State estimation by interval analysis and probabilistic approaches: box particle filtering and localisation applications

Author:
M. Evandro Bernardes
Option: SPID Robotics
Promo: 2017

Supervisor:
Dr. Joaquim Blesa Izquierdo

Institut de Robòtica i Informàtica Industrial, CSIC-UPC
Parc Tecnològic de Barcelona. C/ Llorens i Artigas 4-6,
08028, Barcelona, Spain

August 29, 2017

Abstract — In this project, a box particle filtering algorithm is proposed and implemented. The algorithm combines two types of noise in measurements: stochastic noise affecting the measurements and bounded noise. The method is explained and compared extensively with the conventional particle filtering algorithm, both in a theoretical and in a practical way. Multiple different simulations are performed to further analyse the differences in performance between the conventional particle filter and the box particle filter, including a simple SLAM application.

Keywords: State estimation. Interval analysis. Particle filter, box particle filter, SLAM.

Résumé — Dans ce project, un algorithme de filtre particulaire par intervalles est proposé et implémenté. Cet algorithme prend en compte deux types différents de bruit: le bruit stochastique affectant les mesures et le bruit borné. La méthode est expliquée et comparée avec le filtre particulaire conventionnel. Plusieurs simulations différentes ont été réalisées pour mieux analyser les différences de performance entre le filtre conventionnel et l'implémentation du filtre par intervalles, ainsi qu'une simple application SLAM.

Mots-clé : Estimation d'état. Analyse par intervalles. Filtre particulaire, filtre particulaire par intervalles, SLAM.

Acknowledgements

I would like to thank Dr. Joaquim Izquierdo Blesa, who supervised my work during the whole project, giving me important advice whenever I needed, guiding me through understanding the necessary theory and helping me writing the memoir.

Thanks to Dr. Luc Jaulin, whose course on interval analysis has been crucial to the development of the project, besides having helped me finding the project at IRI.

Special thanks to ENSTA Bretagne and to the Institut de Robòtica i Informàtica Industrial, for providing me Matlab access and all the necessary environment for me to work on this project, and to the Consortium Ensicaen, for providing me a scholarship through Erasmus+ without which my stay in Barcelona would not have been possible.

I would also like to express my gratitude to all my friends who supported me during the project, specially to Joaquina Bustamante for her constructive remarks when reading the earlier versions of the manuscript.

Contents

1 Introduction **1**

2 Project objectives **3**

3 Theoretical concepts **5**

3.1 Problem definition 5

3.2 Bayesian filtering 6

3.3 Interval analysis 8

3.3.1 Set operations 8

3.3.2 Extended operations 9

3.3.3 Boxes (interval vectors) 9

3.3.4 Inclusion functions 10

3.4 Box Particle filter 10

4 Implementation **13**

4.1 Box particle filter 13

4.2 Details 17

4.2.1 Implementation of intervals 17

4.2.2 Landmarks 18

4.3 Obtained data and state estimation 19

5 Simulations and performance comparisons **23**

5.1 Simulations 23

5.1.1 Scenario 1 23

5.1.2 Scenario 2 35
5.2 Time performance 40
5.3 Number of boxes/particles 42
5.4 Simultaneous Localization and Mapping 48
5.4.1 Implementation 48
5.4.2 Results 49

6 Conclusion 51

A Implementation of Interval sine and cosine functions 53
A.1 Cosine function 53
A.2 Sine function 54

List of Figures

3.1 Dead reckoning example. a) In blue, the real path of the robot and in red, the estimated path. b) The error in the distance between the real and estimated paths. 6

3.2 Intersection operation. 9

3.3 Union operation. 9

4.1 Box particle filter example. Initial step. 14

4.2 Box particle filter example. Measurement update step. 15

4.3 Box particle filter example. State update step. 16

4.4 Box particle filter example. Resampling step. 17

4.5 Example of the measured probability distribution. 20

4.6 Example of the measured probability distribution (continuation). 21

5.1 First scenario used for the simulations, showing the path and landmark positions. 24

5.2 First test. Comparison between the real path and the computation of all the estimated path using only the knowledge of the system equations. . . . 25

5.3 First test. Comparison between the real path and the estimation provided by the conventional particle filter. 26

5.4 First test. Comparison between the real path and the estimation provided by the fixed array box particle filter. 27

5.5 First test. Comparison between the real path and the estimation provided by the variable array box particle filter. 28

5.6 First test. Comparison between the real path and the estimation provided by the variable array box particle filter. 29

5.7 Second test, conventional particle filtering. 30

5.8 Second test, fixed array box particle filtering. 31
5.9 Second test, variable array box particle filtering. 32
5.10 Second test, error plot. 33
5.11 Second test, mean error plot. 34
5.12 Second scenario. 35
5.13 Third test, fixed array box particle filtering. 36
5.14 Third test, variable array box particle filtering. 37
5.15 Third test, error plot. 38
5.16 Third test, mean error plot. 39
5.17 Time spent by the computer (in seconds) in each estimator during simulations 1 and 2. 40
5.18 Comparison for different number of number of boxes. Simulations with 2, 4, 8, 16, 32 and 64 boxes. 43
5.19 Comparison for different number of number of boxes. Simulations with 128, 256, 512, 1024, 2048 and 4096 boxes. 44
5.20 Error plot for the simulations with 2, 32 and 4096 boxes. 45
5.21 Error mean for all simulations. 46
5.22 Time spent calculating all simulations, in seconds. 47
5.23 SLAM simulation 1. Real map, estimated map and robot's trajectory. . . . 49
5.24 SLAM simulation 2. Real map, estimated map and robot's trajectory. The robot performs a 360 degrees rotation at each corner, in order to see the entire room. 50

Chapter 1

Introduction

Knowing how to correctly estimate the state of a given system is necessary to solve many problems, such as stabilization and control of dynamical systems, localization of an autonomous vehicle, fault diagnosis and obstacle avoidance. The current state can usually be used to calculate the next state when there is some knowledge of how the system works. For example, if an input signal is given to the system, it might be possible to predict how the system will evolve. In reality, the inputs, the system's response to the input and any other kind of real data, are prone to random errors, and relying only on the current step to predict the following one creates a series of estimations whose error might grow in time. In fact, the error at each step is the integration of the errors in all previous steps. To compensate for this, it is common to rely on sensors which can provide data that can, directly or indirectly, be used to calculate the system's current state more precisely (for example, the distance to a known landmark can be used to calculate the robot's position). These sensors are not exact either, therefore a method of combining all available data to produce the best possible estimation must be used. Bayesian inference methods such as Kalman filtering and particle filtering can be used to have a more reliable estimation of the state, if the probability distribution functions of the errors involved are known.

In this project, a robust implementation of the box particle filtering method described in Blesa et al. [2015] is created and compared with the conventional particle filtering.

First, in Chapter 2, the context of the project, place of development and objectives are precised. Second, in Chapter 3, the theoretical concepts necessary to understand the project are explained. Then, in Chapter 4, it is explained how the actual code was implemented on Matlab, and how all measured data are stored. Consequently, in Chapter 5, the results of various simulations using the box and conventional versions of the particle filter are shown in order to compare their performance and peculiarities. Finally, in Chapter 6, a quick conclusion on the results obtained during this project is presented.

Chapter 2

Project objectives

The project has been carried out at IRI (Institut de Robòtica i Informàtica Industrial) in Barcelona. The IRI is a Research Center of the Spanish Council for Scientific Research (CSIC) and the Technical University of Catalonia (UPC). According to the Institute's website, it has three main objectives: to promote fundamental research in Robotics and Applied Informatics, to cooperate with the community in industrial technological projects, and to offer scientific education through graduate courses.

The project is mainly based on the article Blesa et al. [2015] and has been done under the supervision of Joaquim Blesa Izquierdo. The project has started on 14 April 2017 and it is expected to end on 29 September 2017. Its main objectives are:

- Study Bayesian estimation and commonly used filter algorithms (the Kalman filter, the Extended Kalman filter and the particle filter);
- Study theoretical models of mobile robots;
- Study interval analysis;
- Implement the box particle filter as described in Blesa et al. [2015];
- Run simulations with a theoretical mobile robot model and test the algorithm's performance compared to other commonly used estimators;
- Implement a SLAM (Simultaneous Localization And Mapping) simulation using the box particle filtering algorithm as estimation engine.

Chapter 3

Theoretical concepts

3.1 Problem definition

The problem of the localisation of a mobile robot can be described by the following system of dynamic equations [Jaulin, 2011]:

$$\begin{cases} x(k+1) = f_k(\mathbf{x}(k), \mathbf{u}(k), \mathbf{v}(k)) & \text{(3.1a)} \\ \mathbf{y}(k) = h(\mathbf{x}(k)) + \mathbf{e}(k) & \text{(3.1b)} \end{cases}$$

for $k = 1, 2, ..., N_k$, being $\mathbf{x}$ the state of the robot, $\mathbf{u}(k) \in \mathbb{R}^{n_u}$ the system input, $\mathbf{y}(k) \in \mathbb{R}^{n_y}$ the measured output, $\mathbf{e}(k) \in \mathbb{R}^{n_u}$ a stochastic additive error specified by its known probability distribution function, $\mathbf{v}(k) \in \mathbb{R}^{n_x}$ the process noise.

Since both the state and input at step k are known, the evolution can be predicted using Equation 3.1a. This trivialises calculating the new state at step $k+1$ in a noiseless system. Nevertheless, when the random errors present in the system are taken into account, the subsequent state estimations get increasingly poorer. In fact, computing the state at a given step k for any system using only the inputs $u(k)$, the initial state and the function f is known as path integration. This kind of estimation is unstable and might produce curves which are close to the real path for small values of k (at the beginning) but presents big error values for higher values of k. Considering a discrete system, if $e_{state}(k)$ is the estimation's error resulting from step $k-1$ to k, the total error at a given step is going to be a sum of all those errors. An example of this type of estimation can be seen on Figure 3.1, showing how the error evolves.

To compensate this, sensors which can (directly or indirectly) give information about the current state are used in real applications, using Equation 3.1b. Taking the same example of the mobile robot, it is common to use distance sensors that can calculate the distance of the robot to one or more fixed and well-known locations called “landmarks”.

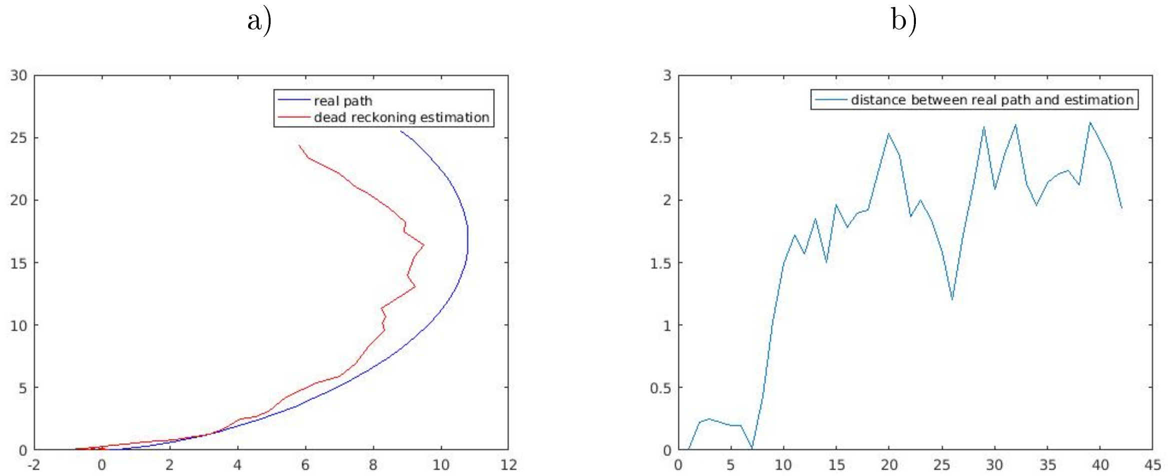

Figure 3.1: Dead reckoning example. a) In blue, the real path of the robot and in red, the estimated path. b) The error in the distance between the real and estimated paths.

Using only the distance to the landmarks to calculate where the robot is at a given time is the classical problem of trilateration.

The main objective of the estimation algorithms studied in this work is to use all the available data, both internal (the systems inputs) and external (the sensor readings), to efficiently compute the states of a given system.

3.2 Bayesian filtering

The estimation methods most commonly used to solve this kind of problems rely on filtering the data using methods of recursive Bayesian estimation. These methods are general probabilistic approaches that combine multiple available data, which makes it possible to compute a more precise estimation. Localisation problems often rely on recursively calculating a robot's position using knowledge of the system combined with new data being gathered by sensors (a thorough explanation can be seen in Sarkkä [2013]). This is the reason why recursive Bayesian estimation is extensively used in robotics.

When the model perfectly fits the real system, the measurement noise is uncorrelated and their distribution functions are known, the conventional Kalman filter is the optimal filter and gives the best possible solution. The conventional Kalman filter cannot be used with non-linear system functions though, so other methods of Bayesian filtering should be used when dealing with this kind of situations.

The most common state estimation algorithm applied for non-linear systems is the

Extended Kalman Filter (EKF). Although, if the non-linearities are severe, the EKF has troubles giving good estimates because it relies on linearisation. The Unscented Kalman Filter (UKF) presents improvements but it still is an approximate non-linear estimator [Simon, 2006]. The Particle filter, which is a completely non-linear estimator, will be described in the following section.

Particle filter

The particle filter is a particle Monte Carlo method used to solve filtering problems like the one studied in this work. It operates by applying multiple "particles" (points in a state space) that represent a probability density distribution. Filtering techniques are then applied on these particles, resulting in new particles that represent the posterior distribution. At the initial step, a first probability density is calculated randomly (and evenly) to work as the first prior distribution.

These particles are an approximation of the probability distribution by point weights at each time k, as shown in Equation 3.2.

$$p(x(k)) \sim \sum_{i=1}^{N} w(k)^i (x(k) - x(k)^i), \tag{3.2}$$

where each point $x(k)^i$ is a particle, and every particle has its corresponding positive weight $w(k)^i$. Moreover, the point weights are normalised so that $\sum_{i=1}^{N} w(k)^i = 1$.

A more profound analysis of the particle filter algorithm can be seen in Gustafsson et al. [2002]. The basic algorithm can be described by the following:

1. **Initialisation**: At this stage, the particles $x(0)^i$ are initialised, and $p(x(0)^i) = w(k-1)^i$, $k = 0$ and $i = 1, 2, ..., N$.

2. **Measurement update:** Computation of the posterior probabilities with the Bayes rule:
$$w(k)^i = \frac{1}{\alpha_k} P(y(k)|x(k)^i) w(k-1)^i, \tag{3.3}$$
while $\alpha_k = \sum_{i=1}^{N} P(x(k)^i|\mathbf{Y}(k))$ is the normalising constant.

3. **Resampling:** An optional step, where both the particles $x(k)^i$ and their corresponding weights $w(k)^i$ are replaced by $x'(k)^i$ and $w'(k)^i$. These new particles can have useful properties, and there are multiple ways to resample. In the particle filtering algorithm used during the simulations of this work, the algorithm called multinomial resampling has been used. This is a useful resampling technique that aims at preventing a problem that may arise when the estimated probability distribution becomes too narrow, eventually causing the particle filter to lose track of the state.

4. **State update**: Every particle from the new set is moved according to the state equations:
$$x(k)^i = f(x(k)^i, u_k, v_k) \tag{3.4}$$

5. **Repeat:** Do the same for the next step.

Since the particle filter is a Monte Carlo method, it does not give an optimal estimation (like the Kalman filter does for linear systems). Nevertheless, it still produces good approximations. One limitation of the algorithm is that it might require too much computer processing time when working with many dimensions.

3.3 Interval analysis

In this section, the main concepts of Interval Analysis used for the development of the box particle filtering algorithm will be introduced. Interval analysis is an approach in which errors (rounding errors or measurement errors, in the studied cases) are modelled as well defined bounds, making it possible for computers to easily perform numerical calculations.

Instead of working with probabilities and density functions, interval analysis is performed by applying a specially defined arithmetic directly to these bounds. In other words, if it can be declared with certainty that a quantity $[x]$[1] must be within the bounds $[\underline{x}, \overline{x}]$, then a quantity $[y] = f([x])$ is an interval $[\underline{y}, \overline{y}]$ which contains all the possible values of $f(x)$ for $x \in [x]$. Interval operations are a special case of set operations (as seen in Jaulin et al. [2001]), from which many of its fundamental properties are derived.

3.3.1 Set operations

The first and most basic kind of operations that can be performed on intervals are derived directly from set theory. Consider there are two sets, $\mathbb{X}$ and $\mathbb{Y}$. Some of the most common set operations are:

[1] The quantity is written inside brackets to denote it is an interval quantity. This is the notation used in this work.

- Intersection:
  $\mathbb{X} \cap \mathbb{Y} = \{x | x \in \mathbb{X} \text{ and } x \in \mathbb{Y}\}$

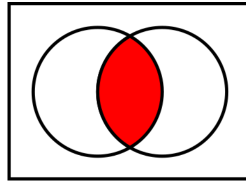

Figure 3.2: Intersection operation.

- Union:
  $\mathbb{X} \cup \mathbb{Y} = \{x | x \in \mathbb{X} \text{ or } x \in \mathbb{Y}\}$

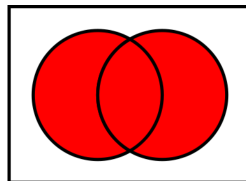

Figure 3.3: Union operation.

3.3.2 Extended operations

Besides the basic set operations, any kind of operation defined for numbers and vectors can be applied to intervals and interval vectors. To represent this with sets, taking a function f defined as $f(x, y) = x \diamond y$, the image of this function can be seen as the following generalization:

$$\mathbb{X} \diamond \mathbb{Y} = \{x \diamond y | x \in \mathbb{X} \, , \, y \in \mathbb{Y}\} \tag{3.5}$$

This can be used to define set addition, subtraction, etc., which are used to define the same operations on intervals. The particularities of the definitions of interval operations can be seen in Jaulin et al. [2001].

3.3.3 Boxes (interval vectors)

Given a real interval, denoted $[x]$, defined as a closed and connected subset of $\mathbb{R}$, a box $[\mathbf{x}]$ of $\mathbb{R}^{n_x}$ is defined as the product of n_x intervals. Boxes extend the concept of vector for intervals, and can be used to represent a region of space, instead of a point. For example, a two-dimensional box (or interval vector) can be used to represent an area, while a two-dimensional real vector would only represent a point in the same 2D space.

3.3.4 Inclusion functions

An inclusion function $[f]$ of a function f is defined so that the image of $[x]$ by $[f]$ is $[f]([x])$. This inclusion function is an interval extension to the function f and should be implemented in a way that the enclosing set is optimal. The value $[f]([x])$ must always enclose *at least* the image of the set $[x]$, but for poor implementations, the resulting interval can be much wider. Elementary arithmetical operations also have to be extended to the bounded error context.

To exemplify the differences between the same function being extended as two different inclusion functions, take the function $f(x) = x - x$. This function can obviously be simplified to wield the value $f(x) = 0$. Defining $[f_1]([x]) = [x] - [x]$ and $[f_2] = [0]$, using the value $[x] = [0, 2]$, two very different interval will be computed as a result:

- $[f_1]([0, 2]) = [-2, 2]$
- $[f_2]([0, 2]) = [0, 0]$

It can be observed that the point $x = 0$ is included in both intervals. The result computed by $[f_1]$ is not optimal though.

In real applications which use complicated equations, it might not be easy to see how to implement an optimal inclusion function. To solve this problem, contractors can be used to provide a better version of the inclusion $[f]$ using the original function f, as seen in Jaulin et al. [2001]. Contractors were not implemented in the algorithm used in this work.

3.4 Box Particle filter

In the last few years, an approach based on *box particles* has been proposed by Abdallah et al. [2008]. The Box Particle Filter handles box states and bounded errors. It uses interval analysis in the state update stage and constrained error techniques to perform the measurement update step. The set of box particles is interpreted as a mixture of uniform pdf's [Gning et al., 2010]. Using box particles has been shown to control quite efficiently the number of required particles, hence reducing the computational cost and providing good results in several experiments.

More recently, a new approach that takes into account the box particle filtering ideas but considering that measurements are tainted by stochastic noise instead of bounded noise have been proposed in Blesa et al. [2015]. The errors affecting the system dynamics are kept bounded because this type of uncertainty really corresponds to many practical situations, for example tolerances on parameter values.

The box particle filter is a state estimation technique that combines ideas from both the conventional particle filter and Interval analysis. The basic idea is to replace the particles $x(k)^i$ from the conventional particle filter by boxes with a finite but non-zero area. Analogous to the conventional particle filter, a probability is given for each box, creating a distribution of weights. This way, each point weight $w(k)^i$ represents now the probability that the real position is inside the ith box.

For the box particle filtering proposed in Blesa et al. [2015], the process noise $\mathbf{v} \in \mathbb{R}^{n_x}$ is considered bounded, where $v_i(k) \in [-\sigma_i, \sigma_i]$ for $i = 1, ..., n_x$. On the other hand, the measurement error $\mathbf{e}(k) \in \mathbb{R}^{n_y}$ is a stochastic additive error (specified by its known pdf p_e) that includes the measurement noise and the discretization error. As explained in Blesa et al. [2015], each state is considered as a set $\mathcal{X}(k)$, approximated by N_k boxes.

$$[\mathbf{x}(k)]^i, i = 1, ..., N_k \tag{3.6}$$

The width of each box must be equal or smaller than a given accuracy parameter δ_j for each component. Each box is given a probability *a priori* denoted as $P([\mathbf{x}(k)]^i|\mathbf{Y}(k-1))$. The objective is to calculate an estimation of the state evolution $\mathcal{X}(k+1)$, computing the corresponding $P([\mathbf{x}(k+1)]^i|\mathbf{Y}(k))$ probabilities.

Despite the fact that the structure of the box particle filtering algorithm is similar to the conventional particle filtering described above, there are substantial differences. The first difference that should be noted in the proposed box particle filtering algorithm is that instead of directly calculating the probability at a point, the integral[2] is computed for each $w(k)^i$, as seen in Equation 3.7.

$$w(k)^i = \frac{1}{\Lambda(k)} P([\mathbf{x}(k)]^i|\mathbf{Y}(k-1)) \int_{\mathbf{x}\in[\mathbf{x}(k)]^i} p_e(\mathbf{x}) d(\mathbf{x}), \tag{3.7}$$

Where $\Lambda(k)$ is a normalization constant used to assure that $\sum_{i=1}^{N_k} w(k)^i = 1$. The deduction of this integral is based upon the principle that the posterior probability of a box can be calculated by adding the weights of all particles inside the box when the number of particles tends to infinity. The proof is detailed in Blesa et al. [2015].

Another difference is that during the state update stage, the box particle filter considers all the process error realizations using interval analysis. For every box $[\mathbf{x}(k)]^i$ a new box is generated

$$[\mathbf{x}(k+1)|\mathbf{x}(k)]^i = [f]([\mathbf{x}(k)]^i, \mathbf{u}(k), [\mathbf{v}(k)]) \tag{3.8}$$

The new boxes $[\mathbf{x}(k+1)|\mathbf{x}(k)]^i$ inherit the weights $w(k)^i$ of their mother boxes $[\mathbf{x}(k)]^i$ $i = 1, \dots, N_k$.

[2] The integral of the probability function is calculated for the whole region in $\mathbb{R}^n_x$ defined by each box.

Finally, once the updated boxes $[\mathbf{x}(k+1)|\mathbf{x}(k)]^i$ and their associated weights $w(k)^i$ have been computed, a new set of disjoint boxes with the same size and their associated weights are computed. Optionally, the maximum number of boxes can be bounded eliminating the boxes with minimum weight if necessary.

Chapter 4

Implementation

In this chapter, the actual Matlab implementation will be explained and analysed.

4.1 Box particle filter

Two different algorithms were implemented:

- Dynamically creating boxes at each step: the box array at step k is used to calculate a new box array for step $k+1$. This way there are no bounds for the region where the robot can be found. This unbounded algorithm is the one used for the SLAM simulation, since no prior knowledge of the environment is supposed.
- Instead of letting the code create an undefined number of boxes of different sizes, a fixed array of equal sized boxes has been used, representing the whole domain of possible states [1].

Both versions of the algorithm use arrays of fixed size boxes, and they usually do not show a big difference in performance when the estimated path does not approach the limits of the environment.

The developed algorithm can be divided into the following steps:

1. **Initialisation:** An array w is used to store the weights of each box at a given time, representing the weight distribution. If the first step is being executed, this array must be initialised either with a uniform distribution (same weight for every box) or

[1] This choice is particularly useful in our case of the localization of a mobile robot, where it can represent the room where the robot is located (assuming there is a bounded area from which the robot cannot escape).

with a known distribution[2]. On Figure 4.1, the initial boxes representing the initial distribution are shown.

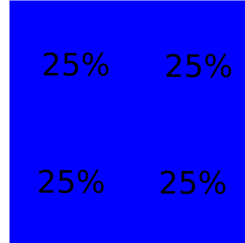

Figure 4.1: Box particle filter example. Initial step.

2. **Measurement update:** In this step, the weight of each box is updated. Using the prior probability $P([\mathbf{x}(k)]^i|\mathbf{Y}(k-1))$ and measurement probability distribution functions p_e, the posterior probability $P([\mathbf{x}(k)]^i|\mathbf{Y}(k))$ can be calculated. The pseudocode for this part can be seen in Algorithm 1.

Algorithm 1 Box particle filter measurement update

```
for all i, w[i] ≠ 0 do
    L = 1
    for j = 1, 2, 3...N of landmarks do
        L ← L × ∫_{x∈[x(k)]^i} p_e^j(x) d(x)
    end for
    w[i] ← w[i] × L
end for
Λ ← Σ_{i=1}^{N_k} w[i]
for i = 1, 2, 3, ..., N_k do
    w[i] ← w[i] ÷ Λ
end for
```

On Figure 4.2, an example of the measurement update is shown.

3. **State update:** This step is analogous to the state update in the conventional particle filtering algorithm. Using interval analysis, an interval extension $[f]$ of the state function f defined in Equation 3.1a is used to give an estimation of the next step, as shown in Equation 3.8.

   On Figure 4.3, an example of the state update is shown.

[2] In case the initial location of the robot is known, for example.

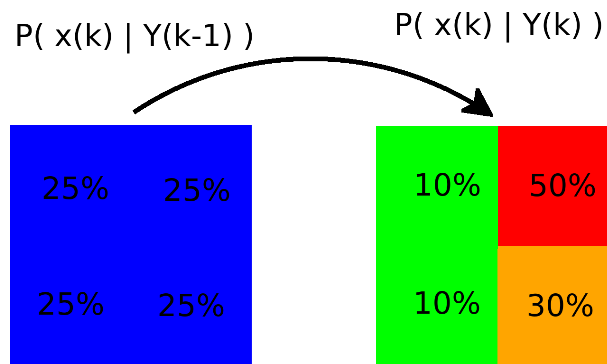

Figure 4.2: Box particle filter example. Measurement update step.

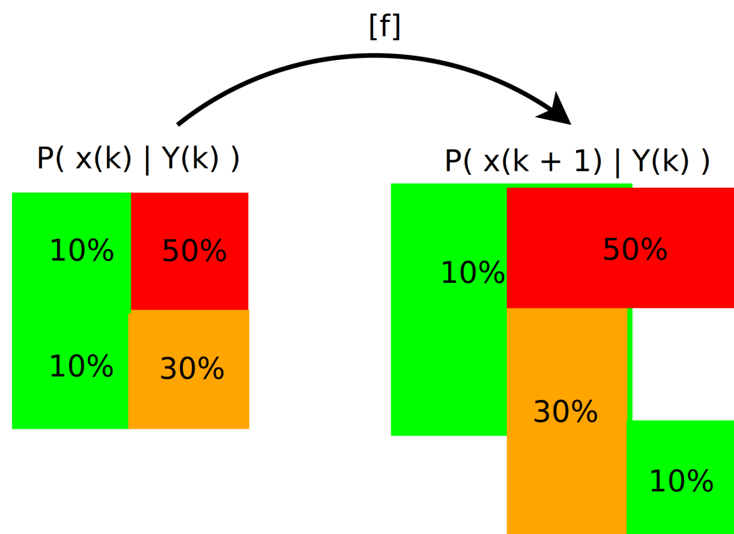

Figure 4.3: Box particle filter example. State update step.

4. **Resampling and repartitioning:** The boxes resulting as the output of the system interval function $[f]$ are proportionally distributed over the existing boxes: Each box $[\mathbf{x}(k)]^i$ lying in the region receives a weight value proportional to the interval intersection between itself and the output box:

$$w(k+1)^i = \sum_{i=1}^{N_k} \left(\sum_{j=1}^{N_k} \left(\frac{A([\mathbf{x}(k+1)]^j \cap [f](\mathbf{x}(k), \mathbf{u}(k)))}{A([f](\mathbf{x}(k), \mathbf{u}(k)))} \right) w(k)^i \right) \tag{4.1}$$

Where $A([\mathbf{x}])$ is the area of the box $[\mathbf{x}]$ (in the 2D case). The new weight distribution is normalised at the end. The pseudocode is shown in Algorithm 2.

On Figure 4.4, the resampling step is represented.

Algorithm 2 Box particle filter resampling

for all $i, w^k[i] \neq 0$ **do**
  $F = [f](\text{Boxes}^k[i])$
  for $j, \text{Boxes}^{k+1}[j] \cap F \neq \emptyset$ **do**
    $A \leftarrow Area(\text{Boxes}^{k+1}[j] \cap F)$
    $w^{k+1}[j] \leftarrow w^{k+1}[j] + A \times w^k[i]$
  end for
end for

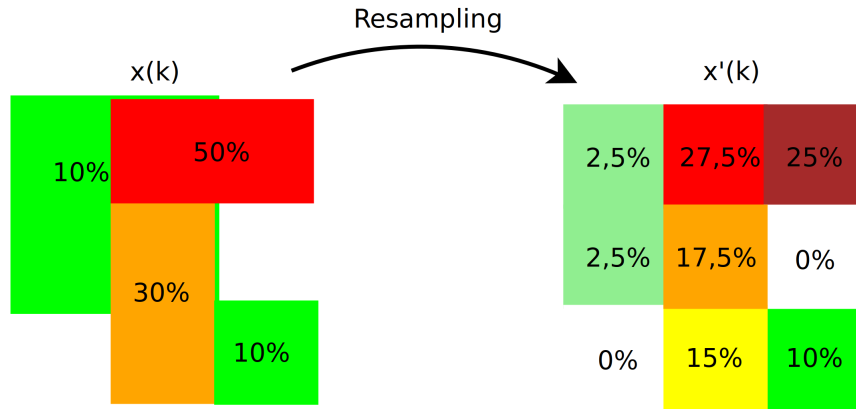

Figure 4.4: Box particle filter example. Resampling step.

It is also important to note that, for the fixed box array version of the algorithm, cases where the resulting boxes $[f](\mathbf{x}(k), \mathbf{u}(k))$ are outside the limits of the domain must be treated separately. The algorithm implements this by defining a big box $[\mathbf{X}_{total}]$ (representing the whole area) and by always checking if the intersection $[f](\mathbf{x}(k), \mathbf{u}(k)) \cap [\mathbf{X}_{total}]$ is empty. If this intersection is empty, the whole weight distribution is assumed to be concentrated in the limit boxes. This has positive effect in the accuracy of estimations computed in this closed-area simulations, since all outliers are being corrected by discarding any possibility of the position being outside of $[\mathbf{X}_{total}]$.

5. **Repeat:** The whole process is repeated for $k = k + 1$ until the whole simulation is completed.

One of the choices made during the implementation is that it was decide to discard boxes whose weights are too small (below a certain threshold). This can lead to a slightly quicker algorithm, but it can also lead to the problem of sample depletion. This problem arises when the density distribution is too narrow (only a few boxes remaining), which might lead to the estimator losing track of the robot's position.

4.2 Details

4.2.1 Implementation of intervals

Interval computation is a crucial part of the algorithm, since the possible positions of the mobile robot are defined as an array of boxes (interval vectors), and its corresponding properties and operations are exploited by the algorithm as seen in Chapter 3, mainly during the state update and resampling stages.

Since no contractors or separators have been used in this algorithm for now, no external interval library has been used and an Interval Matlab class has been created, defining not only intervals and interval vectors, but also their corresponding operations and useful functions. The class definition can be seen in the file *Interval.m*[3]. The simplest way to define a new interval is:

```
1 % creates the Interval x = [a,b]
2 x = Interval([a,b]);
```

Listing 4.1: Defining a new Interval value

The class has been implemented in a way that usual operations with integer, double, etc. values can also be applied in Interval variables in a natural way, overloading the operators. This makes it possible to easily implement new robot model equations.

```
1 x = 5;
2 y = Interval([1,2]);
3 z = x + y;
4 % z is an Interval variable of value = [6,7]
5 w = 2*(x + y/2);
6 % w is an Interval variable of value = [11,12]
```

Listing 4.2: Operations with Intervals

The class also takes advantage of Matlab's use of arrays and tries to treat Interval vectors (boxes) the same way it would treat a common numerical vector.

```
1 x = Interval([0,2],[0,2]);
2 % x is an Interval vector representing the region [0,2]x[0,2]
```

Listing 4.3: Interval vectors

[3]Only the mathematical functions needed by the used model were implemented, like the basic trigonometric functions. An explanation of the sine and cosine functions definitions can be seen in Appendix A.

4.2.2 Landmarks

Each observation is modelled with a probability density function that must be integrated during the measurement update stage. In this work's implementation, a Matlab cell array has been used to store each probability function, and the *measurementUpdate.m* script has been programmed to receive only cell arrays as input.

This is introduced in the program with the variable *pe*, a cell array containing lambda functions of 3 variables: the position (x, y) and the measure at instant k. In the case where only one function is used, it still has to be a cell array of size 1. At each iteration, this array is updated, since the probability function changes: the pde's variance is known and do not change, but the mean is updated for every measurement, offsetting the probability function to make the measured value the most probable value. This creates a new cell array, called *pek* in the main script, whose values are only functions of (x, y). These final two-variable density functions array is the one used as input for the *measurementUpdate.m* script.

4.3 Obtained data and state estimation

At each step of the simulation, the output data obtained by the algorithm are two arrays:

- *Boxes* array: to store the exact limits of each one of the defined boxes.
- w (or "weight") array: to store the probability density of the state in each box. For example, the value $w(i, j)$[4] stores the probability of the correct state being inside the box $Boxes(i, j)$.

To compute the estimated state value from these data, Equation 4.2 was used.

$$\hat{x}_{i,j} = \sum_{i=1}^{N} \sum_{j=1}^{M} w(i, j) med(Boxes(i, j)) \tag{4.2}$$

Where $med([\mathbf{x}])$ is the mean point of box $[\mathbf{x}]$, given by $med([\mathbf{x}]) = \underline{x} + width([\mathbf{x}])/2$.

On Figures 4.5 and 4.6, the measured probability distribution, the real position and the estimated position calculated with Equation 4.2 can be seen for two different steps of the same simulation.

[4] i and j are used because this example is in the 2D space.

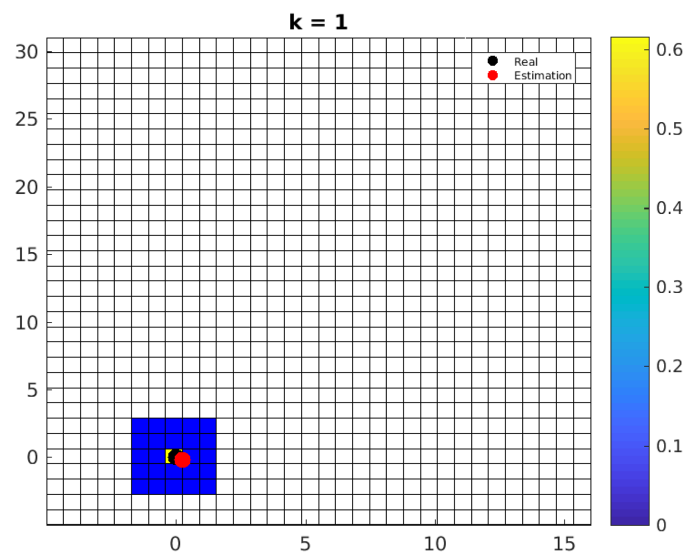

Figure 4.5: Example of the measured probability distribution.

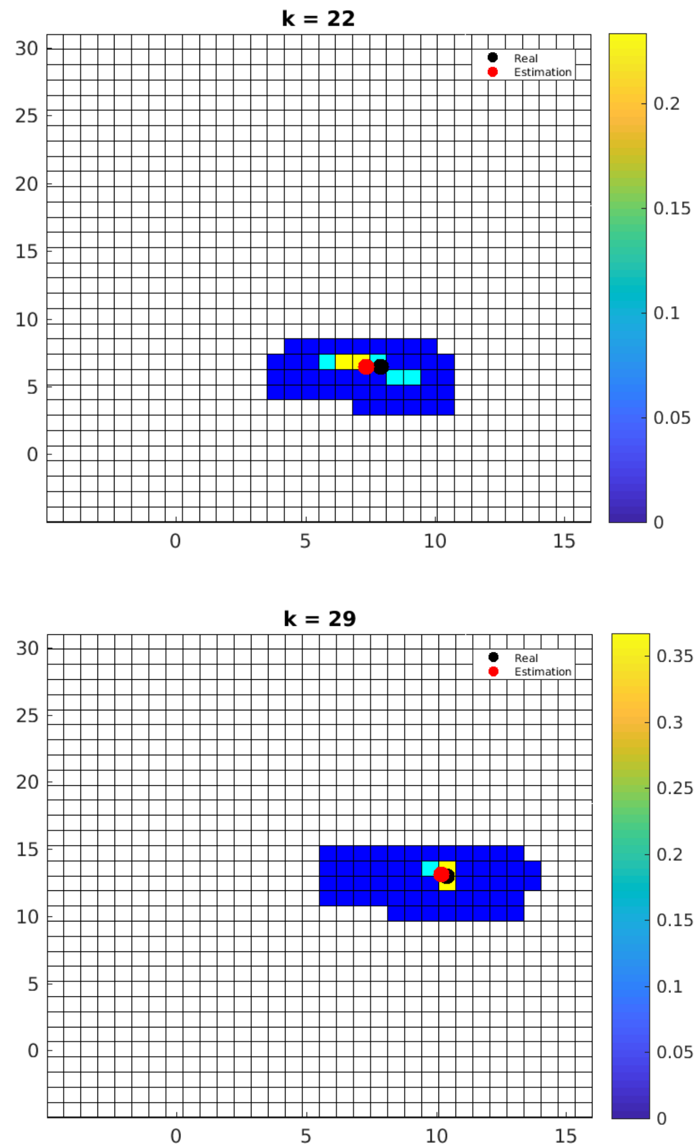

Figure 4.6: Example of the measured probability distribution (continuation).

Chapter 5

Simulations and performance comparisons

In this chapter, some simulations comparing the performance of the box particle filter algorithms with the conventional particle filter will be shown. Differences in precision, shape of the estimated path and running time of the Matlab codes will be compared. For all simulations, a sampling time of $t_s = 0.5$ was used. In addition to that, the equations of the robot's state is given by $(\mathbf{x}, v, \theta)$, representing the robot's position, speed and angle. The state equations of the theoretical model are the following:

$$\begin{cases} x_1(k+1) = x_1(k) + t_s v \cos(\theta) \\ x_2(k+1) = x_2(k) + t_s v \sin(\theta) \end{cases} \tag{5.1}$$

5.1 Simulations

5.1.1 Scenario 1

The first scenario that will be presented is the path shown on Figure 5.1. In addition to that, the measurement equations are:

$$\begin{cases} y_1 = v + e_v \\ y_2 = \theta + e_\theta \\ y_3 = |\mathbf{x} - \mathbf{m_1}| + e^1_{distance} \\ y_4 = |\mathbf{x} - \mathbf{m_2}| + e^2_{distance} \\ y_5 = |\mathbf{x} - \mathbf{m_3}| + e^3_{distance} \end{cases} \tag{5.2}$$

Where the first two equations represent the robots internal speed and angle sensors, the third, fourth and fifth equations represent the measurements of the distance sensor to each

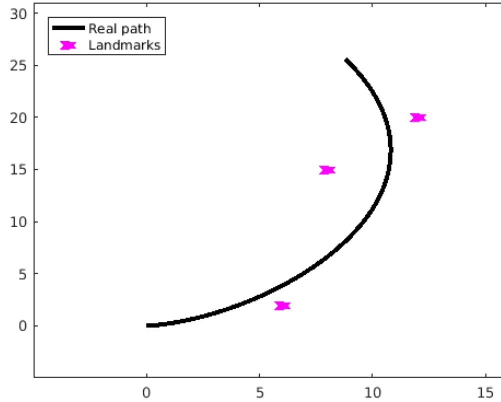

Figure 5.1: First scenario used for the simulations, showing the path and landmark positions.

one of the sensors $\mathbf{m_1}$, $\mathbf{m_2}$ and $\mathbf{m_3}$, and e_v, e_θ and $e_{distance}$ represent the errors of the speed, angle and distance sensors, respectively.

Test 1

The first test was done with the following variance values for the density functions:

$$\begin{cases} \sigma^2 = 0.500 \\ \sigma_v^2 = 0.500 \\ \sigma_\theta^2 = 0.500 \end{cases} \tag{5.3}$$

Being σ^2, σ_v^2 and σ_θ^2 the error variances from the distance, speed and angle sensors, respectively. The values for the number of particles and boxes for the conventional and fixed array particle filtering are:

- Boxes width $= [0.913043, 0.913043]$
- Number of particles/boxes $= 512$

The same box dimensions used for the fixed array algorithm were used in the variable array algorithm, in order to compare both box algorithms using the same accuracy. The estimated paths can be seen on Figures 5.2, 5.3 and 5.4.

It can be seen that all three estimators were able to calculate paths that are quite close to the real path. To better compare them, the error plot of all three estimators are shown on Figure 5.6, also comparing them with the error measured by using only the system

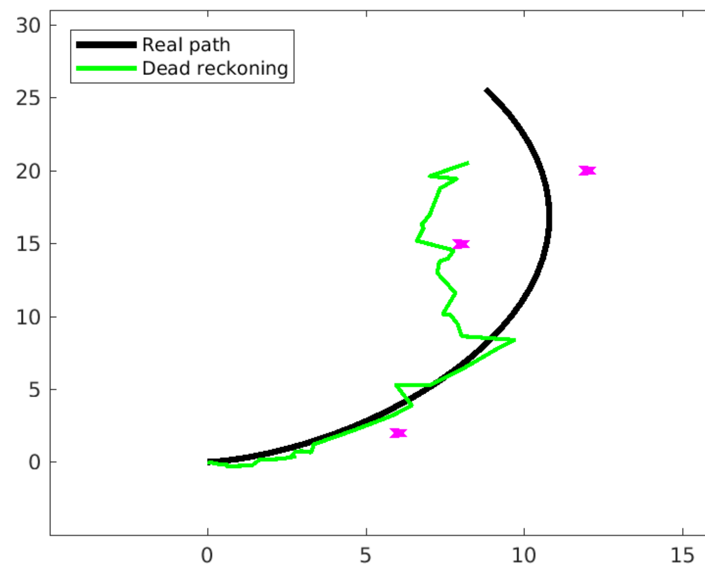

Figure 5.2: First test. Comparison between the real path and the computation of all the estimated path using only the knowledge of the system equations.

equations without the filtering (dead reckoning). It can be seen that all three estimators give low errors. The conventional particle filter gives a good estimation, with the error plot showing a similar performance to both box particle filtering applications. Moreover, since the particle filter is a Monte Carlo method, big errors can be observed at the beginning even for the small values of variance used (as shown in Equation 5.3). Another important thing to notice is that the estimators maintain an almost constant error, while simply using the system equations and the inputs makes the estimation behave exactly as expected: the error starts small at the beginning but becomes increasingly bigger.

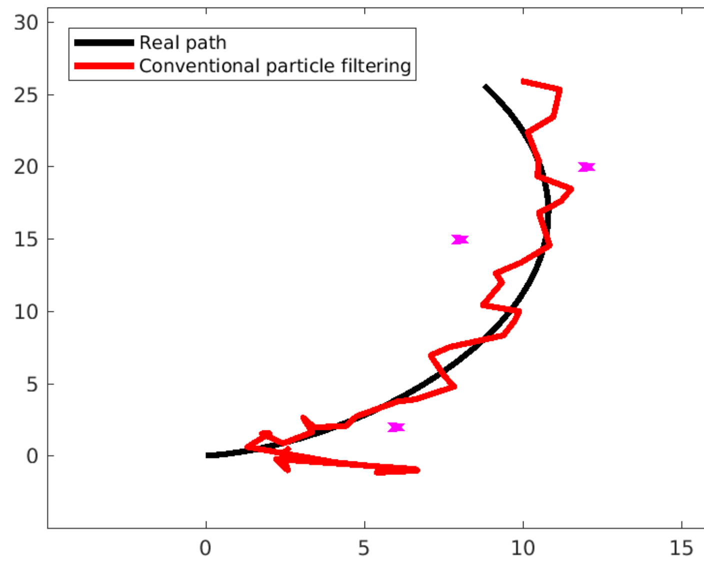

Figure 5.3: First test. Comparison between the real path and the estimation provided by the conventional particle filter.

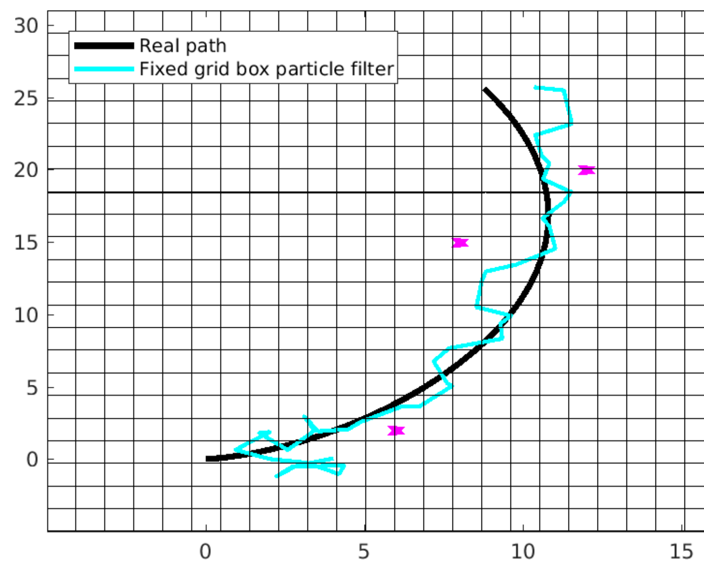

Figure 5.4: First test. Comparison between the real path and the estimation provided by the fixed array box particle filter.

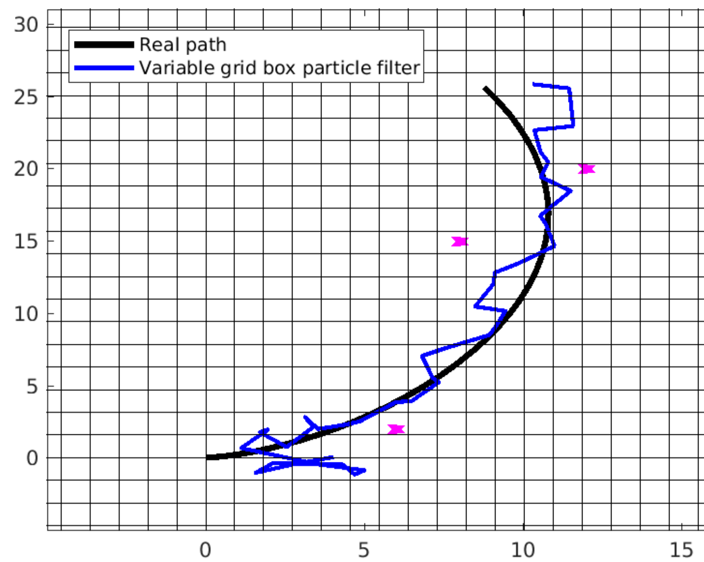

Figure 5.5: First test. Comparison between the real path and the estimation provided by the variable array box particle filter.

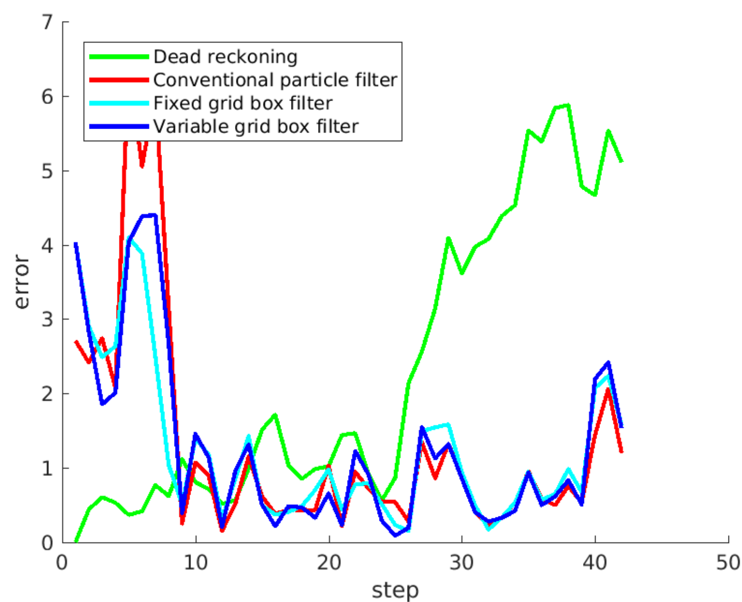

Figure 5.6: First test. Comparison between the real path and the estimation provided by the variable array box particle filter.

Test 2

The first test was done with rather small variance values. To further analyse the estimators performances, a new simulation was carried out with the following values:

$$\begin{cases} \sigma^2 = 0.050 \\ \sigma_v^2 = 4.000 \\ \sigma_\theta^2 = 1.000 \end{cases} \tag{5.4}$$

In this new test, the values of the velocity and angle measurement variances are bigger.

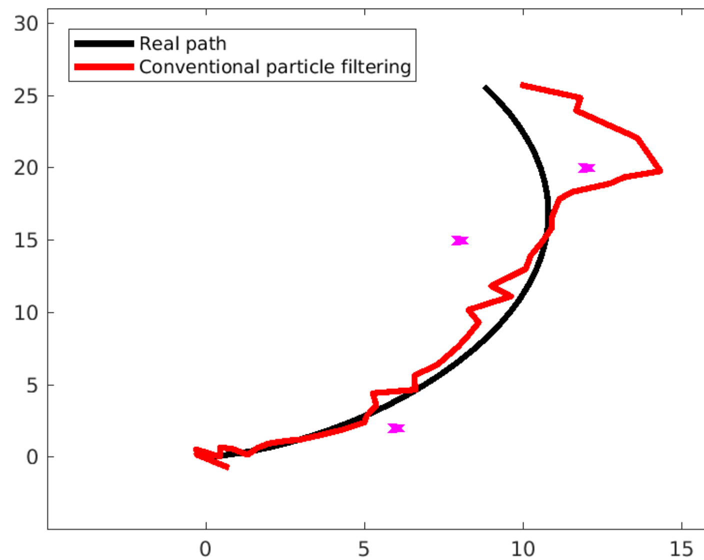

Figure 5.7: Second test, conventional particle filtering.

On Figure 5.10, it is possible to see how both box particle filtering algorithms outperform the conventional particle filter, giving better estimations. The conventional particle filter needs some time to converge, while the box particle filters produce consistent estimations during the whole simulation. In the first test, this only affected the beginning of the simulation (all estimators having a similar performance afterwards). In the second test errors can be bigger, because the conventional particle filtering estimator might have the same problem in having to converge again if ever a big error is observed. The mean error can be calculated using Equation 5.5, where $\mathbf{x}(k)$ is the real position on step k, $\hat{\mathbf{x}}(k)$ is the estimated position on step k and $|\cdot|$ is the norm operator. It can be seen on Figure 5.11 how the mean error values are small for the box particle filtering estimators.

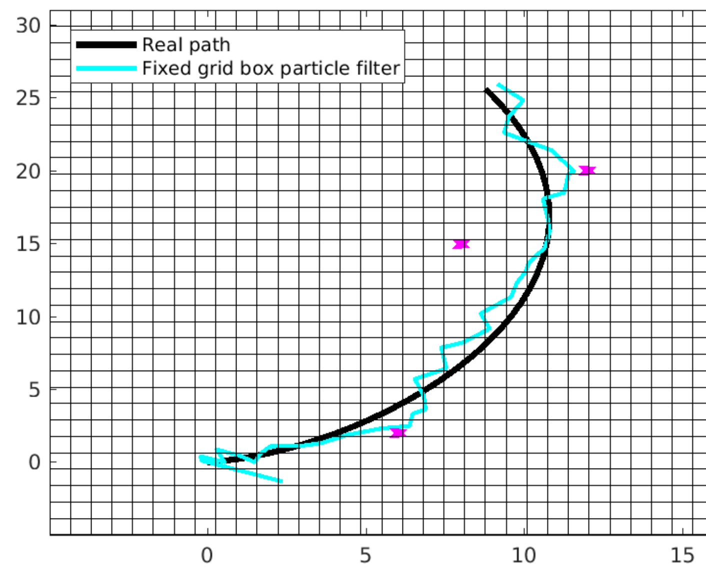

Figure 5.8: Second test, fixed array box particle filtering.

$$e_{mean} = \sqrt{\sum_{k=1}^{N} |\mathbf{x}(k) - \hat{\mathbf{x}}(k)|^2} \tag{5.5}$$

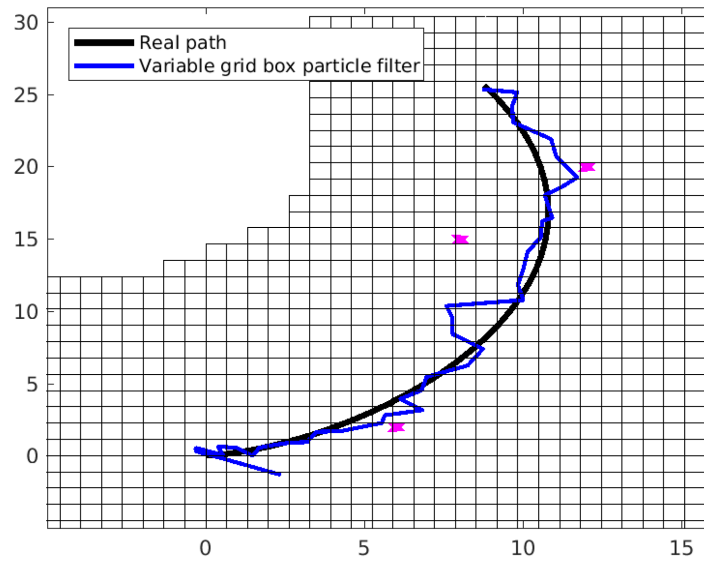

Figure 5.9: Second test, variable array box particle filtering.

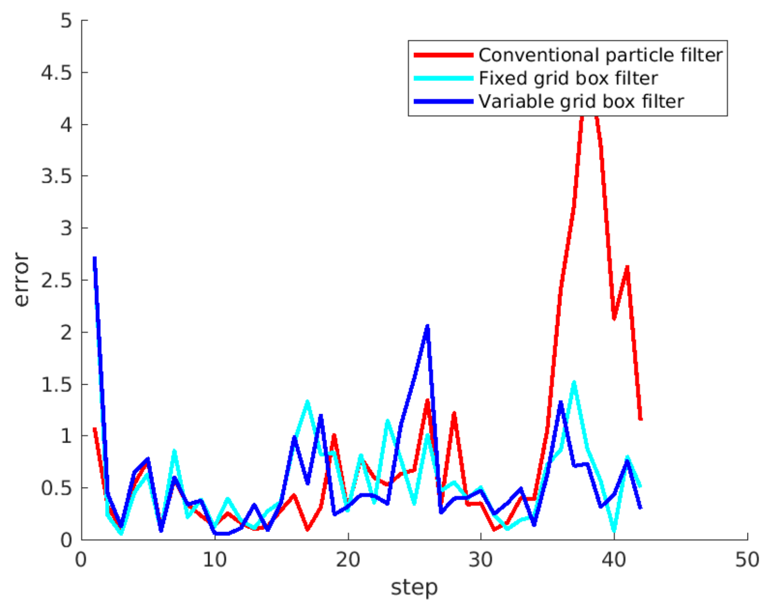

Figure 5.10: Second test, error plot.

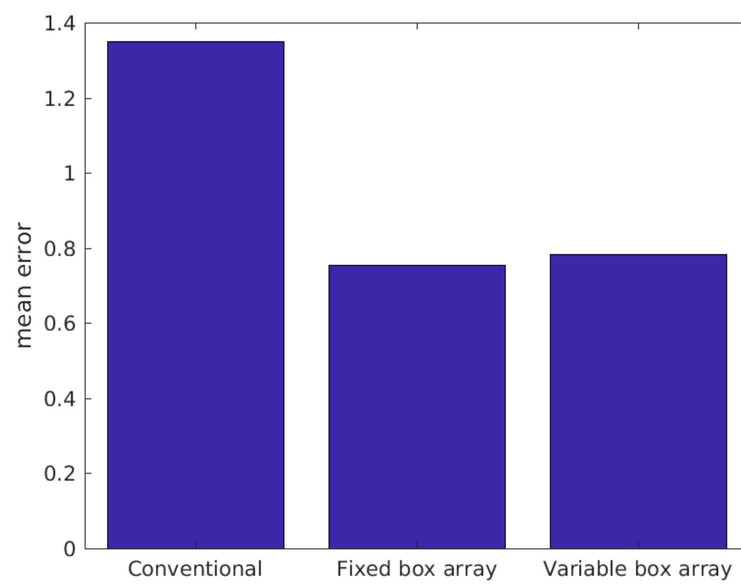

Figure 5.11: Second test, mean error plot.

5.1.2 Scenario 2

The second scenario that has been tested is based on the simulations in Jaulin [2011], and the path shown on Figure 5.12. The measurement equations are presented in Equation

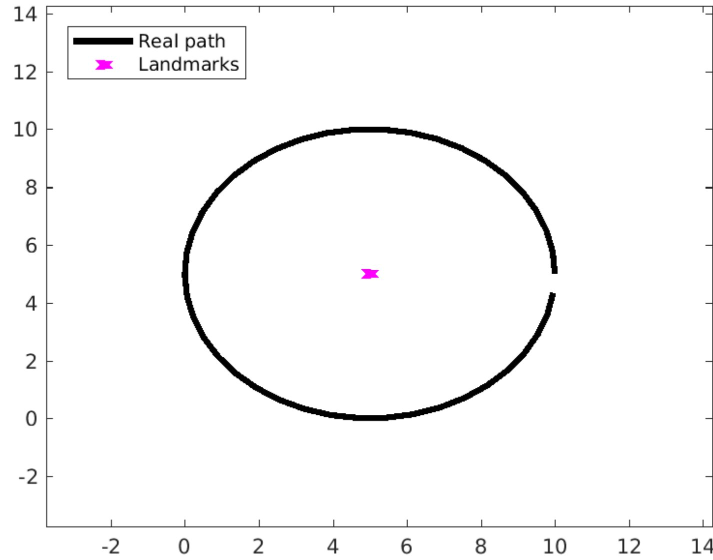

Figure 5.12: Second scenario.

5.6. These equations show that the robots has 3 sensors: an angle sensor (a compass) and a velocity sensor, like in the first scenario, and an angle difference sensor.

$$\begin{cases} y_1 = v + e_v \\ y_2 = \theta + e_\theta \\ y_3 = \text{atan2}(\mathbf{x}_2 - \mathbf{m}_y, \mathbf{x}_1 - \mathbf{m}_x) - \theta(k) + e_{measure} \end{cases} \tag{5.6}$$

One reason why this example is interesting is because only one landmark is used, and that makes it impossible to have an estimate of the robots position using only geometry. Using filtering algorithms it is still possible to estimate where the robot is.

Test 3

The values for the variances used are:

$$\begin{cases} \sigma^2 = 0.500 \\ \sigma_v^2 = 4.000 \\ \sigma_\theta^2 = 1.000 \end{cases} \tag{5.7}$$

Being σ^2, σ_v^2 and σ_θ^2 the error variances from the measure of the angle difference, speed and robot angle sensors, respectively. It can be seen that the box particle filtering algorithms have performed differently during this simulation. Since the size of the possible space (size of the room) is not much bigger than the space where the robots passes, the fixed array performed a lot better than the other algorithms. That happens because the probability outside of this region is always zero, which means that the algorithm performs a second test and corrects the estimation based on it.

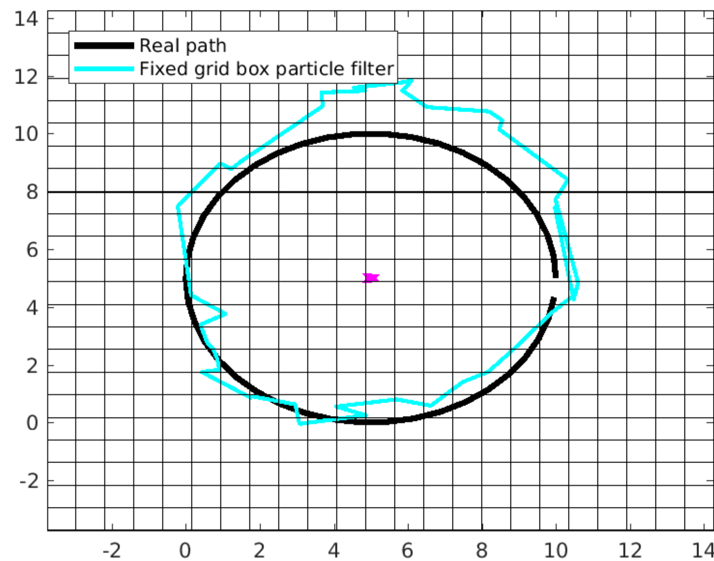

Figure 5.13: Third test, fixed array box particle filtering.

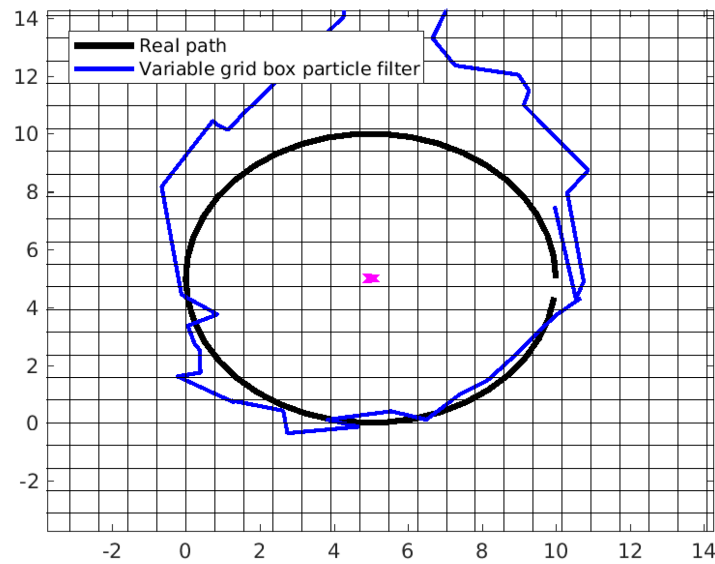

Figure 5.14: Third test, variable array box particle filtering.

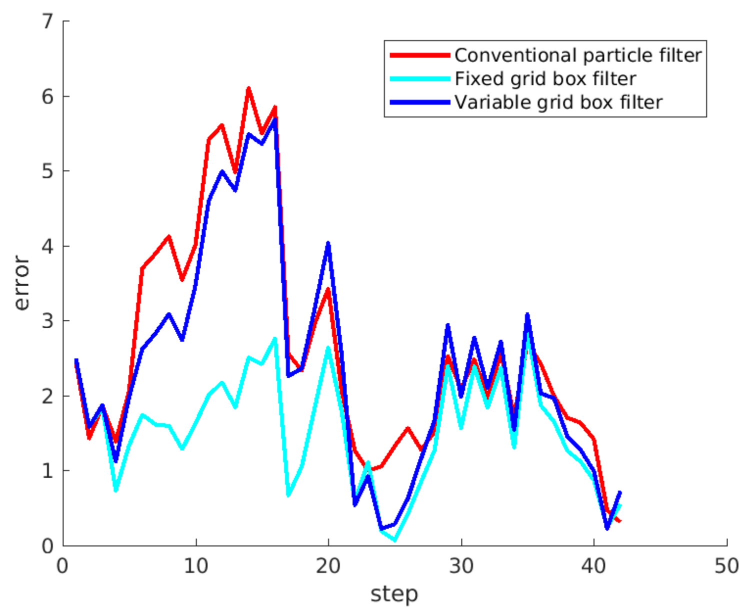

Figure 5.15: Third test, error plot.

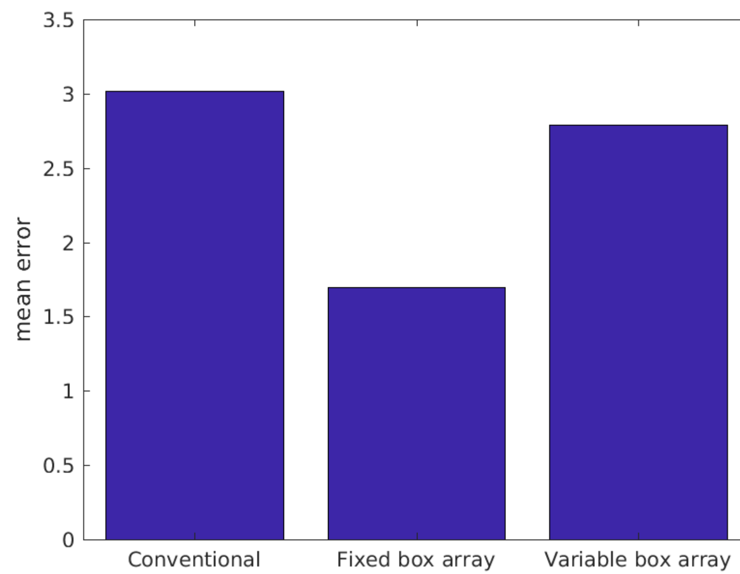

Figure 5.16: Third test, mean error plot.

5.2 Time performance

Even though the box particle filtering might give better estimations than the conventional particle filtering, the conventional particle filtering was quicker than the interval versions in the implemented Matlab code. Analysing the first and second tests of scenario 1, it can be seen on Figure 5.17 that the interval algorithms may need much more processing time than the conventional algorithm.

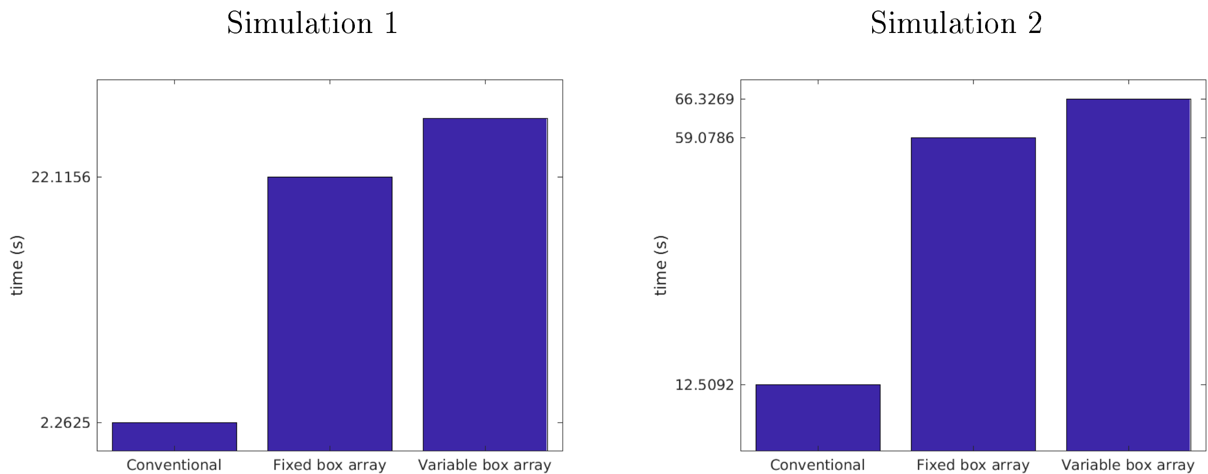

Figure 5.17: Time spent by the computer (in seconds) in each estimator during simulations 1 and 2.

This can be caused by many factors. At first, it seemed obvious that both box particle filtering algorithms should be slower since they have to compute the integral of the bidimensional probability density function for each box. A better analysis of the code using Matlab tools showed that both the measurement update and the state update phases were far slower than the same stages in the conventional particle filtering implementation. That happens because the access times of the elements in the interval class arrays and Matlab cell arrays (also used in the implementations) were bigger than the access time of the arrays of other common inbuilt variable types. Implementing the same algorithm in other languages (like C or C++) or using a more completely developed interval library as the basic engine could make the algorithm more efficient.

This was not a problem for this project since the objective has always been to implement it and compute simulations to calculate the difference in estimates between the interval algorithms and the conventional particle filter algorithms. Nevertheless, for a real application on a robot having to calculate each step in real time, a more efficient approach would be crucial for the feasibility of the implementation.

5.3 Number of boxes/particles

In this section, 10 different simulations with increasingly bigger numbers of boxes are compared, using the variance values shown in Equation 5.8.

$$\begin{cases} \sigma^2 = 0.500 \\ \sigma_v^2 = 3.000 \\ \sigma_\theta^2 = 1.047 \end{cases} \tag{5.8}$$

On Figure 5.18 and Figure 5.19, the grid, real path and estimated path are all shown for each of the simulations. It can be seen that the quality of the estimation does not change much for a number of boxes higher than 32, meaning that the estimation will still be limited to the intrinsic randomness of the system.

On Figure 5.20 the error plot of some of these simulations can be seen and on Figure 5.21 the mean error for all the simulations is represented. As it can be seen on Figure 5.21, from 2 to 32 boxes, the error mean lowers considerably, but it seems to be constant for bigger number of boxes. The time spent calculating the simulations were increasingly higher though, as shown on Figure 5.22. Even though the quality of the estimations stayed the same, the computational cost got higher with higher numbers of boxes. Increasing the number of boxes in the box particle filter increases the possible accuracy of the estimator, but it also decreases the efficiency much more than augmenting the number of particles in the conventional particle filtering. Therefore, this accuracy parameter must be calibrated according to the characteristics of the system in order not to waste computer power.

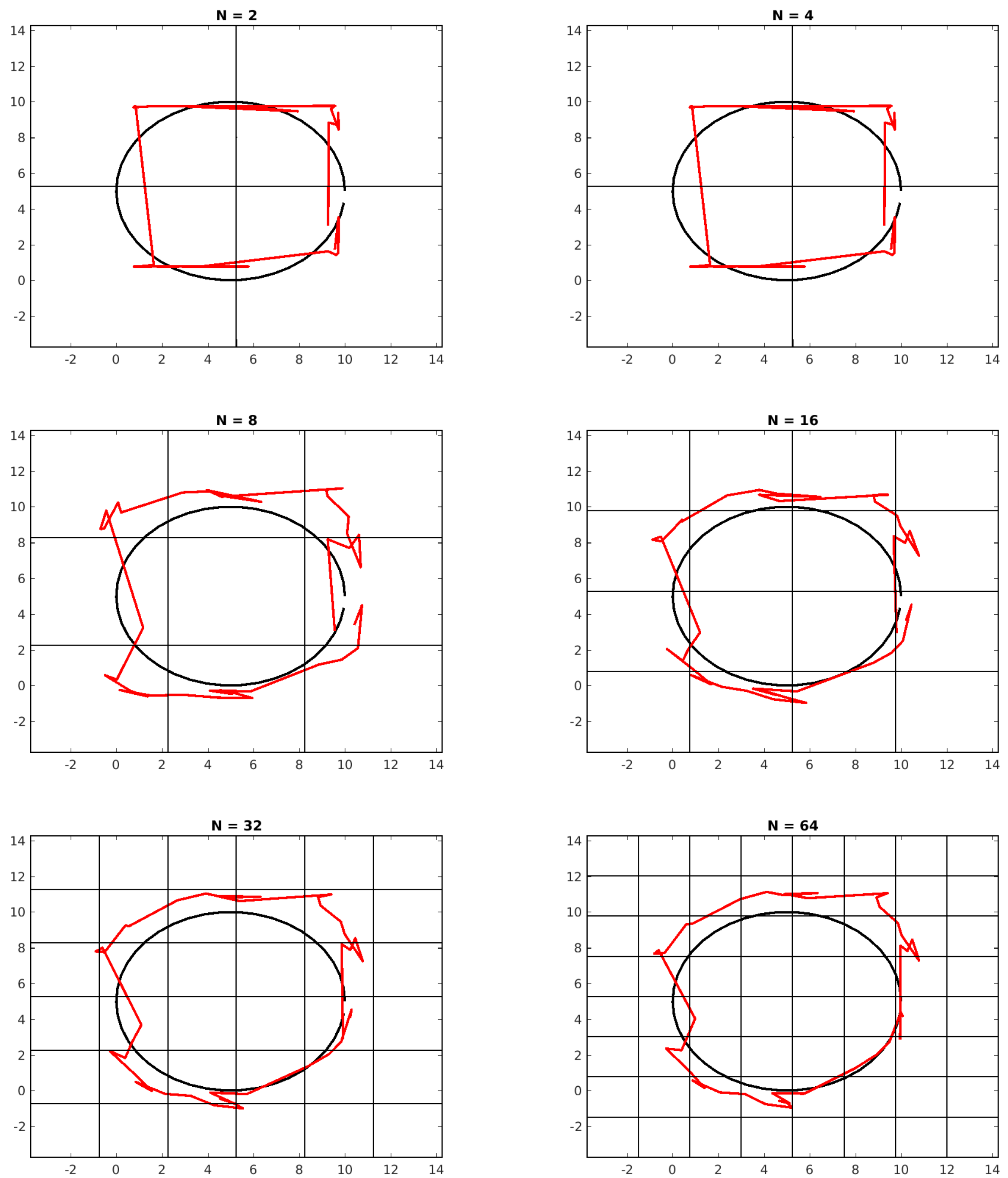

Figure 5.18: Comparison for different number of number of boxes. Simulations with 2, 4, 8, 16, 32 and 64 boxes.

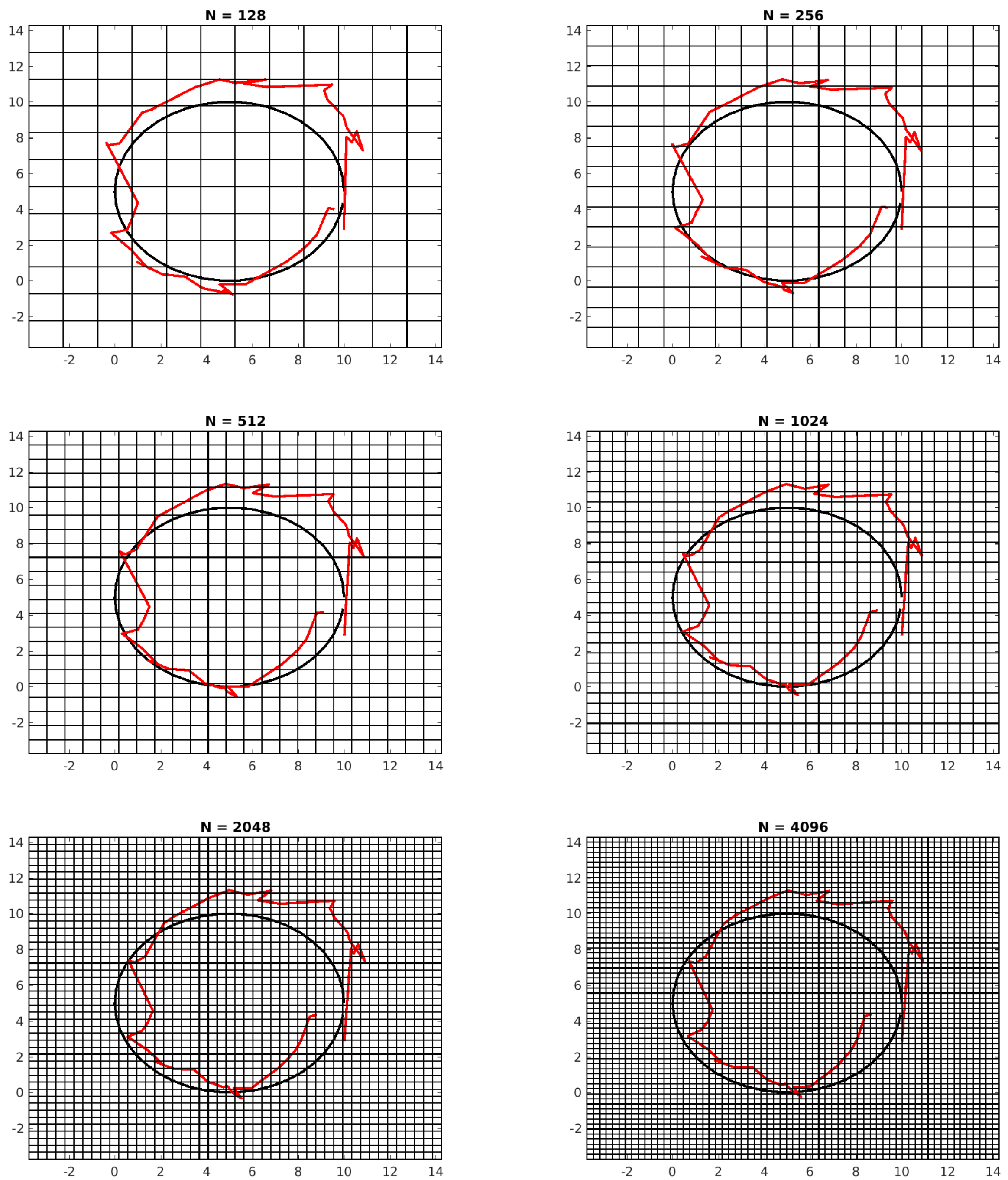

Figure 5.19: Comparison for different number of number of boxes. Simulations with 128, 256, 512, 1024, 2048 and 4096 boxes.

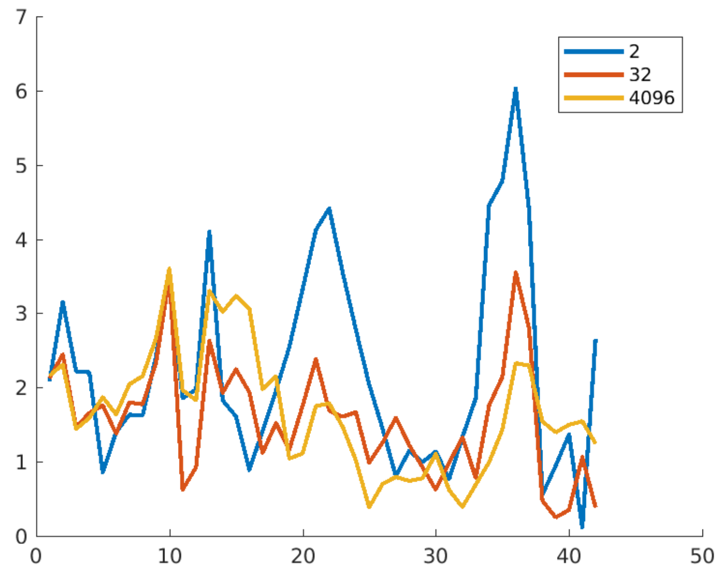

Figure 5.20: Error plot for the simulations with 2, 32 and 4096 boxes.

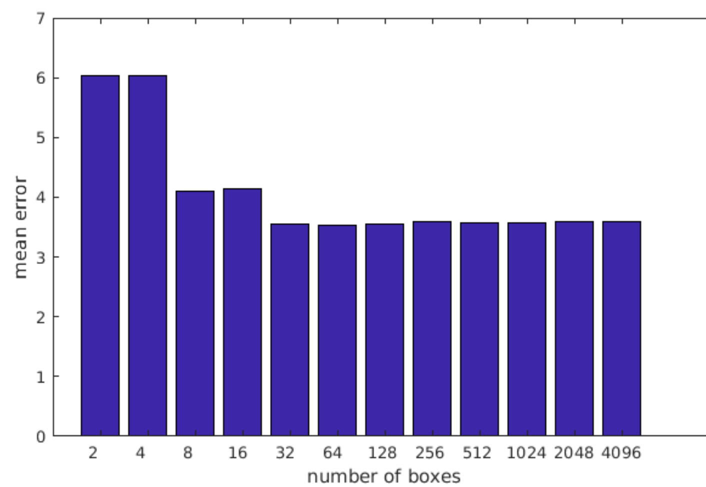

Figure 5.21: Error mean for all simulations.

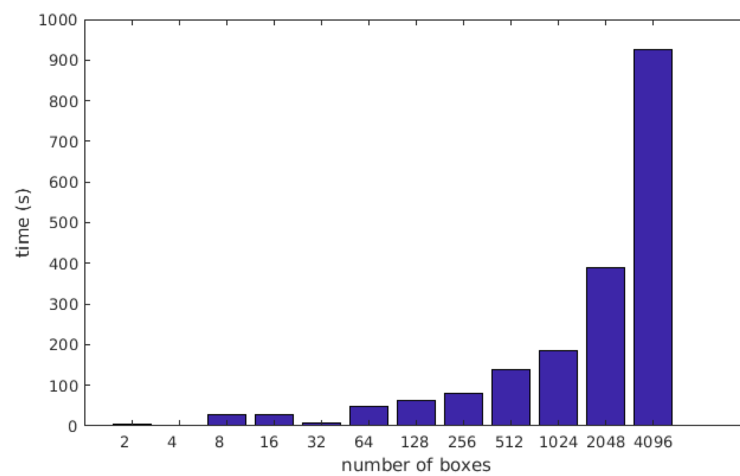

Figure 5.22: Time spent calculating all simulations, in seconds.

5.4 Simultaneous Localization and Mapping

Simultaneous localization and mapping (SLAM) is the problem of estimating the surrounding environment (creating a map) while simultaneously getting localized in it using sensors. Since a big part of the SLAM problem consists in estimating the position in a map (even though the map is being created in real time), an estimator is an important part of the algorithm. The most used estimator used with SLAM is the extended Kalman filter, as showed in the seminal work Smith and Cheeseman [1987]. A simple SLAM simulation has been carried out using the Box Particle Filter as estimator engine. The implementation is based on the online course Solà [2014].

5.4.1 Implementation

The robot follows the same model shown in Equation 5.1. The difference is that the sensors are supposed to be distance sensors which can detect the distance of walls (or objects) inside an angle interval $-\delta_\phi < \phi < \delta_\phi$.

After initialising the robot at position $(0, 0)$ with 100% certainty, the SLAM algorithm consists of the following operations:

- The robot moves, obtaining a new view of the environment. The movement is calculated according to equation 3.1a, increasing the uncertainty. In this implementation, the robot's movement is analogous to the box particle filter state update stage, also applying the resampling process.
- The robot observes new features on the map, extracting some measurement from them (like distance, for example). These new detected landmarks have uncertain locations because of the errors in the sensors and in the robot's position. In this project's simulation, the landmarks are all stored in an array which contains their real positions. In a real application, a landmark detection system must be implemented.
- The robot detects previously known landmarks, and uses them to refine the estimation of both the robot's position and the other landmarks.

  In this implementation, for simplicity, the detection of previous landmarks only refine the estimation of the robot and the detected landmarks (the position of each landmark is defined to be the mean of all the estimated positions of the same landmark). This is achieved by keeping track of the number of times n each landmark has been observed, and every time a new observation is done, the new estimation is given by:

$$\hat{\mathbf{x}}_{n+1}^i = \frac{\bar{\mathbf{x}}_{n+1}^i + n\hat{\mathbf{x}}_n^i}{n+1}, \tag{5.9}$$

  where $\bar{\mathbf{x}}_n^i$ is the nth observation and $\hat{\mathbf{x}}_n^i$ is the mean after the nth observation.

5.4.2 Results

The first test was done using the following variance values:

$$\begin{cases} \sigma^2 = 0.700 \\ \sigma_v^2 = 2.000 \\ \sigma_\theta^2 = 0.500 \end{cases} \tag{5.10}$$

The room, the resulting estimated map and the robot's trajectory can be seen on Figure 5.23. For this simulation, the robot ran through the proposed path twice. Since each landmark's estimated position is the mean of all observations of it, passing more than once gave a better estimation of the map.

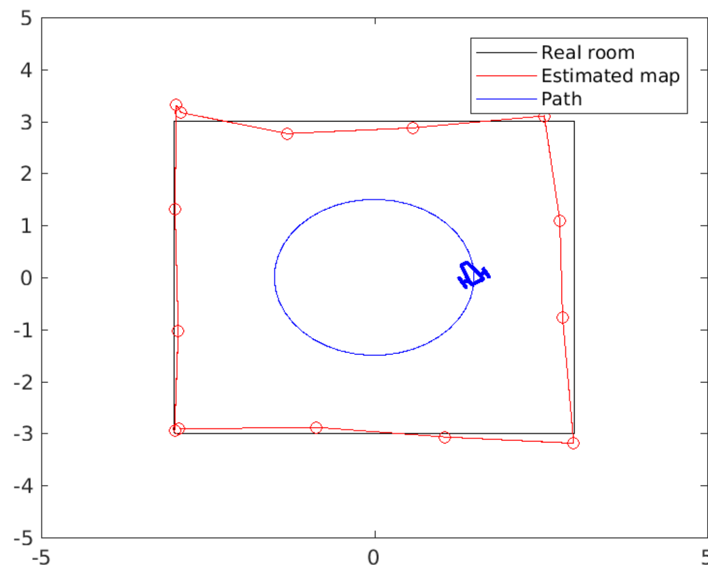

Figure 5.23: SLAM simulation 1. Real map, estimated map and robot's trajectory.

On Figure 5.24, a second simulation with a more complex environment can be seen, using the following values:

$$\begin{cases} \sigma^2 = 1.200 \\ \sigma_v^2 = 0.200 \\ \sigma_\theta^2 = 0.010 \end{cases} \tag{5.11}$$

Both SLAM simulations showed how the implemented box particle filter can be used as an estimator engine in applications.

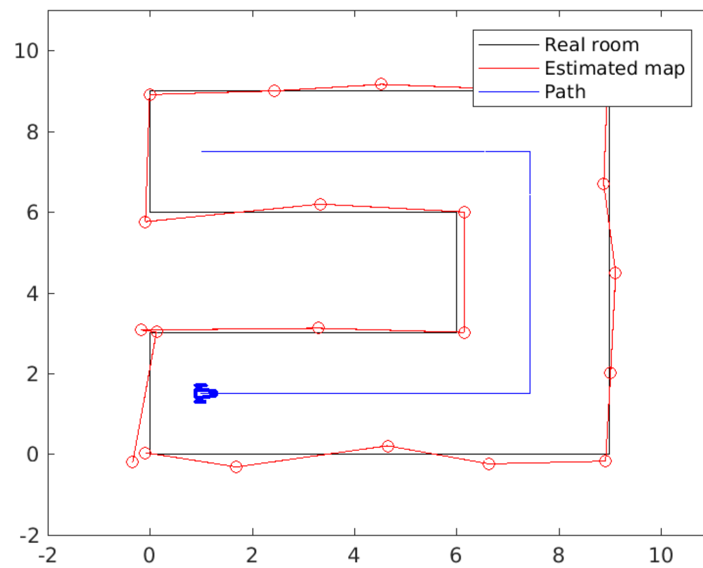

Figure 5.24: SLAM simulation 2. Real map, estimated map and robot's trajectory. The robot performs a 360 degrees rotation at each corner, in order to see the entire room.

Sample depletion

During the SLAM simulations, it was observed that if boxes of small width values are used, the estimator might lose track of the robot. This is one of the problems of particle filters in general, and it is important to further study the resampling stage, since it is the stage that might create this problem.

Chapter 6

Conclusion

The proposed box particle filtering algorithm proposed in Blesa et al. [2015] has been successfully implemented and tested in a 2D robot localisation problem. Results coming from various simulations show that the estimator can compute estimations usually as good as the conventional particle filtering, but with more robustness when the error is big and with less chance of causing sample depletion.

It has also been shown that the Box Particle Filter can be used as an estimator engine for a SLAM implementation.

Sample depletion problems were observed during the SLAM tests. These problems were discussed in Blesa et al. [2015], and the resampling problem must be further studied in order to better handle this problem.

The box particle filter was implemented in Matlab, and the time used by the computer during its execution were higher than the time needed by the conventional particle filter algorithm. To fully analyse if it can be used in practice as a substitute for the conventional particle filter, a more efficient implementation must be developed, using the computer's resources in a better way.

Appendix A

Implementation of Interval sine and cosine functions

In this section, the used realisation of the Interval sine and cosine functions implemented in the Matlab Interval class will be explained.

A.1 Cosine function

Suppose the function the angle is defined as the interval $[x]$ and its cosine $[y] = cos([x])$ must be computed.

1. The first step is to measure the width of the interval $[x]$. If $width([x]) = x_{high} - x_{low} \geq 2\pi$, then the resulting interval is $[y] = [-1, 1]$.
2. If not, an offset is computed using the following property:

   $$cos(x + n\pi) = (-1)^n cos(x)$$

   The objective of this offset is to place the angle interval between $-\pi$ and π. The value n is computed and a new angle $[x'] = [x] - n\pi$ is used as the input of the function.
3. A list L is initialised with the values $L(1) = (-1)^n \cos(x'_{low})$, $L(2) = (-1)^n \cos(x'_{high})$ and, if the $[x'] \supset 0$, the value $L(3) = (-1)^n$ is also added to the list.
4. The limits of the resulting value $[y]$ will be calculated as: y_{low} is the smallest value of L and y_{high} is the biggest value of L.

The Matlab code is giving in the following:

```
function res = cosInterval(x)
    if(x.width >= 2*pi)
        res = Interval([-1,1]);
    else
        if(x.ub > pi)
            n = floor(x.ub/pi);
            x = x-n*pi;
        elseif(x.lb < pi)
            n = floor(abs(x.ub/pi));
            x = x+n*pi;
        end

        L = cos([x.lb,x.ub]);
        if(x.contains(0))
            L = [L,1];
        end
        L = ((-1)^n)*L;
        res = Interval([min(L),max(L)]);
    end
end
```

Listing A.1: Matlab implementation of Interval cosine

A.2 Sine function

No new function has been implemented for the sine function. The cosine function is executed using the fact that $\cos(x - \pi/2) = \sin(x)$.

Bibliography

F. Abdallah, A. Gning, and P. Bonnifait. Box particle filtering for nonlinear state estimation using interval analysis. *Automatica*, 44(3):807–815, 2008.

Joaquim Blesa, Françoise Le Gall, Carine Jauberthie, and Louise Travé-Massuyès. State estimation and fault detection using box particle filtering with stochastic measurements. *International Workshop on Principles of Diagnosis*, 2015.

A. Gning, L. Mihaylova, and F. Abdallah. Mixture of uniform probability density functions for non linear state estimation using interval analysis. In *Information Fusion (FUSION), 2010 13th Conference on*, pages 1–8. IEEE, 2010.

Frefrik Gustafsson, Fredrik Gunnarsson, Niclas Bergman, Urban Forssell, Jonas Jansson, Rickard Karlsson, and Per-Johan Nordlung. Particle filters for positioning, navigation and tracking. *IEEE Transactions on Signal Processing*, 2002.

L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics,*. Springer-Verlag, 2001.

Luc Jaulin. Set-membership localization with probabilistic errors. *Elsevier*, 2011.

Simo Sarkkä. *Bayesian Filtering and Smoothing.* Cambridge University Press, 2013.

Dan Simon. *Optimal State Estimation. Kalman,* H_∞ *and Nonlinear Approaches.* John Wiley & Sons, Inc, 2006.

R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *Int. Journal of Robotics Research, 5(4):56–68*, 1987.

Joan Solà. Simulataneous localization and mapping with the extended kalman filter, a very quick guide. http://www.joansola.eu/, 2014.