiment consists of running the simulation with different values of σ_{xy}.

Figure 5.7 shows the Cumulative Translation Error for $\sigma_{xy} = \{0.05, 0.005, 0.0025, 0.0005\}$ compared with the odometry error. The experiment uses 1000 particles, with $\sigma_\theta = 20$ and $p = 1$ values. The standard deviation value that obtains better results (less cumulative error) is $\sigma_{xy} = 0.0025$.

$\boldsymbol{\sigma_{xy}}$	**RMSDT (cm)**
0.05	3.43
0.005	2.66
0.0025	2.55
0.0005	4.86
Odometry	9.68

Table 5.5: RMSD metrics for different values of σ_{xy}

Table 5.5 highlights the best value obtained for σ_{xy} which corresponds to the same value found in figure 5.7.

When the σ_{xy} value is close to 0 (as in the case of $\sigma_{xy} = 0.0005$) the prediction errors are not far from the odometry errors because the exploration state space of

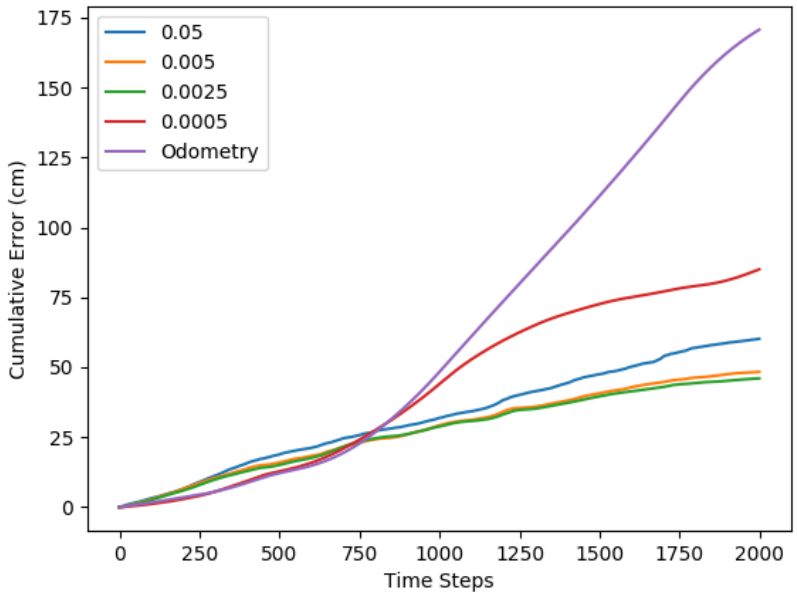


Figure 5.7: CTE for different values of σ_{xy}.

the particles has no more influence over the odometry data. Thus the particles start being distant from the true robot state in early time steps which makes it difficult for them to recover.

5.3.3 Standard Deviation for Particles Rotation

The particles rotation is affected by a noise that comes from a normal distribution with mean 0 and standard deviation of σ_θ.

Figure 5.8 shows the Cumulative Rotation Error for different values of $\sigma_\theta = \{0.005, 0.5, 1, 20\}$ compared with odometry error. The experiment uses 1000 particles with $\sigma_{xy} = 0.0025$ and $p = 1$ values. The best value obtained is for $\sigma_\theta = 0.5$.

$\boldsymbol{\sigma_\theta}$	**RMSDR (degrees)**
0.005	21.49
0.5	8.87
1	9.86
20	17.89
Odometry	21.48

Table 5.6: RMSD metrics for different values of σ_θ

Table 5.6 shows the RMSD values for different values of σ_θ. The best value obtained is highlighted and corresponds to the value reported by the CRE.

Similarly, as discussed previously in section 5.3.2, when this value is close to 0 the

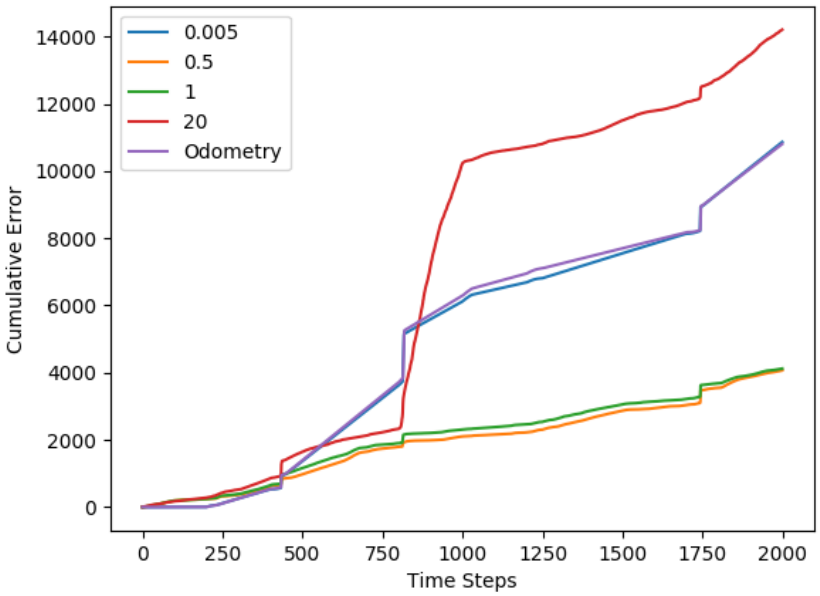


Figure 5.8: CRE for different values of σ_θ.

prediction error matches with the odometry error as for $\sigma_\theta = 0.005$. If this value is large such as in $\sigma_\theta = 20$, when the robot makes a long turn (such as in time step 750) the particles take more time steps to predict a value that is approximated to the true state which induces bad predictions penalized by the CRE.

5.3.4 Hyperparameter: *p*

When calculating the weight of the particles, the error obtained between the true measurements and the predicted measurements is calculated for each sensor and powered to the parameter p as defined in equation 4.2. This experiment studies the influence of such a parameter in the algorithm accuracy.

Figure 5.9 shows the CTE for different values of $p = \{0.05, 1, 2, 3\}$ using 1000 particles with $\sigma_{xy} = 0.0025$, and $\sigma_\theta = 0.5$ values. The larger this value, the more the particles whose sensor predictions are far from the true sensor measurements are penalized. Therefore the weights of these particles will be small, making them less suitable to be chosen when resampling which can cause some particles that are close to the real state to be dropped in early stages. Consequently, the error will be larger as shown on the graph for $p = 3$ compared to $p = 1$. The closer this value to zero, the less importance given to the sensors prediction is given and thus the predicted state will be close to the odometry data offering little improvement as shown on the graph when $p = 0.05$.

Table 5.7 shows the RMSD metrics for different values of p highlighting the best value obtained.

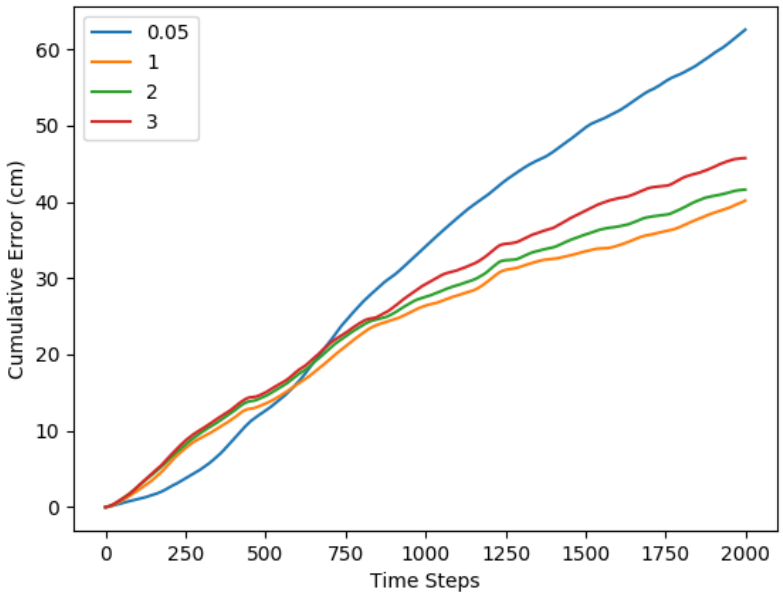


Figure 5.9: CTE for different values of p.

p	RMSDT (cm)
0.05	3.32
0.5	2.42
1	2.24
2	2.32
3	2.51
Odometry	9.68

Table 5.7: RMSD metrics for different values of p

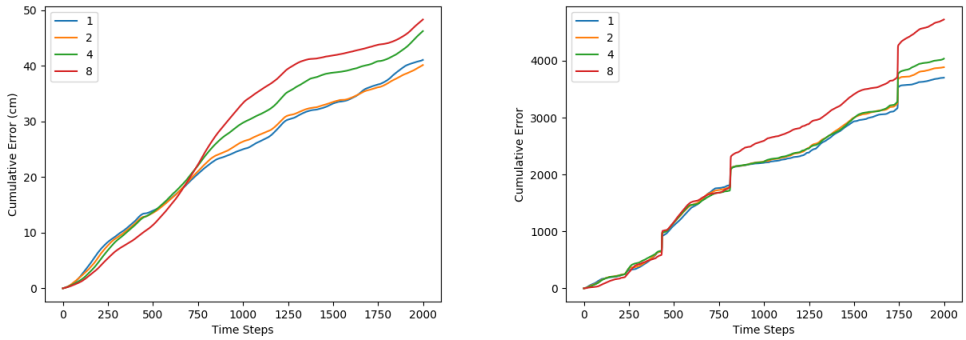
The value of p that gives better results is $p = 1$ with a CTE of 40cm and RMSDT of 2.24cm.

5.3.5 Resampling

In this experiment, the resampling was made each l time-steps where $l = \{1, 2, 4, 8\}$ using 1000 particles with $\sigma_{xy} = 0.0025$, $\sigma_\theta = 0.5$, and $p = 1$ values.

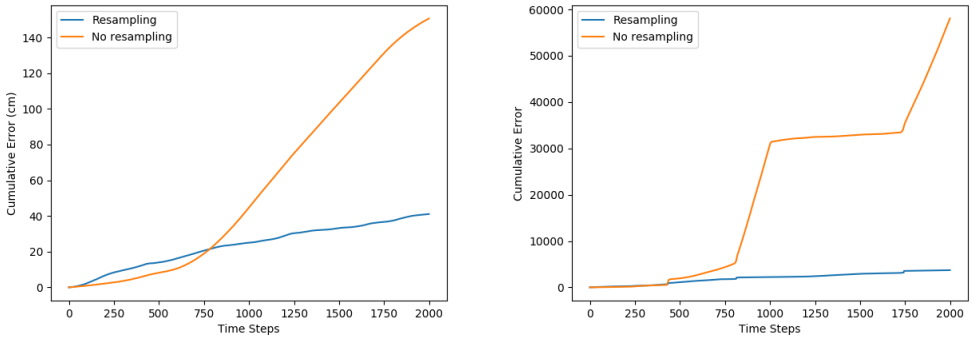
Figure 5.10 shows that after 2000 steps resampling every time step results in fewer prediction errors in the algorithm considering both CTE and CRE. Resampling every eight time-steps has a major benefit in early stages; however, the cumulative error in prediction increases when the robot makes a large turn at time step 750. Thus large turns make the robot state change very fast and the particles start deviating from the true robot state when there is not enough frequent resampling.

Figure 5.11 compares resampling at every time step versus no resampling. The



(a) CTE (b) CRE

Figure 5.10: Cumulative error for resampling each $\{1, 2, 4, 8\}$ steps.



(a) CTE (b) CRE

Figure 5.11: Resampling vs no resampling.

graph clearly shows that when resampling, the prediction made by the particles is more precise than the prediction made when no resampling is used. In the latter case the particles that are far from the true state are still kept even though their weights are close to 0, and the particles that are close to the real state do not have the opportunity to be duplicated and start deviating from the real state at some point.

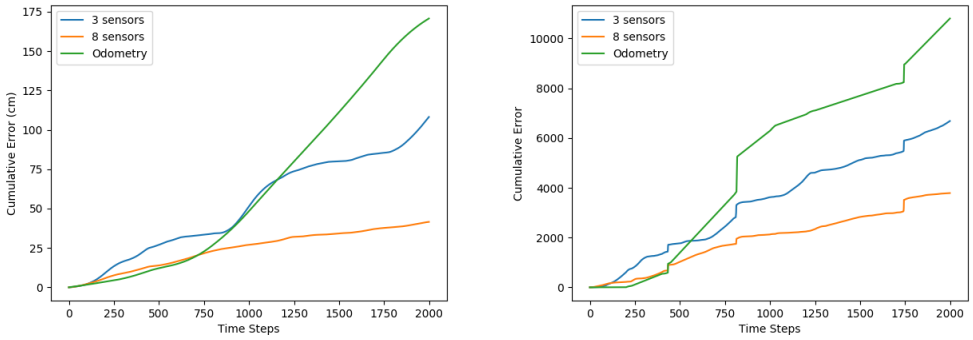
# Resampling	RMSDT (cm)	RMSDR (degrees)
1	2.29	8.70
2	2.24	9.31
4	2.56	9.49
8	2.81	11.18
No Resampling	8.66	54.45

Table 5.8: RMSD metrics when resampling l times for $l = \{0, 1, 2, 4, 8\}$

Table 5.8 reports the RMSD values obtained. Resampling every two time-steps gave better results for positioning and every time-step for translation. It also reports high values when there is no resampling.

5.3.6 Number of Sensors

There are sometimes technical problems with the sensors which can make them stop reporting data. In this experiment, the simulation was run with only 3 sensors (one in the front, and two in the back of the robot) instead of 8 during 2000 time steps, with 1000 particles, $\sigma_{xy} = 0.0025$, $\sigma_\theta = 0.5$ and $p = 1$.



(a) CTE (b) CRE

Figure 5.12: Resampling vs no resampling.

Figure 5.12 illustrates CTE and CRE values. The precision of the prediction is affected especially when the robot turns; however, the error reported by CTE and CRE when the robot walks in a straight line does not increase significantly. After 2000 time steps the CTE and CRE reported values are less than the odometry alone.

# Sensors	RMSDT (cm)	RMSDR (degrees)
3	6.83	11.48
8	2.31	9.32
Odometry	9.68	21.48

Table 5.9: RMSD metrics for 3 sensors, 8 sensors, and odometry.

Similarly, table 5.9 reports lower values for the robot with 8 sensors.

5.4 Increasing Sensor Noise

The objective of this experiment is to analyze how precise the algorithm is when the sensors are not very accurate. Thus the sensor's noise was incremented, and all the processes of collecting data, training the models, and evaluating their performance is repeated. The introduced sensor noise is defined by the lookup table 5.10.

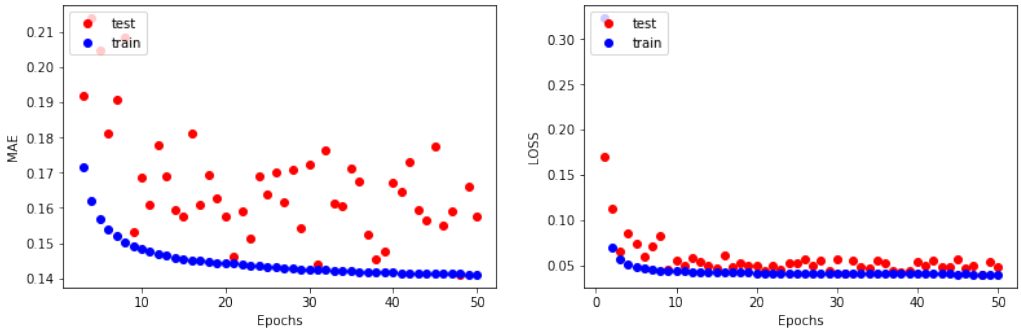
The amount of collected data and the neural network architecture used to train the models are the same as in section 5.1.

Input distances	Desired response values	Standard deviation
0	0	0.2
10	10	0.2

Table 5.10: Noise on laser sensors

5.4.1 Retrain Models

The MAE and loss values obtained for training and testing are shown in figure 5.13. They were obtained by doing a k-fold for $k = 4$ during 50 epochs. The MAE decreases down to 0.1396 for training and the loss value to 0.0394. Note that these values are higher than the values reported in section 5.1.



(a) MAE (b) loss

Figure 5.13: MAE and loss values for the third sensor.

The eight sensor prediction models are trained in a similar fashion obtaining similar results reported in table 5.11.

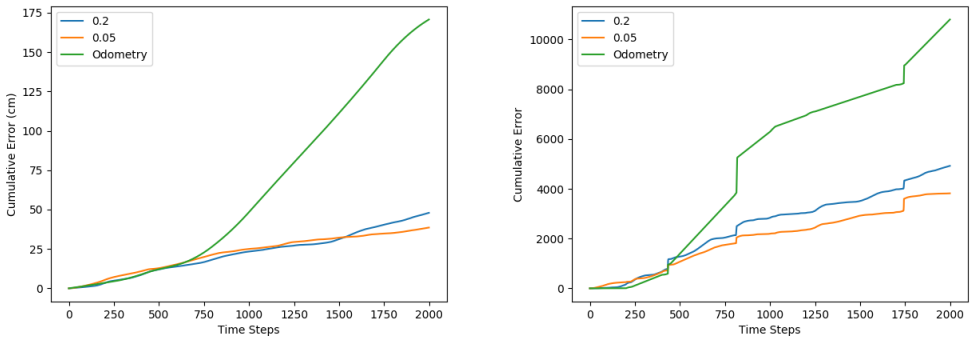
Sensor #	loss	MAE
1	0.1154	0.1643
2	0.0503	0.1462
3	0.0394	0.1396
4	0.0373	0.1367
5	0.0368	0.1359
6	0.0376	0.1362
7	0.0467	0.1420
8	0.1122	0.1629

Table 5.11: Loss and MAE for the eight trained models.

5.4.2 Comparing High vs Low Noise

The simulation was run two times, one using the prediction models obtained in this section where the sensor error is high and the other using the prediction models obtained in the previous section where the sensor error is low. The parameters used

in both simulations were 2000 time steps, 1000 particles, $\sigma_{xy} = 0.0025$, $\sigma_\theta = 0.5$ and $p = 1$. The path followed by the robot is the same as in figure 5.5.



(a) CTE (b) CRE

Figure 5.14: CTE and CRE reported for sensor noise of 0.05 and 0.2.

Figure 5.14 reports CTE and CRE values. Figure 5.14a shows that after 2000 time steps the CTE value for both simulations is close to each other. Figure 5.14b illustrates that the precision of the algorithm for the rotation angles is affected when the sensors are not very precise; however, the cumulative error is in both cases much less than using odometry data alone which demonstrates that even if the sensors are noisy the state predictions are still good.

Sensor Noise	**RMSDT (cm)**	**RMSDR (degrees)**
0.2	2.64	8.68
0.05	2.15	9.45
Odometry	9.68	21.48

Table 5.12: RMSD metrics for sensors with a noise of 0.2 and 0.05.

Table 5.12 reports a lower RMSDT for the model with less noise and a lower RMSDR for the model with more noise.

5.5 Incremental Wheel Diameter

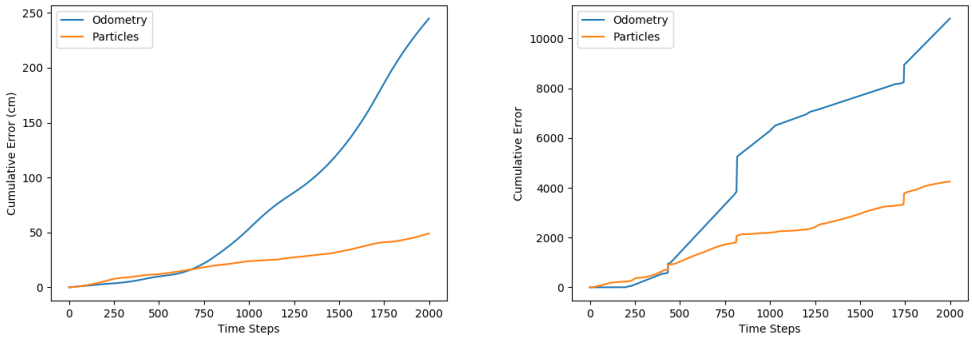
In this experiment, the left and right robot's wheel diameter was increased from 5 cm to 6.5 cm gradually during simulation through all the 2000 time steps according to equation 5.7 where D_l and D_r are the left and right wheels diameter respectively.

$$\begin{aligned} D_l &\leftarrow D_l + 0.00075 \\ D_r &\leftarrow D_r + 0.00075 \end{aligned} \tag{5.7}$$

This change makes the odometry data less and less precise because this data calculation depends highly on the wheels' diameter (see section 2.3.2). Thus the objective

of the experiment is to check if the algorithm precision is being affected by such changes.

To run this experiment 1000 particles were used with $\sigma_{xy} = 0.0025$, $\sigma_\theta = 0.5$, and $p = 1$ values.



(a) CTE (b) CRE

Figure 5.15: CTE and CRE reported when the wheel diameter increases from 5 cm to 6.5 cm.

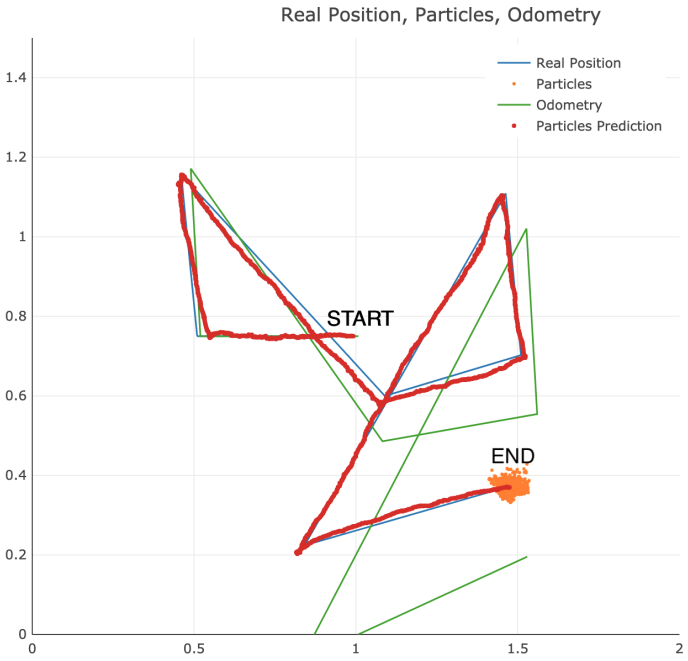
Figure 5.15 shows how the odometry error increases as time goes on; however, the prediction made by the particles is not affected heavily. Additionally, the odometry rotation data is the same as in previous experiments since this value depends mainly on the axis wheel ratio and not on the diameter of the wheels.

	RMSDT (cm)	**RMSDR (degrees)**
Particles	2.67	9.22
Odometry	15.03	21.48

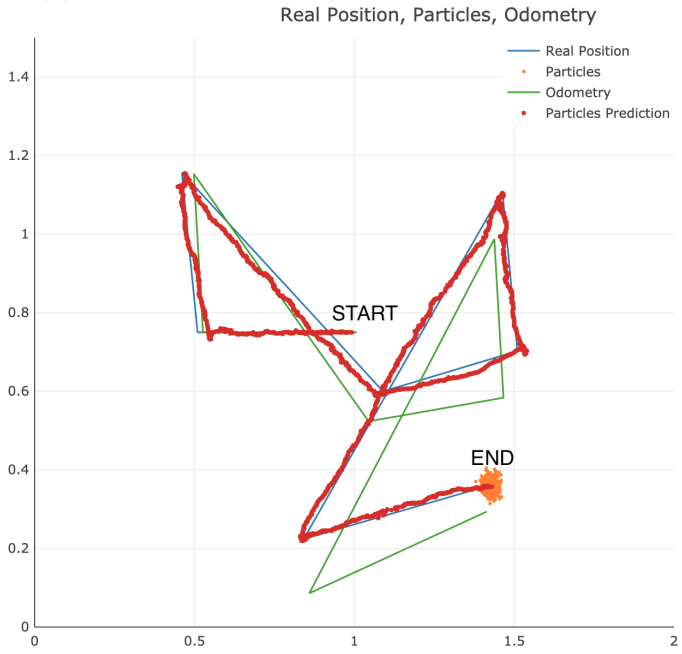
Table 5.13: RMSD metrics for incremented wheel diameter.

Table 5.13 shows the RMSD values for the prediction of the particles compared to the odometry data. Even though there is an increment of 0.52 cm on the RMSDT value compared to the best value found on previous experiments such as in table 5.12, this increment is not significant considering the inaccuracy of odometry data received as input.

Figure 5.16 compares the odometry data and predictions made for a robot whose wheel diameter does not increment versus a robot whose wheel diameter increments in time. Note that in figure 5.16a the odometry data is less accurate compared to figure 5.16b, especially for the last time steps when the wheel diameter has almost reached 6.5 cm; however, the prediction of the particles is near to the true robot state constantly which demonstrates the robustness of the algorithm.



(a) Trajectory with incremental wheel diameter.



(b) Trajectory with no incremental wheel diameter.

Figure 5.16: Incremental wheel diameter vs no incremental wheel diameter trajectories.

5.6 Other Environments

In this section two more environments have been created. The process of capturing data, training models, and assessing the algorithm accuracy is repeated for both environments using the previously defined methods. The robot follows a custom-made path. The objective of such experiments is to check if the localization algorithm can be accurate in a more complex environment.

Before the experiment the robot does not have any knowledge about the environment (it has not seen it yet). After training, the robot does learn about what data the sensors should report given the robot's pose in the environment. This information is then used by the particles filter to estimate the true robot pose while moving.

5.6.1 Medium Complexity Arena

Figure 5.17 shows the defined arena together with the robot trajectory which starts and ends where the red labels are, following the direction of the blue arrows. The arena has dimensions 3×3 meters.

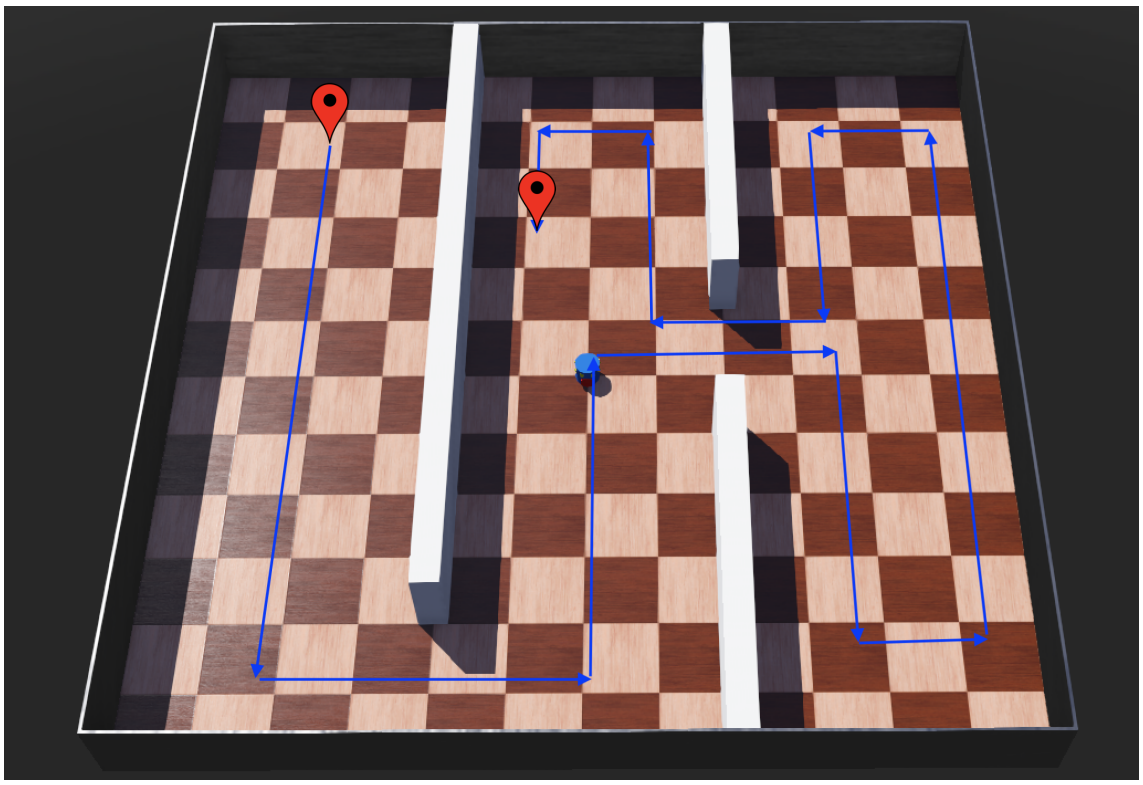


Figure 5.17: Arena with robot trajectory.

To capture sensor and positioning data on the new environment, the simulation was run for 30 minutes in fast mode which collected more than 3 million samples. They were shuffled and only 1125965 samples were used to train the models as in previous experiments, and thus the training time did not increase. The neural network architecture used was the same as in section 5.1.3. Figure 5.18 shows the MAE and loss values obtained after k-folding for 50 epochs with $k = 4$.

The other sensor models were trained similarly. Table 5.14 shows their corresponding loss and MAE values for training and testing after 50 epochs.

Figure 5.19 shows the trajectory made by the robot from a top view of the arena.

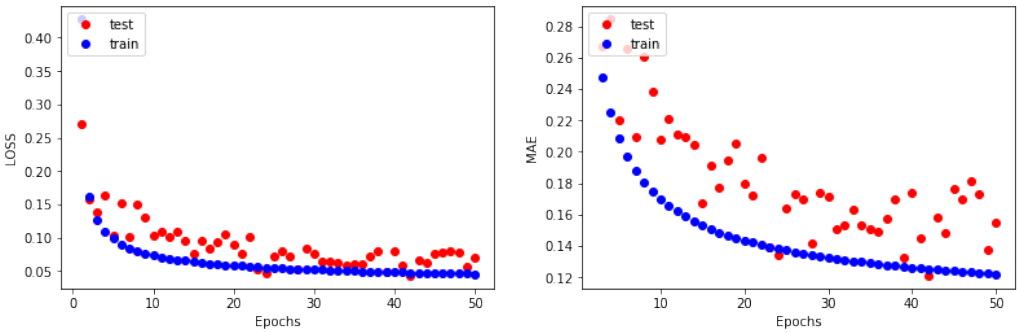


Figure 5.18: MAE and loss for the third sensor after k-folding with $k = 4$.

Sensor #	loss	MAE
1	0.1091	0.1402
2	0.0631	0.1252
3	0.0412	0.1143
4	0.0480	0.1269
5	0.0519	0.1276
6	0.0428	0.1121
7	0.0584	0.1175
8	0.1122	0.1444

Table 5.14: MAE and loss for all the eight trained models.

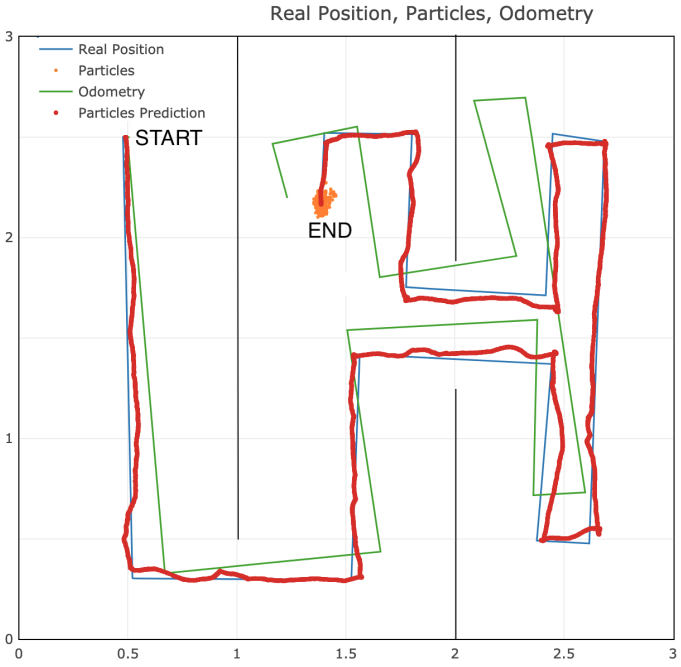
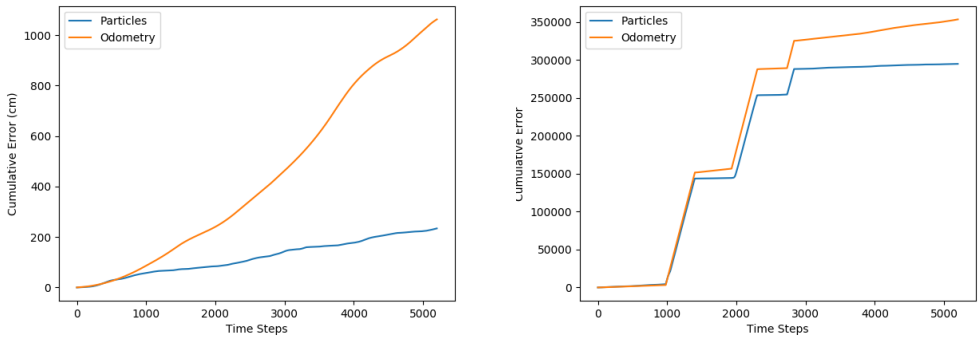


Figure 5.19: Robot trajectory: true state vs predicted state vs odometry.

The algorithm uses 1000 particles with $\sigma_{xy} = 0.0025$, $\sigma_\theta = 0.5$, and $p = 1$ values. The particles prediction which is in red is close to the real robot state showed in blue while odometry data showed in green deviates quickly from the true state.

Figure 5.20 shows the CTE and CRE values for the trajectory made. It can be seen

that both the positioning and rotation predictions accumulated errors are less than the errors that accumulate odometry alone.



(a) CTE (b) CRE

Figure 5.20: CTE and CRE reported for trajectory made in the medium complexity arena.

Finally, table 5.15 shows the RMSD values in which both RMSDT and RMSDR are less for the prediction algorithm compared to odometry alone. A significant increment in the rotational angles prediction error can be reported compared to previous experiments.

Sensor #	**RMSDT (cm)**	**RMSDR (degrees)**
Particles	5.24	134.59
Odometry	22.41	146.73

Table 5.15: RMSD metrics for medium complexity arena path.

5.6.2 Medium-High Complexity Arena

Figure 5.21 shows the arena together with the robot trajectory. The arena has a dimension of 3×3 meters.

To collect sensor and positioning data, and train the sensor prediction models, the same procedure reported as in section 5.6.1 was followed. Figure 5.22 shows the MAE and loss values during training and testing in 50 epochs for the third sensor after k-folding with $k = 4$.

Table 5.16 shows the loss and MAE values for the rest of the sensor prediction models after 50 epochs.

The algorithm was run using 1000 particles with $\sigma_{xy} = 0.0025$, $\sigma_\theta = 0.5$, and $p = 1$ values. Figure 5.23 shows the trajectory made by the robot from a top view of the arena. It can be seen at the end of the trajectory how the odometry state estimation ends 50 cm away from the true robot positioning while the prediction of the particles is near the true state. Even though the particle's estimation becomes inaccurate at

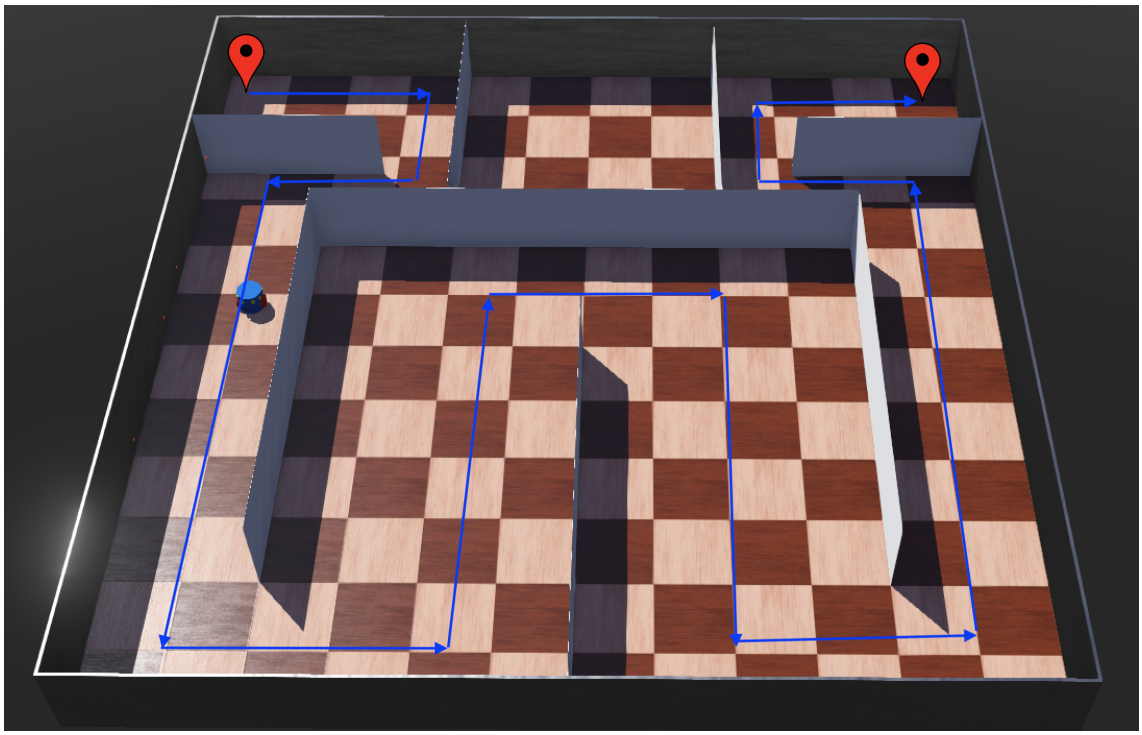


Figure 5.21: Arena with robot trajectory.

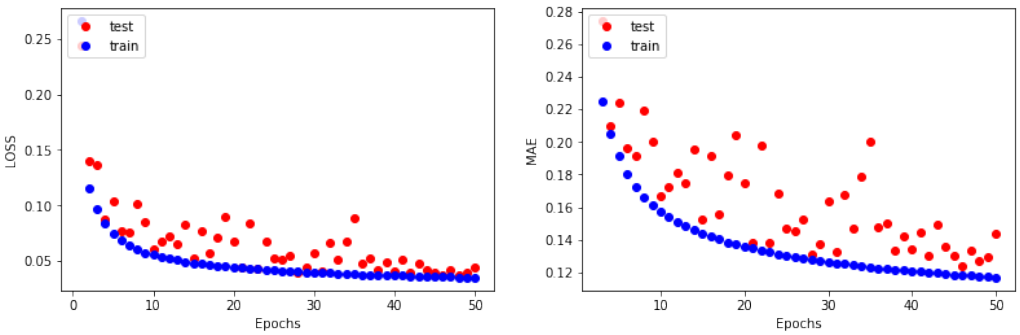


Figure 5.22: MAE and loss for the third sensor after k-folding with $k = 4$.

Sensor #	**loss**	**MAE**
1	0.0636	0.1141
2	0.0398	0.1143
3	0.0327	0.1118
4	0.0367	0.1088
5	0.0425	0.1191
6	0.0341	0.1061
7	0.0437	0.1138
8	0.0617	0.1134

Table 5.16: MAE and loss for all the eight trained models.

some point after the third robot turn, it recovers in the fifth robot turn and starts getting nearer to the true robot states until the end of the trajectory.

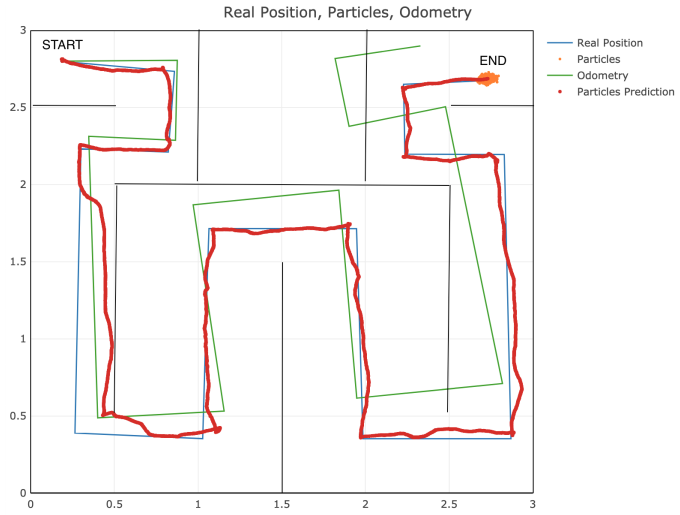
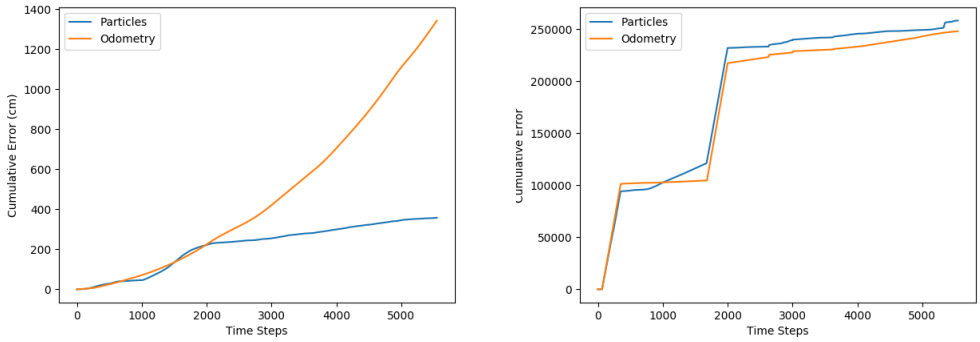


Figure 5.23: Robot trajectory: true state vs predicted state vs odometry.

Figure 5.24 shows the CTE and CRE values. Even though the CTE values when using the prediction algorithm are much less than using odometry alone, the CRE values report a small or null benefit when using the prediction algorithm. This can be caused by the sensor prediction models being trained with the same amount of data regardless of the complexity of the environment, which could not be enough to make a good generalization concerning the rotational angles of the robot.



(a) CTE (b) CRE

Figure 5.24: CTE and CRE reported for trajectory made in the medium-high complexity arena.

Similarly, table 5.17 shows the RMSD values reporting a small improvement when using the prediction algorithm for the rotation states prediction compared to odometry alone; however, there is a significant improvement regarding the predicted translation states when using the prediction algorithm.

In conclusion, the predictions made by the algorithm are close to the true state especially for positioning states.

Sensor #	RMSDT (cm)	RMSDR (degrees)
Particles	8.93	113.28
Odometry	27.39	118.00

Table 5.17: RMSD metrics for medium-high complexity arena path.

5.7 Global Localization

For the global localization problem, the particles do not know the initial state and thus they should be able to figure out where the robot is in the environment. This experiment is run using the medium-high complexity arena with 3000 particles, $\sigma_{xy} = 0.005$, $\sigma_\theta = 10$, and $p = 1$ values. The particles' state was initialized randomly in the arena and the experiment was run 10 times with different initial positions of the robot. The goal is that the particles state should be able to converge near the true robot state after some time steps which is called a *success*; otherwise, it is considered a *failure*.

x	y	result
2.25	1	Success
1.5	2.5	Success
2.5	2.5	Success
0.5	1.5	Success
0.75	1.7	Success
0.5	0.3	Success
2.5	0.3	Success
2.75	1	Failure
1.75	1.3	Failure
0.5	2.5	Failure

Table 5.18: Initial robot states on the arena and the experiment results. Seven out of ten gave a positive result where the particles' state converged near the true state; however, three out of ten failed.

Table 5.18 shows the result for the ten simulations in which seven out of ten of the particles' state converged near the true robot state. The symmetry of the arena makes the problem even harder for the particles since it is difficult to disambiguate their pose.

Figure 5.25 illustrates a successful experiment for different time steps. The particles' state is represented with orange dots and the robot true state is indicated with a blue arrow. After certain time steps the particles' state converges near the true robot state.

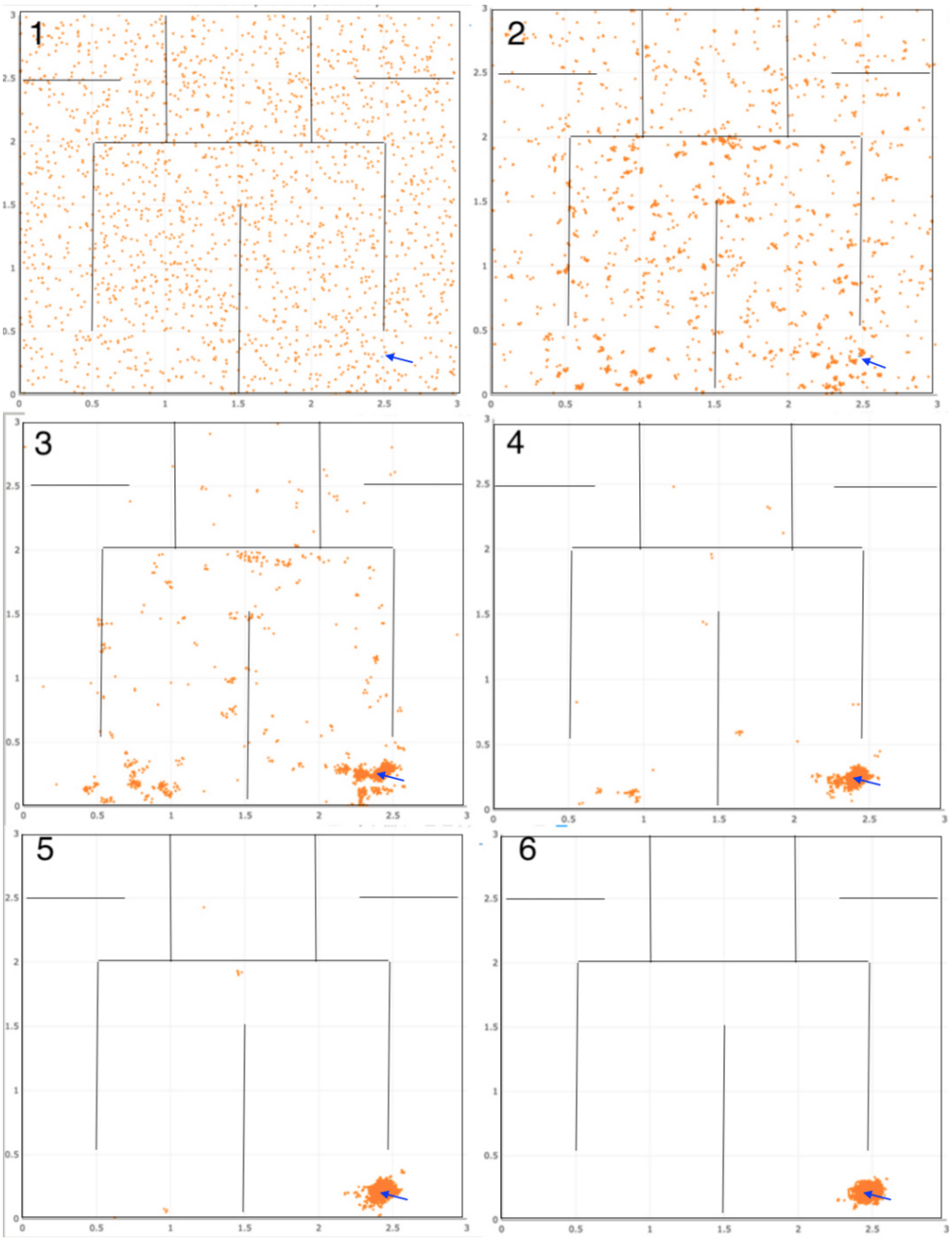


Figure 5.25: Global localization problem for the medium-high complex arena. The first image shows the particles' state initialization randomly over the state space, the blue arrow indicates the true robot state which was initialized in the (2.5, 0.3) x, y coordinates respectively. When the robot starts moving, the particles manage different hypotheses about the true robot state. After certain time steps, they converge to the true robot state as shown in the sixth image.

5.8 Summary

Several experiments have been conducted in order to see the performance of the proposed particle filter approach for robot localization.

The experiments and results are summarized in the following list:

- The particle filter parameters were empirically optimized. The values obtained are summarized in table 5.19.

Parameter	**Value**
Number of particles m	1000
Standard deviation for particles translation σ_{xy}	0.0025
Standard deviation for particles rotation σ_{θ}	0.5
Hyperparameter p	1

Table 5.19: Optimal values empirically found for particle filter.

- Resampling at each time step is a useful technique that drops the particles whose weight is near zero.
- Only three sensors were made available instead of eight, obtaining still good results when tracking the robot.
- The sensor noise was increased from 0.05 to 0.2. As a result, the robot tracking precision was minimally affected by this change.
- Odometry data is used by the particle filter as a reference to estimate the true robot state. Odometry depends highly on the robot's wheel diameter. In this experiment, the wheel's diameter was gradually incremented from 5cm to 6.5cm. In consequence, odometry is less and less reliable. The experiment shows that robot tracking is still very precise under the current circumstances.
- The robot walks through a predefined path in a medium-complexity and a medium-high complexity arena. At the end of the experiment the robot state estimate is close to the true robot state.
- Up until this point the particle filter algorithm dealt with the local localization problem; nevertheless, the difficulty in this last experiment is then incremented. The robot localized itself with no knowledge about its initial position. Consequently, the robot was under the global localization problem. The experiment was repeated ten times, each time the robot was initialized in a different position in the environment. In seven out of ten times the particles pose was condensed around the true robot state.

The proposed particle filter approach was successfully used to deal with the local and global localization problems. Thus, the robot was able to localize itself in most cases even under global uncertainty. Once the particle's state is condensed near a state far from the true robot state, it is not possible for the particles to recover.

Chapter 6

Conclusion

6.1 Main Results

The performed experiments show that the developed particle filter is effective to deal with both local and global localization in the created virtual environments, even if the robot has noisy sensors and limited number of them. The benefit of using machine learning on particle filter is that once the prediction model is trained, the update function for the particles' weight becomes trivial to implement compared to classical approaches.

Particle filter has one disadvantage against other non-parametric probabilistic models and that is they need a lot of computational resources when using a considerably large set of particles to be effective. Simple multi-layer neural networks are enough to deal with the sensor predictions when they are fed with enough data while training. When they are combined with the particle filter algorithm to address the local robot positioning problem, they need to make thousands of predictions per second when using a large set of particles, which is one of the main benefits of using neural networks: they are fast, at least when comparing them with other prediction models such as a random forest. Even though random forest is a powerful estimator, it needs to construct many trees to make an estimation which translates to more computational resources. Additionally, random forest is bounded by the CPU while neural networks can run in parallel on GPU.

Once the prediction model is trained, it is not updated anymore. Some experiments were performed in the following way: at each robot step a sensor measurement prediction is obtained given the true robot state, the predicted value is compared to the perceived data and the error is backpropagated so the neural network can "learn"; however, two problems were encountered with this approach: first, the true state is known in simulation only and second, the results were worse compared to the proposed particle filter without this technique but still better than using odometry only. The experiments did not present the expected relevance and therefore they were not included in this work. The developed plugin was used extensively for

understanding the influence of the parameters and checking the accuracy of the algorithm. It was released as an open-source plugin that can be used in academia to illustrate the particle filter algorithm.

Webots is a powerful tool for simulation. Their functionalities can be extended through the use of plugins, there is no limitation for the robots or worlds creation and all the experiments run smoothly; however, the worlds and robots are created by modifying a tree structure and its associated set of parameters. It would be more practical to have a graphical tool to do so as in the case of Gazebo. An extensive level of expertise in robotics is not required in order to use Webots. Additionally, its active community of developers is always glad to help.

6.2 Encountered Limitations

The success in the robot state prediction of the proposed particle filter algorithm depends mainly on three factors:

- The subsumption architecture used by the robot controller when collecting data (see section 3.2.2) is a key component. If the robot is not moved to a random location every 200 time steps, the collected data will be sequential and dependent on a single path. In consequence, the neural network will have trouble generalizing when training.
- The selected model to make the sensor prediction. A model that is inaccurate will make wrong predictions and thus the particles near the true robot state may be dropped when resampling.
- The noise introduced to the particles' state after a robot action is executed has resulted in a key component of the algorithm. Very large values disperse the particles' state making difficult to be near the true state. Very small values make the particles' state exploration very limited and thus once they lose track of the true state, the task of finding it again becomes impossible.

If one of these factors is missing or is faulty, the robot would be unable to localize itself.

A particle obtains a sensor measurement prediction given its state and a previously trained prediction model. This predicted sensor measurement is then compared to what the robot is perceiving in order to assign a weight to the particle that will determine if it is close or not to the true state. Here, the sensor measurements come from eight laser sensors but a problem arises when these sensor measurements are generated using another device such as a high resolution camera attached to the robot. Following the same principle as before, the prediction model should generate a high resolution image for that particle's state that should be compared to the image produced by the robot camera. The problem is that the prediction model might have

trouble generating a high resolution image from a simple state, the images cannot be compared using the same technique as in section 4.2.3 (predicted and perceived sensor measurements difference). If both problems are somehow addressed, the performance of the algorithm might be affected due to the amount of two-wise-image comparisons made at each time step for every particle. In consequence, the proposed particle filter approach is restricted to the kind of sensors used by the robot.

6.3 Future Work

- Improve the current approach: the use of a spatial view-embedding map of the environment as in Differentiable Mapping Network [63] together with sensor measurements might feed a prediction model to estimate the particle's likelihood, instead of a sensor measurement as in the current approach. A similar idea was first proposed in PF-nets by Karkus et al. [60] and it might solve the restriction of the proposed approach regarding the type of sensors used.
- Instead of using laser sensors, use a camera integrated to the robot (RGB, RGB-D, etc.) moving through visual robot localization. The benefit of doing so is that it is easier to integrate a camera to most of the robots (if they do not come with one already) rather than laser sensors (very few robots come with them). Thus the algorithm can be more generic.
- Create more complex predictive models. Moving to visual robot localization implies that the prediction models should deal with images. Therefore, the use of convolutional neural networks would be more appropriate.
- Collect synthetic data using a hybrid approach. First, make the set of virtual environments created by Webots dynamic by adding, for instance, movement to the objects. Second, make the robot walk randomly through the House3D data set [16] and the Webots dynamic environments using the proposed subsumption architecture while collecting data.
- The improved current approach would produce a prediction model which is independent of the type of environment. This model trained with the collected data will allow predictions, even if the robot is placed on previously unseen dynamic environments.
- Add semantic information to the maps such as labeling the objects, using computer vision techniques and combining this information to improve state estimation.
- Implement the improved approach in an end-to-end differentiable fashion. The benefit of doing so is that the prediction models can be updated and improved using the perceived data while localizing the robot.
- As a long-term goal, implement navigation techniques that are guided by voice commands such as *"Go to the kitchen and stand next to the table"*. Given a map of the environment, the robot should first localize itself. Next, use

recently developed techniques on the natural language processing area to get a semantic network representation of the voice command that can be executed directly by a computer program as robot actions. The work of Nevens et al. [75] for computational construction grammar for visual question answering might serve as a starting point for the semantic part. Finally, the robot can execute the necessary actions according to the semantic network to accomplish its goal.

Part III

Appendix

Appendix A

Appendix

A.1 Correctness of the Bayes Filter Algorithm

The current posterior distribution is calculated as shown in equation A.1.

$$p(x_t|z_{1:t}, u_{1:t}) = \frac{p(z_t|x_t, z_{1:t-1}, u_{1:t})\, p(x_t|z_{1:t-1}, u_{1:t})}{p(z_t|z_{1:t-1}, u_{1:t})} \tag{A.1}$$

where,

- $p(x_t|z_{1:t-1}, u_{1:t})$ is the prior distribution. That is the information of state x_t before seeing the observation at time t.
- $p(z_t|x_t, z_{1:t-1}, u_{1:t})$ is the likelihood model for the measurements. A causal, but noisy relationship[32].
- $p(z_t|z_{1:t-1}, u_{1:t})$ is the normalization constant defined as η

Thus equation A.1 can be summarized as follows:

$$p(x_t|z_{1:t}, u_{1:t}) = \eta\, p(z_t|x_t, z_{1:t-1}, u_{1:t})\, p(x_t|z_{1:t-1}, u_{1:t}) \tag{A.2}$$

The prediction of observation z_t based on the state x_t, the previous observation $z_{1:t-1}$ and the control action $u_{1:t}$ have conditional independence regarding the previous observation and the control action because they do not provide any information while predicting z_t. Thus the likelihood model for the measurements can be simplified as follows:

$$p(z_t|x_t, z_{1:t-1}, u_{1:t}) = p(z_t|x_t) \tag{A.3}$$

Therefore equation A.2 reduces to:

$$p(x_t|z_{1:t}, u_{1:t}) = \eta\, p(z_t|x_t)\, p(x_t|z_{1:t-1}, u_{1:t}) \tag{A.4}$$

In equation A.5 the prior distribution is expanded.

$$p(x_t|z_{1:t-1}, u_{1:t}) = \int p(x_t|x_{t-1}, u_t)\, p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) dx_{t-1} = \overline{bel}(x_t) \tag{A.5}$$

Replacing equation A.5 in A.4, equation A.6 is obtained which is calculated by the algorithm in list 1, third line.

$$p(x_t|z_{1:t}, u_{1:t}) = \eta\, p(z_t|x_t)\, \overline{bel}(x_t) \tag{A.6}$$

A.2 Behind the Scenes

The Python code that implements some important parts of the algorithm is illustrated here.

A.2.1 The Supervisor Function

```
robot = Supervisor()
robot_sup = robot.getFromDef("e-puck")
robot_trans = robot_sup.getField("translation")
robot_rotation = robot_sup.getField("rotation")

robot_trans.setSFVec3f([0.5, 0, 0.5])
robot_rotation.setSFRotation([0, 1, 0, 1.48])
robot_sup.resetPhysics()
```

A.2.2 Initialization of Particles

The Python implementation is shown on listing A.1. Notice that the particles are represented as an array containing separately the positioning, rotation states, together with the weights. Such values are themselves arrays of lengths equal to the number of particles. Even though this implementation could have been abstracted one level further and everything put into a class, arrays are easier to manipulate and transform, and therefore they present a flexibility advantage over the use of a class.

```
# weights or importance factors for M particles
self.weights = [1/self.number_of_particles] * self.
    number_of_particles

# initial state for all the particles
x = np.random.normal(robot_initial_config.x, self.sigma_xy,
     self.number_of_particles)
y = np.random.normal(robot_initial_config.y, self.sigma_xy,
     self.number_of_particles)
theta = np.random.normal(robot_initial_config.theta, self.
    sigma_theta, self.number_of_particles)

# define the set of particles as [x, y, theta, weights]
self.particles = np.array([x, y, theta, self.weights])
```

Listing A.1: Initialization of particles state on Python.

A.2.3 Movement of Particles

Listing A.2 shows the Python code for moving the particles. The variable `delta_robot_config` represents the control action that the robot executes. Notice how the clipping process for the angle differs from the clipping process for the positioning.

```
# get noise
noise_x = np.random.normal(self.mu, self.sigma_xy, self.
    number_of_particles)
noise_y = np.random.normal(self.mu, self.sigma_xy, self.
    number_of_particles)
noise_theta = np.random.normal(self.mu, self.sigma_theta,
    self.number_of_particles)

# move particles
particles[0] += delta_robot_config[0] + noise_x
particles[1] += delta_robot_config[1] + noise_y
particles[2] += delta_robot_config[2] + noise_theta

# clip the positioning to not be outside the arena or
    grater than 360 degrees on orientation
# X
particles[0] = np.clip(particles[0], 0, self.
    environment_config.environment_dim_x)
# Y
particles[1] = np.clip(particles[1], 0, self.
    environment_config.environment_dim_y)
# Theta
particles[2] = (360 + particles[2]) % 360
```

Listing A.2: Movement of particles.

A.2.4 Particles Weight Calculation

Listing A.3 shows the Python code for obtaining the particles' weight. Notice that the `particles_state` variable is an array containing every particles' state. Therefore the prediction of the sensor measurements for every particle is made at once instead of doing it particle by particle. This saves computational time. Notice also that when a sensor measurement is not present (for instance, due to hardware problems on the robot) the error is not calculated for that specific sensor.

```
def get_particles_weight(self, particles_state, sensors, p
    =3):
```

```
        """
        Given the particles state and robot sensor measurements it returns the weight of each particle.

        :param particles_state: [[x1,y1,theta1],[x2,y2,theta2],[x3,y4,theta4], ..., [x1000,y1000,theta1000]]
        :param sensors: [z1, z2, z3, z4,..., z8]
        :param p: hyper parameter
        :return: weights: [w1,w2,w3,..., w1000]
        """
        predicted_sensor_measurements = []

        # predict the sensor measurements (8 sensors) for every particle
        for i in range(1, 9):
            predicted_sensor_measurements.append(
                self.models[i-1].predict(particles_state))

        err = 0
        bad_data = True

        # calculate error: how far the predicted values are from the real sensor measurements
        for ix, elem in enumerate(predicted_sensor_measurements):
            if not math.isnan(sensors[ix]):
                bad_data = False
                err += np.absolute(elem - sensors[ix]) ** p

        # the bigger the error, the less the weight of the particle and thus the far from the real state
        weight = 1/err
        weight = weight.reshape((weight.shape[0],))

        return weight, bad_data
```

Listing A.3: Weight calculation.

The particles' weight should be normalized to contain a value greater than zero and less than one. The normalized weights should sum up to one to proceed with the resampling step. To normalize the weights the first step is to scale them so that the minimum value becomes one. The second step is to scale the previous values so that their sum becomes one. Listing A.4 illustrates such a process.

```
def normalize_weights(self):
    """
    Normalize weights to sum up to 1
```

```
4       """
5       # w_t = w_t/min(w_t)
6       self.particles[3] = self.particles[3] / self.particles
        [3].min()
7       # w_t = w_t/sum(w_t)
8       self.particles[3] = self.particles[3] / self.particles
        [3].sum()
```

Listing A.4: Initialization of particles state on Python.

A.2.5 Resampling

Listing A.5 shows the resampling process on Python. It uses the `random.choice` method of the `numpy` library where it samples with replacement from the `indexes` list with probability p equals the particles' weight. Finally, it replaces the list of particles with the ones selected randomly.

```
1 # resampling based on weights
2 if resampling:
3     indexes = range(0, self.number_of_particles)
4     indexes = np.random.choice(indexes,
5                                self.number_of_particles,
6                                p=self.particles[3],
7                                replace=True)
8     self.particles[0:3] = particle_transpose[indexes].
    transpose()
```

Listing A.5: Resampling on Python.

A.3 Installation and Configuration Guideline

This project is hosted on this GitHub link: `https://github.com/joangerard/webots-thesis`

It is about robot positioning estimation using:

- Webots robot simulator: Webots2020a
- Python 3.7
- Machine Learning Libraries for Python such that Tensorflow and Keras.
- Neural Networks
- Javascript, HTML and bootstrap
- Particles Filter

Youtube demo here: `https://www.youtube.com/watch?v=fISDutZca5g`.

The following folders can be found in this project:

- **documents**: it has the presentations made during the development process together with the project documentation under the `documennts/thesis-doc/main.pdf` file.
- **experiments**: the data associated with the experiments run.
- **webots-project**: Webots project containing the code used.

A.3.1 User Installation Guidelines

This section is focused for those who want to have the application running and are not interested in running the code.

1. Download Webots2020a from here `https://github.com/cyberbotics/webots/releases/tag/R2020a` and install it.
2. Download Python 3.7 from here `https://www.python.org/downloads/release/python-377/` and install it if you have not done it already. **DO NOT** USE `brew` command to install Python because you may have several permission problems while running it from Webots.
3. Clone this repository into your local machine typing `git clone https://github.com/joangerard/webots-thesis` in a terminal.

4. Install the Python library dependencies following these instructions:

   4.1. It is highly recommended to install a virtual environment in python to isolate the libraries' installation.

   4.2. In order to install a virtual environment please open a terminal and go to the recently cloned repository directory. Then go to the path `webots-project/controllers/pos-prediction` and install a virtual environment with this command.

   ```
   // For MacOs and Linux:
   python3 -m pip install --user virtualenv

   // For Windows:
   py -m pip install --user virtualenv
   ```

   4.3. Create a new environment called `env` (or whatever name fits for you) using the command:

   ```
   //For MacOs and Linux:
   python3 -m venv env

   //For Windows:
   py -m venv env
   ```

   4.4. Activate the environment: venv will create a virtual Python installation in the `env` folder.

   ```
   //For MacOs and Linux:
   source env/bin/activate

   //For Windows:
   .\env\Scripts\activate
   ```

   4.5. Confirm that your environment was correctly installed and it is active with the command:

   ```
   // For MacOS and Linux:
   which python
   .../env/bin/python <-- you should see this in your terminal

   // For Windows:
   where python
   .../env/bin/python.exe <-- you should see this in your terminal
   ```

   4.6. Now that you have your virtual environment up and it is activated install all the project dependencies on it.

   Go to the path `webots-project/controllers/pos-prediction` and run the command `pip install -r requirements.txt`.

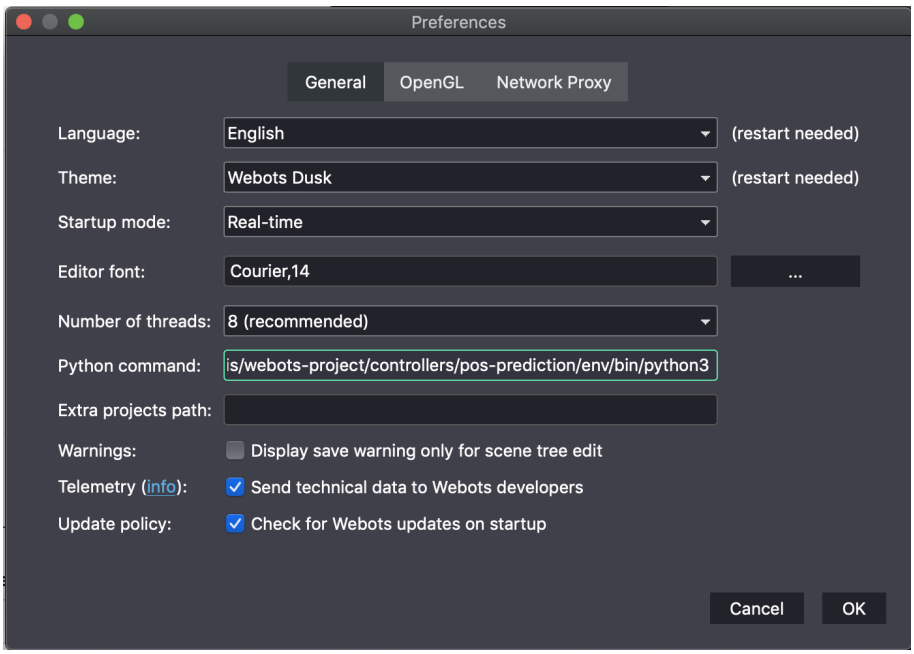
   This command will install all the project dependencies which are in the `requirements.txt` file.

5. Go to the folder `webots-project/worlds` you will see two worlds:

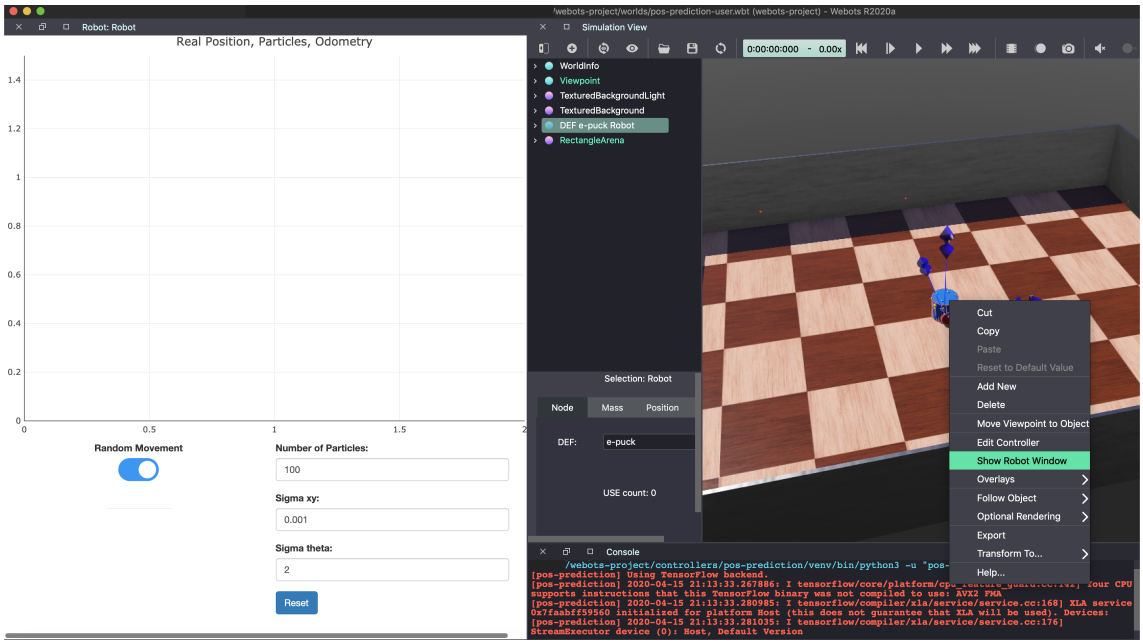
   - `pos-prediction-user.wbt`: This world is configured to run the robot controller inside the Webots tool.
   - `pos-prediction-dev.wbt`: This world is configured to run the robot controller outside the Webots tool using an IDE or Python Command lines, etc.

   In this section the file called `pos-prediction-user.wbt` will be used. Open it with a double click to launch the Webots application and the world will be loaded into it.

6. In Webots open the preferences window under the `Webots/Preferences...` option and configure the Python command to be the python executable which is in the recently created virtual environment in the path: `webots-project/controllers/pos-prediction/env/bin/python3`



7. Open the Robot Window: click on the robot then right-click on it and select the option Show Window Robot. The robot window will be displayed on the left side of the screen.



A.3.2 Developer Installation Guidelines

This section is dedicated to people who want to extend this plugin, reuse it or simply want to see how the code works.

IntelliJ Ultimate 2018.2 is used as IDE tool but any other IDE should be similarly configurable.

The robot controller code was programmed using Python 3.7 and it is under the directory `webots-project/controllers/pos-prediction`.

The robot window plugin was programmed using JavaScript 6 and it is under the directory `/webots-project/plugins/robot_windows/pos-prediction`.

Installation

1. Follow steps 1 to 4 from the User Guidelines section.

2. Configure the following environment variables in MacOs using the `export` command

Environment Variable	Value
WEBOTS_HOME	/Applications/Webots.app
DYLD_LIBRARY_PATH	add ${WEBOTS_HOME}/lib/controller
PYTHONPATH	add ${WEBOTS_HOME}/lib/controller/python37

For instance, you could add these lines to the `~/.bash_profile`:

```
export WEBOTS_HOME="/Applications/Webots.app"
export DYLD_LIBRARY_PATH="${WEBOTS_HOME}/lib/controller:$DYLD_LIBRARY_PATH"
export PYTHONPATH="${WEBOTS_HOME}/lib/controller/python37:$PYTHONPATH"
export PYTHONIOENCODING="UTF-8"
```

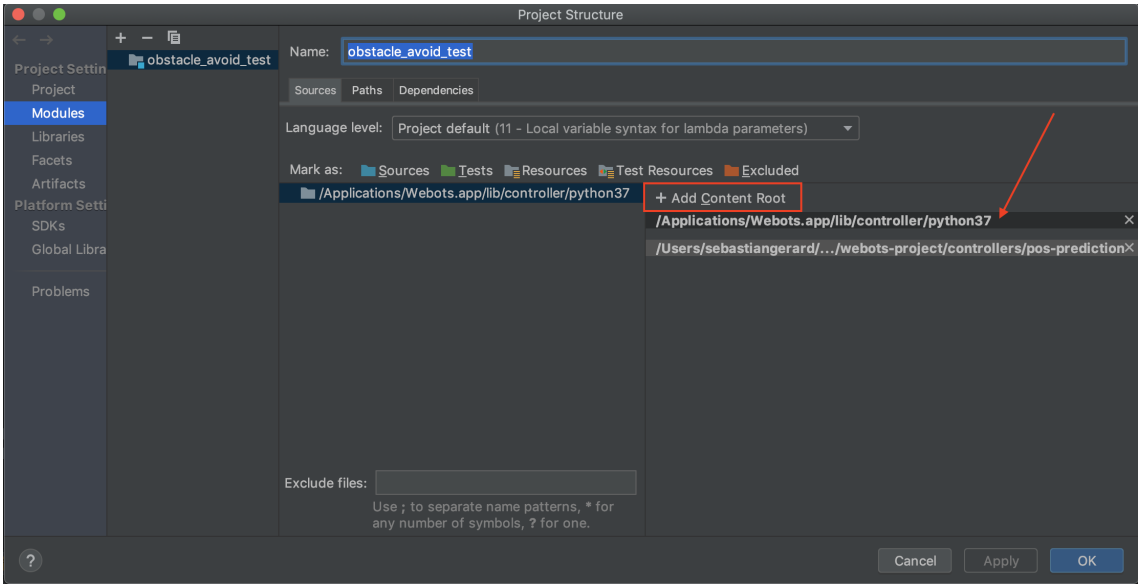
And execute the `source ~/.bash_profile` command.

For other operative systems please refer to this link: `https://cyberbotics.com/doc/guide/running-extern-robot-controllers?tab-os=macos&tab-language=python`

3. Configure IntelliJ to lunch the robot controller.

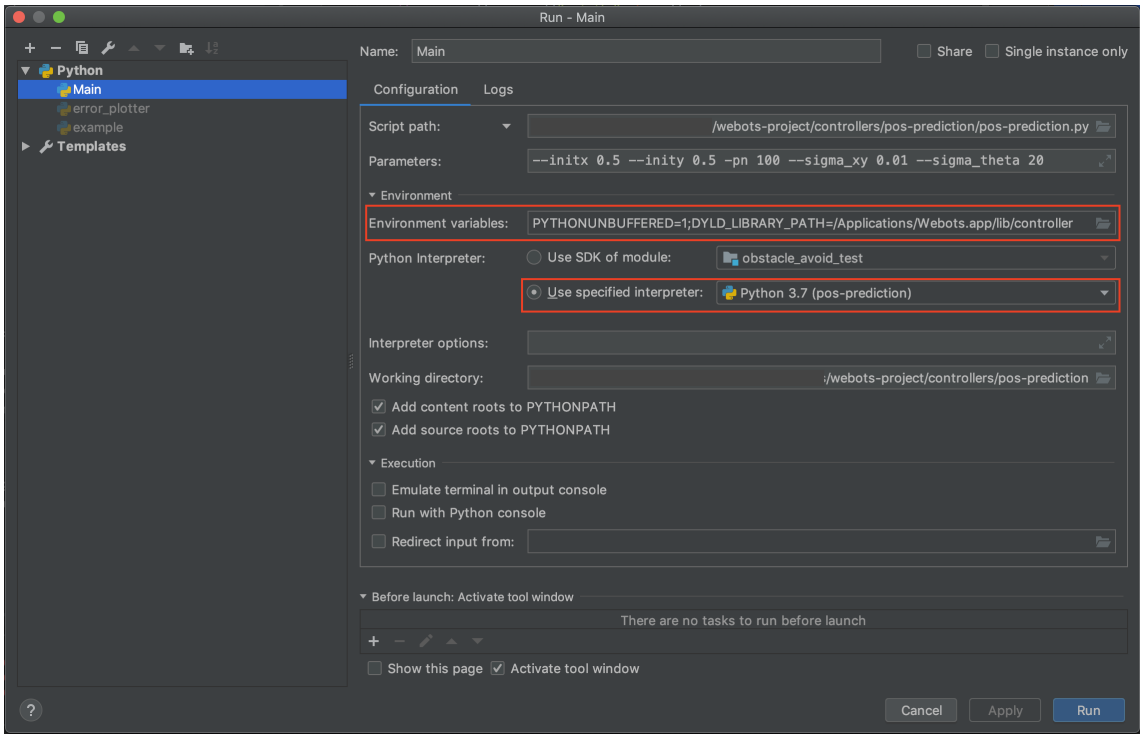
3.1. Open the `webots-project/controllers/pos-prediction` project on IntelliJ.

3.2. Click on `File/Project Structure...`, click on the `Add Content Root`, and select the Python version of Webots. In MacOs this can be found here: `Applications/Webots.app/lib/controller/python37`.



3.3. Click on `Run/Run.../Edit Configurations...`, in the environment variables add:

```
PYTHONUNBUFFERED=1;
DYLD_LIBRARY_PATH=/Applications/Webots.app/lib/controller
```



N.B. Make sure you selected the interpreter that is associated with your virtual environment. In the picture below, the virtual environment is called `pos-prediction` and it was created with the IDE selecting the option `File/Project Structure.../SDKs` and click on `+` sign to add another one. This is equivalent to the venv command previously used. It has the same purpose of isolating the installed libraries.

4. Open the `webots-project/worlds/pos-prediction-dev.wbt` using Webots(a right-click is usually enough) and click on the play button. This will do nothing but wait for an external controller to run the simulation.

5. Run the `pos-prediction.py` file on IntelliJ. You can simply press the `Run` button or right click on the file name and press `Run`.

6. You will see the simulation starts to run on Webots if everything was successful.

Command Line Arguments

It is possible to specify some arguments to pass to the controller before it runs, like the initial robot position or the number of particles, etc. In order to do so IntelliJ will be used to pass arguments each time we run the controller. This can be done through the Parameters field in the Running Configuration Window (Last image of step 2 in this section).

The list of arguments are listed below:

```
usage: pos-prediction.py [-h] [--initx INITX] [--inity INITY]
```

```
                        [--experiment_duration_steps EXPERIMENT_DURATION_STEPS]
                        [-pn PARTICLES_NUMBER] [--sigma_xy SIGMA_XY]
                        [--sigma_theta SIGMA_THETA] [--calculate_pred_error]
                        [--calculate_odo_error]
                        [--pred_error_file PRED_ERROR_FILE]
                        [--pred_odo_file PRED_ODO_FILE] [--go_straight]
                        [--capture_data]

optional arguments:
  -h, --help            show this help message and exit
  --initx INITX         Initial X position of robot
  --inity INITY         Initial Y position of robot
  --experiment_duration_steps EXPERIMENT_DURATION_STEPS
                        Max number of duration steps
  -pn PARTICLES_NUMBER, --particles_number PARTICLES_NUMBER
                        Number of particles
  --sigma_xy SIGMA_XY   Sigma XY that controls the SDE of the x,y coordinates
  --sigma_theta SIGMA_THETA
                        Sigma Theta that controls the SDE of the robot angle
  --calculate_pred_error
                        Store the prediction error in form of .pkl file
  --calculate_odo_error
                        Store the odometry error in form of .pkl file
  --pred_error_file PRED_ERROR_FILE
                        Name of the file where to save the prediction error
  --pred_odo_file PRED_ODO_FILE
                        Name of the file where to save the odometry error
  --go_straight         Let the robot go straight
  --capture_data        Capture data mode on
```

Examples of Arguments

This command tells the robot to start in position (0.5, 0.5) on the arena.

```
pos-prediction.py --initx 0.5 --inity 0.5
```

This command tells the robot to use 1000 particles with a standard deviation of 0.01 meters in the coordinates and 20 degrees in the angle.

```
pos-prediction.py --particles_number 1000 --sigma_xy 0.01 --sigma_theta 20
```

This command captures the prediction error at each time step and it saves it into a .pkl file. Then the array of values can be loaded from the file into a variable

using the Pickle Python Library. Each position of the array contains the value of $\sqrt{(\hat{x}_t - x_t)^2 + (\hat{y}_t - y)^2}$

```
pos-prediction.py --calculate_pred_error --pred_error_file
"data_30_partices_001_Sigma.pkl"
```

This was used for obtaining comparable results of the experiments.

Bibliography

[1] J. J. Leonard and H. F. Durrant-Whyte. "Mobile robot localization by tracking geometric beacons". In: *IEEE Transactions on Robotics and Automation* 7.3 (1991), pp. 376–382.

[2] Rodney Brooks. "New Approaches to Robotics". In: *Science (New York, N.Y.)* 253 (Oct. 1991), pp. 1227–32. DOI: 10.1126/science.253.5025.1227.

[3] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 2nd. Cambridge, MA, USA: MIT Press, 2018. ISBN: 9780262039246.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. 3th. USA: Prentice Hall, 2010. ISBN: 9780136042594.

[5] R. Siegwart et al. *Introduction to autonomous mobile robots*. 2011, pp. 265–366.

[6] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.

[7] João Filipe Ferreira and Jorge Dias. *Probabilistic Approaches to Robotic Perception*. Jan. 2014, pp. 3–36.

[8] L (Liqiang) Feng, Bart Everett, and J (Johann) Borenstein. *Where am I? : sensors and methods for autonomous mobile robot positioning*. Apr. 1996.

[9] Rodney A. Brooks. "The Role of Learning in Autonomous Robots". In: *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*. COLT '91. Santa Cruz, California, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 5–10. ISBN: 1558602135.

[10] Sebastian Raschka. *Python Machine Learning*. Packt Publishing, 2015.

[11] Sebastian Thrun. "Is Robotics Going Statistics? The Field of Probabilistic Robotics". In: *Communications of The ACM - CACM* (Jan. 2001).

[12] G. Borriello et al. "Bayesian Filtering for Location Estimation". In: *IEEE Pervasive Computing* 2.03 (July 2003), pp. 24–33. ISSN: 1536-1268.

[13] Leon Žlajpah. "Simulation in robotics". In: *Mathematics and Computers in Simulation* 79.4 (2008). 5th Vienna International Conference on Mathematical Modelling/Workshop on Scientific Computing in Electronic Engineering of the 2006 International Conference on Computational Science/Structural Dynamical Systems: Computational Aspects, pp. 879–897. ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2008.02.017. URL: http://www.sciencedirect.com/science/article/pii/S0378475408001183.

[14] Herman Bruyninckx. "Simulation, Modeling and Programming for Autonomous Robots: The Open Source Perspective". In: *Simulation, Modeling, and Programming for Autonomous Robots*. Ed. by Stefano Carpin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-89076-8.

[15] Josie Hughes, Masaru Shimizu, and Arnoud Visser. "A Review of Robot Rescue Simulation Platforms for Robotics Education". In: *RoboCup 2019: Robot World Cup XXIII*. Ed. by Stephan Chalup et al. Cham: Springer International Publishing, 2019, pp. 86–98. ISBN: 978-3-030-35699-6.

[16] Yi Wu et al. "Building Generalizable Agents with a Realistic and Rich 3D Environment". In: (Jan. 2018).

[17] Manolis Savva et al. "Habitat: A Platform for Embodied AI Research". In: *CoRR* abs/1904.01201 (2019). arXiv: 1904.01201. URL: http://arxiv.org/abs/1904.01201.

[18] Francesco Amigoni and Viola Schiaffonati. "Good Experimental Methodologies and Simulation in Autonomous Mobile Robotics". In: vol. 314. Sept. 2010, pp. 315–332. DOI: 10.1007/978-3-642-15223-8_18.

[19] George E. P. Box. "Science and Statistics". In: *Journal of the American Statistical Association* 71.356 (1976), pp. 791–799. DOI: 10.1080/01621459.1976.10480949. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1976.10480949. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1976.10480949.

[20] Dimitri P. Bertsekas and John N. Tsitsiklis. "Introduction to Probability". In: 2002.

[21] Arak Mathai and Hans Haubold. *Probability and Statistics: A Course for Physicists and Engineers*. Dec. 2017. ISBN: 978-3-11-056253-8. DOI: 10.1515/9783110562545.

[22] Gianluca Bontempi. *Statistical foundations of machine learning*. 2017.

[23] Harold Jeffreys. "An invariant form for the prior probability in estimation problems". In: *Royal Society* (1946).

[24] Wessel N. Van Wieringen A et al. *Rice,Mathematical Statistics and Data Analysis*. 1995.

[25] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[26] Gianluca Bontempi and Eythan Levy. *Cours de Modélisation et Simulation*. Université Libre de Bruxelles, Belgique.

[27] Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python*. 1st. USA: O'Reilly Media, 2016. ISBN: 978-1-449-36941-5.

[28] Gareth James et al. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN: 1461471370.

[29] Nikos Vlassis, B Terwijn, and B Krose. "Auxiliary Particle Filter Robot Localization from High-Dimensional Sensor Observations." In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Vol. 1. Feb. 2002, 7–12 vol.1. ISBN: 0-7803-7272-7.

[30] Reza Jazar. *Theory of applied robotics: Kinematics, dynamics, and control (2nd Edition)*. Jan. 2010, pp. 7, 17–20.

[31] G Cook. *Mobile robots: Navigation, control and remote sensing*. 2011, pp. 79–92.

[32] Simo Särkkä. *Bayesian Filtering and Smoothing*. Institute of Mathematical Statistics Textbooks. Cambridge University Press, 2013.

[33] N. Privault. *Understanding Markov chains: Examples and applications*. 2018, pp. 89–108.

[34] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: *Transactions of the ASME–Journal of Basic Engineering, 82 (Series D)* (Mar. 1960), pp. 35–45.

[35] Axel Barrau and Silvère Bonnabel. "Invariant Kalman Filtering". In: *Annual Review of Control, Robotics, and Autonomous Systems* 1 (May 2018).

[36] Matthew Rhudy, Roger A Salguero, and Keaton Holappa. "A Kalman Filtering Tutorial for Undergraduate Students". In: *International Journal of Computer Science & Engineering Survey* 08 (Feb. 2017), pp. 01–18.

[37] BD Chen et al. "Maximum Correntropy Kalman Filter". In: *Automatica* 76 (Sept. 2015).

[38] Paul Zarchan and Howard Musoff. *Fundamentals of Kalman Filtering: A Practical Approach (Third Edition)*. Progress in Astronautics and Aeronautics. American Institute of Aeronautics and Astronautics, Inc., 2009.

[39] Simon J. Julier and Jeffrey K. Uhlmann. "A New Extension of the Kalman Filter to Nonlinear Systems". In: *Proc. SPIE* 3068 (Feb. 1999).

[40] E. A. Wan and R. Van Der Merwe. "The unscented Kalman filter for nonlinear estimation". In: *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*. Oct. 2000, pp. 153–158.

[41] Axel Barrau and Silvère Bonnabel. "The Invariant Extended Kalman Filter as a Stable Observer". In: *IEEE Transactions on Automatic Control* 62 (Oct. 2014).

[42] Wolfram Burgard et al. "Estimating the Absolute Position of a Mobile Robot Using Position Probability Grids". In: *Proceedings of the National Conference on Artificial Intelligence* 2 (Mar. 2000).

[43] S. Thrun et al. "Robust Monte Carlo Localization for Mobile Robots". In: *Artificial Intelligence* 128.1-2 (2000), pp. 99–141.

[44] Ioannis Rekleitis. "A particle filter tutorial for mobile robot localization". In: *Statistics for Engineering and Information Science* (Jan. 2004).

[45] Sebastian Thrun. "Particle Filters in Robotics". In: *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*. UAI'02. Alberta, Canada: Morgan Kaufmann Publishers Inc., 2002, pp. 511–518. ISBN: 1-55860-897-4.

[46] M. S. Arulampalam et al. "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking". In: *IEEE Transactions on Signal Processing* 50.2 (2002), pp. 174–188.

[47] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". In: *IEE Proceedings F - Radar and Signal Processing* 140.2 (1993), pp. 107–113.

[48] Michael Pitt and Neil Shephard. "Filtering via Simulation: Auxiliary Particle Filters". In: *Journal of the American Statistical Association* 94 (Nov. 1997). DOI: 10.2307/2670179.

[49] Bradley P. Carlin, Nicholas G. Polson, and David S. Stoffer. "A Monte Carlo Approach to Nonnormal and Nonlinear State-Space Modeling". In: *Journal of the American Statistical Association* 87.418 (1992), pp. 493–500. DOI: 10.1080/01621459.1992.10475231. eprint: https://amstat.tandfonline.com/doi/pdf/10.1080/01621459.1992.10475231. URL: https://amstat.tandfonline.com/doi/abs/10.1080/01621459.1992.10475231.

[50] Christian Musso, Nadia Oudjane, and Francois Le Gland. "Improving Regularised Particle Filters". In: *Sequential Monte Carlo Methods in Practice*. Ed. by Arnaud Doucet, Nando de Freitas, and Neil Gordon. New York, NY: Springer New York, 2001, pp. 247–271. DOI: 10.1007/978-1-4757-3437-9_12. URL: https://doi.org/10.1007/978-1-4757-3437-9_12.

[51] Armin Burchardt, Tim Laue, and Thomas Röfer. "Optimizing Particle Filter Parameters for Self-localization". In: Jan. 2010, pp. 145–156. DOI: 10.1007/978-3-642-20217-9_13.

[52] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4. 1995, 1942–1948 vol.4.

[53] *Cyberbotics*. 2019. URL: https://cyberbotics.com.

[54] *Neural Networks, Types, and Functional Programming*. 2015. URL: http://colah.github.io/posts/2015-09-NN-Types-FP/.

[55] Francois Chollet. *Deep Learning with Python*. 1st. USA: Manning Publications Co., 2017. ISBN: 1617294438.

[56] Simone Scardapane. *Deep learning from a programmer's perspective (aka Differentiable Programming)*. https://towardsdatascience.com/deep-learning-from-a-programmers-perspective-aka-differentiable-programming-ec6e8d1b7c60. 2019.

[57] Fei Wang et al. "Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 10180–10191. URL: http://papers.nips.cc/paper/8221-backpropagation-with-callbacks-foundations-for-efficient-and-expressive-differentiable-programming.pdf.

[58] Haomiao Zhou et al. "A new sampling method in particle filter based on Pearson correlation coefficient". In: *Neurocomputing* 216 (July 2016).

[59] Rico Jonschkowski, Divyam Rastogi, and Oliver Brock. "Differentiable Particle Filters: End-to-End Learning with Algorithmic Priors". In: *CoRR* abs/1805.11122 (2018). arXiv: 1805.11122. URL: http://arxiv.org/abs/1805.11122.

[60] Peter Karkus, David Hsu, and Wee Sun Lee. *Particle Filter Networks with Application to Visual Localization*. 2018. arXiv: 1805.08975 [cs.RO].

[61] Rico Jonschkowski and Oliver Brock. "End-to-End Learnable Histogram Filters". In: 2017.

[62] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[63] Peter Karkus et al. "Differentiable Mapping Networks: Learning Structured Map Representations for Sparse Visual Localization". In: *International Conference on Robotics and Automation (ICRA)*. 2020.

[64] S. Eslami et al. "Neural scene representation and rendering". In: *Science* 360 (June 2018), pp. 1204–1210. DOI: 10.1126/science.aar6170.

[65] Dan Rosenbaum et al. "Learning models for visual 3D localization with implicit mapping". In: *ArXiv* abs/1807.03149 (2018).

[66] Piotr Mirowski et al. "Learning to Navigate in Cities Without a Map". In: (Mar. 2018).

[67] Xiao Ma et al. *Discriminative Particle Filter Reinforcement Learning for Complex Partial Observations*. 2020. arXiv: 2002.09884 [cs.LG].

[68] Maximilian Igl et al. "Deep Variational Reinforcement Learning for POMDPs". In: (June 2018).

[69] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[70] Fei Xia et al. "Gibson Env: Real-World Perception for Embodied Agents". In: *CoRR* abs/1808.10654 (2018). arXiv: 1808.10654. URL: http://arxiv.org/abs/1808.10654.

[71] Xiao Ma et al. "Particle Filter Recurrent Neural Networks". In: *CoRR* abs/1905.12885 (2019). arXiv: 1905.12885. URL: http://arxiv.org/abs/1905.12885.

[72] *IntelliJ IDEA*. https://www.jetbrains.com/es-es/idea/. [Online; accessed 23-April-2020]. 2020.

[73] *Cyberbotics' Robot Curriculum/Advanced Programming Exercises*. 2018. URL: https://en.wikibooks.org/wiki/Cyberbotics%5C%27_Robot_Curriculum/Advanced_Programming_Exercises.

[74] Daniel L Fulks, Michael K Staton, and Leonard J Kazmier. "Business statistics : based on Schaum's outline of theory and problems of business statistics". In: 2003.

[75] Jens Nevens, Paul Van Eecke, and Katrien Beuls. "Computational Construction Grammar for Visual Question Answering". eng. In: *Linguistics Vanguard* 5.1 (2019), pp. 1–16. ISSN: 2199-174X.