Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Departamento de Automática, Ingeniería Electrónica e Informática Industrial

Master on Industrial Electronics

Hardware-Based Particle Filter with Evolutionary Resampling Stage

Author: Alfonso Rodríguez Medina

Advisor: Félix Moreno González

March 2014

Master Thesis

Preface

The present document reports all the work that I have done so far at Centro de Electrónica Industrial (Universidad Politécnica de Madrid) in order to complete the Master of Research (MRes) degree program.

The chosen research field for this thesis is artificial intelligence, which has been a very hot topic in our society for a long time. Lots of science-fiction novels portrait intelligent machines, which are capable of showing intelligent behavior. Real-world systems are still far from those levels of intelligence and reasoning capabilities. However, more and more complex ideas appear as technology evolves (e.g. autonomous vehicles, robot assistants, etc.).

Artificial intelligence constitutes itself a huge research field, with a large number of different branches (actually, new branches keep appearing almost every year). In this thesis, I have focused my efforts on two of these branches: particle filtering and evolutionary computation. This document presents a general overview of these two important topics, introducing the basic theory concepts needed to understand the proposed architecture and the latter results, which are also included in the final chapters.

Alfonso Rodríguez

Madrid, Spain

March 2014

Contents

Introduction

I. Motivation

Autonomous systems require, in most of the cases, reasoning and decision-making capabilities. Moreover, the decision process has to occur in real time. Real-time computing means that every situation or event has to have an answer before a temporal deadline. In complex applications, these deadlines are usually in the order of milliseconds or even microseconds if the application is very demanding. In order to comply with these timing requirements, computing tasks have to be performed as fast as possible. The problem arises when computations are no longer simple, but very time-consuming operations.

A good example can be found in autonomous navigation systems with visual-tracking submodules where Kalman filtering is the most extended solution. However, in recent years, some interesting new approaches have been developed. Particle filtering, given its more general problem-solving features, has reached an important position in the field.

II. Aim

The aim of this thesis is to design, implement and validate a hardware platform that constitutes itself an embedded intelligent system. The proposed system would combine particle filtering and evolutionary computation algorithms to generate intelligent behavior.

Traditional approaches to particle filtering or evolutionary computation have been developed in software platforms, including parallel capabilities to some extent. In this work, an additional goal is fully exploiting hardware implementation advantages. By using the computational resources available in a FPGA device, better performance results in terms of computation time are expected. These hardware resources will be in charge of extensive repetitive computations. With this hardware-based implementation, real-time features are also expected.

III. Previous Work

Embedded intelligence has already been studied at CEI (Centro de Electrónica Industrial). In [1], two different approaches are evaluated: on the one hand, a particle filter for vehicle trajectory prediction; on the other hand, an artificial-neural-network-based cognitive architecture. In addition, the author gives reasons to embed intelligence on chip, and presents some interesting examples. Another example of

artificial intelligence applications can be found in [2], where the author proposes a novel distributed artificial network for image compression in wireless visual sensor networks (WVSNs).

A lot of research has also been conducted at CEI on the field of evolutionary computation. For instance, in [3] and all its related works and publications, an evolutionary algorithm is used in order to generate a self-adaptive evolvable hardware platform, suitable for image processing tasks.

IV. References

[1] Salvador, Rubén, "Sistemas Embebidos Inteligentes," *Master Thesis*, Sept. 2008

[2] Aledo, David, "Compresión de imágenes optimizada en consumo energético para redes inalámbricas," *Master Thesis*, Feb. 2013

[3] Mora, Javier, "Noise-Agnostic Self-Adaptive Evolvable Hardware for Real Time Video Filtering Applications," *Master Thesis*, Sept. 2013

Particle Filtering

I. Introduction

Real-world systems represent a great challenge when trying to analyze them. State estimation and prediction have been considered major concerns in the field. Hence, a lot of research has been conducted regarding these topics. One of the most significant examples is the so-called Kalman filter. First introduced in 1960 [1], it has been deeply studied and cited in the literature [2].

The Kalman filter is used to estimate the estate of a discrete process in which some measurements are taken. The model can be expressed using the following equations:

$$x_k = A \cdot x_{k-1} + B \cdot u_{k-1} + w_{k-1}$$

$$z_k = H \cdot x_k + v_k$$

The first equation corresponds to the dynamic evolution of the process (it is also called process model), whereas the second represents the measurement model, i.e. which state variables can be observed (notice that not all state variables might be observable). The variables w_{k-1} and v_k represent the process noise and the measurement noise respectively, and each follows a normal distribution with the following parameters:

$$w_{k-1} \sim N(0, Q)$$

$$v_k \sim N(0, R)$$

Kalman filters have two main stages: the prediction stage, in which the process model equation is used in order to predict the next state; and the update stage, in which the measurement model is used to correct that prediction. The correction algorithm adjusts each prediction using the actual measurement and least squares optimization.

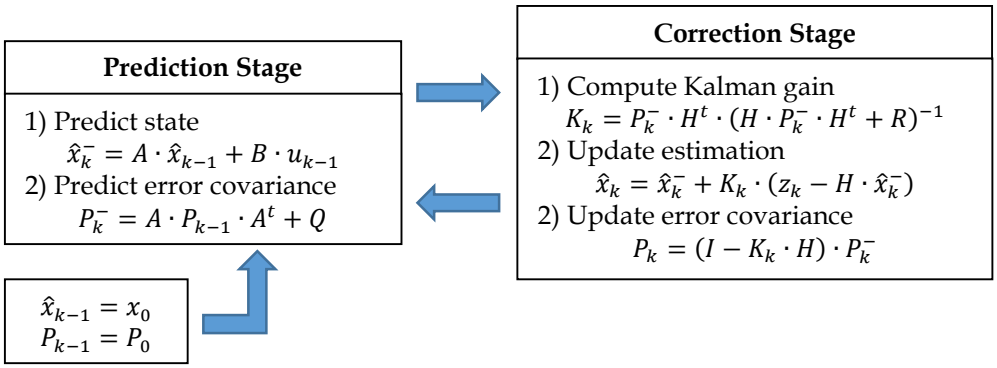


Fig. I.1. Kalman filter algorithm

In Fig. I.1, the common algorithmic implementation of the discrete Kalman filter is shown. Once the filter has been initialized, the algorithm iterates over each time step performing the two aforementioned stages.

The research field on which the Kalman filter has had larger impact is autonomous or assisted navigation. However, these filters have some important limitations, since they are linear Gaussian-based estimators. Real systems are sometimes non-linear and their noise does not necessarily have to be Gaussian, thus having to work with approximate models (e.g. linearized systems) which can lead to inaccurate results. In order to overcome these limitations, more complex approaches have been developed. Some of these new strategies will be discussed in following sections.

II. Particle Filters

In this section, the basic theory regarding particle filtering will be exposed. Particle filters are based upon complex mathematical concepts. Therefore, only a few hints will be provided regarding each constituting element, i.e. the basic knowledge needed in order to understand how a particle filter works.

II.1 Hidden Markov Models

Linear approximations of complex non-lineal systems tend to be inaccurate when the operating conditions suffer large variations, i.e. when we operate far from the linearization point. Therefore, these models are no longer useful for complex applications. In this section, a general overview of hidden Markov models (HMM) will be provided.

A Markov model is a stochastic model (i.e. systems which show random behavior) in which the Markov property is satisfied. The Markov property is usually used to refer to the memoryless property of a stochastic process, i.e. future states of the process depend only upon the present state, and not on the previous history of the system.

A hidden Markov model is a statistical Markov model in which the system is not fully observable (i.e. not all the states are visible to the observer). The basic idea is that the state sequence is unknown, i.e. "hidden". Since each state has a distribution function over each of the possible output values, this state sequence can be determined using the output values (i.e. the observable variables).

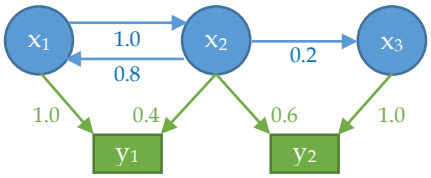


Fig. II.1. Hidden Markov model example

A simple example of a hidden Markov model has been provided in Fig. II.1. The stochastic system has three states (x_1, x_2, x_3) but only two possible output values or measurements (y_1, y_2). Transitions between each state have been represented as blue arrows, and output transitions as green arrows. These transitions have different probabilities (the numbers placed close to the arrows). Note that the sum of all the probabilities of the outgoing arrows with the same color in each state equals one.

II.2 Bayesian Inference

Inference can be defined as the process of drawing conclusions. If this process is carried out using a data set that may suffer random variations, it can be then specified as statistical inference. Bayesian inference is a method of inference in which Bayes' rule is used.

Bayes' rule computes the posterior distribution from the prior distribution and the so-called likelihood distribution (e.g. experience or knowledge). This computation method is nothing but an updating process. Bayes' rule can be expressed as follows:

$$P(a|b) = \frac{P(b|a) \cdot P(a)}{P(b)}$$

In the previous formula, $P(a|b)$ represents the posterior distribution, $P(a)$ is the prior distribution, $P(b|a)$ is the likelihood distribution (i.e. what is the probability of obtaining b after having observed a) and $P(b)$ is the marginal likelihood, which is independent of the hypothesis which is being tested (i.e. it does not affect the posterior distribution if the hypothesis is changed).

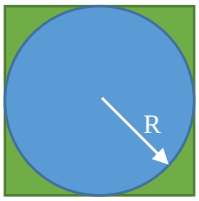
Bayes' rule can be explained in terms of a simple example: imagine that an old friend tells us that he has bought a new house. Consider three different hypothesis: the new house is in a big city, the new house is in the countryside and the new house is under the sea. Now imagine that our friend gives us a photograph in which the house location appears. This photograph represents the prior distribution.

- If the picture shows a city, we will consider that it is likely that the new house is in a big city. If the picture shows a large green field, we will follow the same reasoning process to state that it is likely that the house is in the countryside.
- However, it the picture shows the sea, we will still consider the third hypothesis unlikely. The reason for this is simple: in the first two cases, our previous knowledge (i.e. the likelihood distribution) tells us that it is possible for a person to live in the city or in the countryside, whereas in the third one, our experience tells us that people do not live underwater.

With this example, it is possible to notice that Bayesian inference does not only rely on evidence, but in previous knowledge or experience to update the conclusions that are drawn. Particle filtering takes advantage of this specific updating process.

II.3 Monte Carlo Methods

Monte Carlo methods, also called Monte Carlo simulations, are a set of computational algorithms that rely on repetitive random sampling in order to obtain numerical results. These algorithms can be illustrated with a simple example. Imagine that we want to compute the value of π using Monte Carlo methods. First, it is important to keep in mind the following relationships:



$$A_{circle} = \pi \cdot r^2 = \pi \cdot R^2$$

$$A_{square} = l^2 = (2 \cdot R)^2$$

Running a Monte Carlo simulation consists of throwing random samples to the whole search space until a significant population is generated (i.e. with enough particles to consider the distribution as unbiased). Keeping this in mind, the value of π can be approximated using the following expressions:

$$A_{ratio} = \frac{A_{circle}}{A_{square}} = \frac{\pi}{4}$$

$$\pi = 4 \cdot A_{ratio} \xrightarrow{yields} \pi \approx 4 \cdot \frac{n_{circle}}{N}$$

The area ratio has been approximated as the quotient of the number of random samples inside the circle (n_{circle}) over the total number of samples drawn (N).

From now on, we will assume that processes are modelled as Markovian, non-linear, non-Gaussian state-space models. The hidden states will be noted as x_t, and the observations as y_t. The equations of the model will be expressed as follows:

$$p(x_t|x_{t-1})$$

$$p(y_t|x_t)$$

The first equation corresponds to the process model (i.e. the dynamic equations of the system) and the second to the measurement model. Note the difference between this problem statement and the equations that were used to introduce the Kalman filter, which were less general. Using Bayes' rule, which can also be named Bayes' theorem, it is possible to obtain the posterior distribution:

$$p(x_{0:t}|y_{1:t}) = \frac{p(y_{1:t}|x_{0:t}) \cdot p(x_{0:t})}{\int p(y_{1:t}|x_{0:t}) \cdot p(x_{0:t}) \cdot dx_{0:t}}$$

The ideal situation would be to be able to simulate (or generate) N independent and identically distributed samples (i.e. the so-called particles) from the posterior distribution $p(x_{0:t}|y_{1:t})$. However, in real-world applications this sampling strategy is not available in most of the cases.

In order to cope with those processes in which perfect Monte Carlo sampling is not available, another sampling strategy called Importance Sampling (IS) is used. An arbitrary distribution, the so-called importance sampling distribution (also referred to as the proposal distribution or the importance function) is introduced. Given that now the samples are not drawn from the posterior distribution, but from the arbitrary importance distribution, they have to be weighted in order to obtain the same results.

$$\pi(x_{0:t}|y_{1:t})$$

$$w(x_{0:t}) = \frac{p(x_{0:t}|y_{1:t})}{\pi(x_{0:t}|y_{1:t})}$$

The previous expressions represent the importance sampling distribution and the weight functions respectively.

Importance sampling is considered a good Monte Carlo integration method. However, it is not suitable for iterative implementations, as in the Kalman filter (refer to Fig. I.1). Therefore, some modifications have to be introduced in the algorithm so that all equations can be expressed in a recursive manner. The following equations are the result of this modifying process, and this implementation is named sequential Monte Carlo method:

$$\pi(x_{0:t}|y_{1:t}) = \pi(x_{0:t-1}|y_{1:t-1}) \cdot \pi(x_t|x_{0:t-1}, y_{1:t})$$

$$w(x_t) \propto w(x_{t-1}) \cdot \frac{p(y_t|x_t) \cdot p(x_t|x_{t-1})}{\pi(x_t|x_{0:t-1}, y_{1:t})}$$

This approach is said to be recursive because the current value is computed using the previous values and performing an arithmetic operation on them (e.g. multiplication).

A special case appears when the chosen importance sampling distribution is the prior distribution. The equations are then expressed as follows:

$$\pi(x_{0:t}|y_{1:t}) = p(x_{0:t}) = p(x_{0:t-1}) \cdot p(x_t|x_{t-1})$$

$$w(x_t) \propto w(x_{t-1}) \cdot p(y_t|x_t)$$

In conclusion, Sequential Monte Carlo methods are based upon:

- Importance Sampling from prior distribution, i.e. process model $p(x_t|x_{t-1})$.
- Weight update using the measurement model $p(y_t|x_t)$.
- Recursive implementation (i.e. sequential).

For further information on sequential Monte Carlo methods, please refer to [3].

II.4 Characteristics

The basic particle filter is based upon SIS (Sequential Importance Sampling). However, this sampling strategy generates the so-called particle degeneracy problem. It is assumed that a given population has suffered from this problem when, having run a finite time, the simulation has reached a state in which only one particle has its weight with non-zero value.

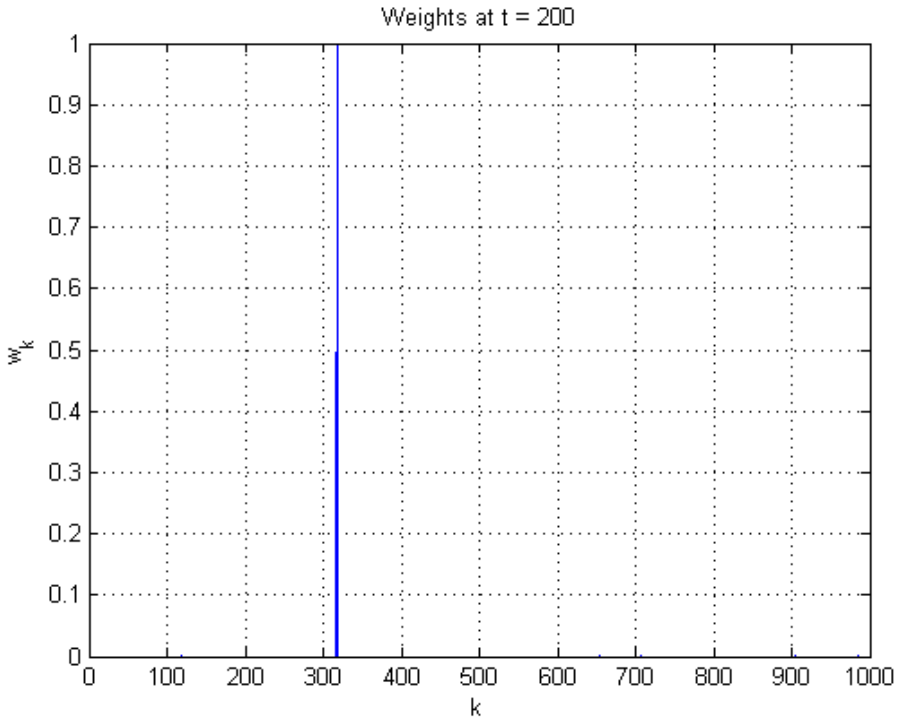


Fig. II.2. Particle degeneracy phenomenon

In Fig. II.2, a real particle filter application is shown. The x-axis represents the number of particles, whereas the y-axis represents the value of the different weights. Note that the effects of particle degeneracy are very significant. Although the simulation has run only up to t = 200 (i.e. finite time), the population is now biased, having only one particle that is representative.

The particle degeneracy phenomenon has been thoroughly analyzed and studied. In 1993, Gordon, Salmond and Smith introduced the so-called Bootstrap Filter [4]. In their work, a novel resampling strategy was presented in order to mitigate the effects of the aforementioned problem. Some authors consider this proposal as the first formal particle filter in history.

The resampling stage of the Bootstrap Filter is very simple: if the particle population has N individuals, then N random numbers from a uniform distribution $U(0,1)$ are drawn. Afterwards, this random value is compared with the cumulative sum vector of weights, selecting the resampled element as follows:

$$q_k = \sum_{i=0}^{k} w_i$$

$$u_j \in U(0,1),\ q_{n-1} < u_j \leq q_n \xrightarrow{yields} r_j = x_n$$

In the previous equations, q is the cumulative sum vector of weights, u_j is the random sample drawn from the uniform distribution, r is the resampled population vector and x is the posterior population vector (i.e. before the resampling stage).

Note that with this approach, there is a resampling process in each time step. More recent examples perform adaptive resampling strategies checking whether the effective number of particles is below some threshold or not, but the underlying concept is the same. The effective number of particles is usually computed as $\frac{1}{\sum_i w_i^2}$.

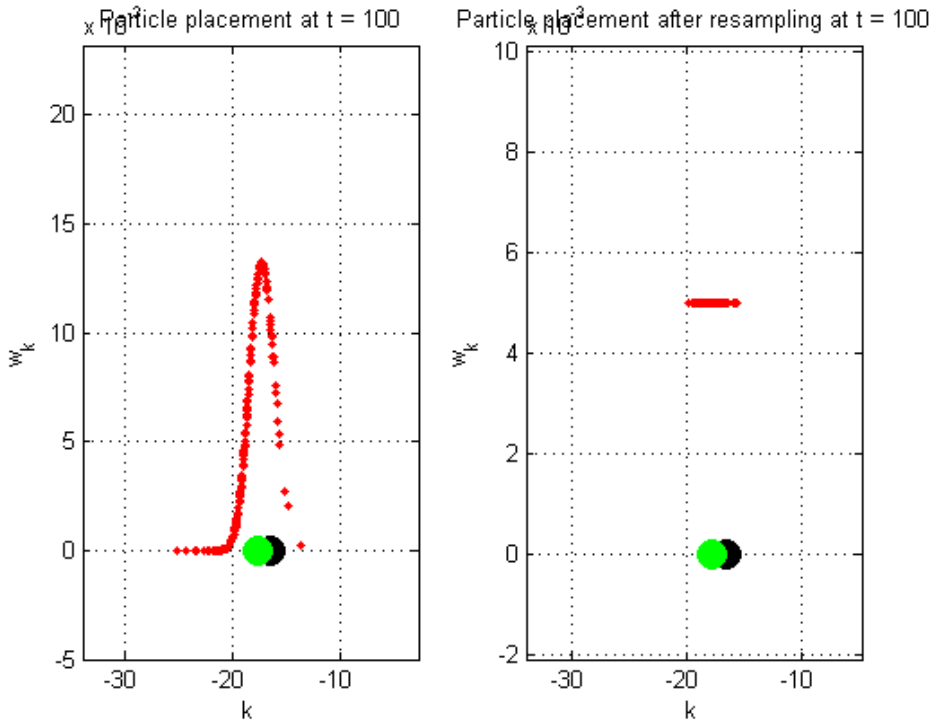


Fig. II.3. Sample impoverishment phenomenon (measurement in black; estimation in green)

Further analysis regarding resampling stages have discovered a new problem: the so-called sample impoverishment phenomenon. This phenomenon occurs when the whole set of individuals do not approximate accurately the posterior distribution, usually because a vast majority of them is at the same point (i.e. the particles are the same). Studies have also shown that diversity loss is caused by using suboptimal resampling strategies (as the one used in the Bootstrap Filter).

A good example of the sample impoverishment phenomenon can be found in Fig. II.3. The picture shows the particle distribution and the weights before (on the left side) and after (on the right side) performing resampling. It is clear that the resampling stage modifies the population in order to reduce the particle degeneracy problem (which indeed is mitigated), but there is a clear loss in terms of population diversity. Note that after performing resampling, all the weights have the same value. This is part of the resampling stage itself.

Once the main problems of particle filtering have been discussed, a question arises: how can particle filtering estimate the state not as a probability distribution but as a unique point? The answer is quite simple, and has been represented in the following equation:

$$\hat{x} = \sum_{k} w_k \cdot x_k$$

Therefore, the estimated state is the weighted sum of all the particle states.

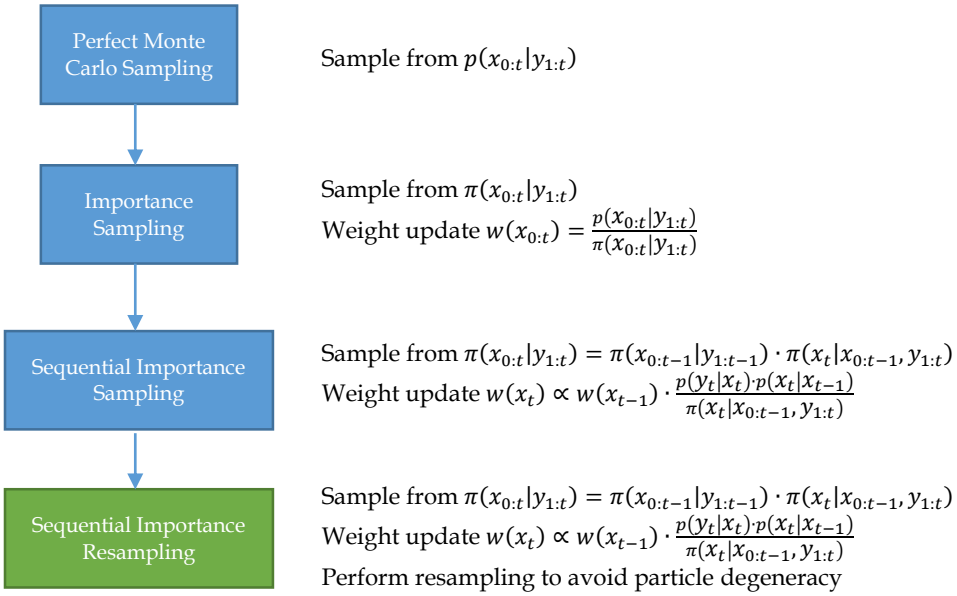


Fig. II.4. Particle filter algorithmic evolution

The natural evolution of particle filtering is shown in Fig. II.4. From perfect Monte Carlo sampling (i.e. samples drawn from the posterior distribution) to SIR (Sequential Importance Resampling), including IS and SIS, each stage goes a step further into the final approach. Current research lines are focused on the green rectangle (i.e. Sequential Importance Resampling), trying to improve the resampling strategies that are used. These lines will be presented in the next section.

To summarize, particle filters are a very useful tool in estimation and prediction tasks, especially when dealing with complex systems (non-linear non-Gaussian models). However, particle filter designers have to deal with two big problems:

- Particle degeneracy: the effective number of particles (i.e. those whose weights are bigger than zero) tends to one in a finite simulation time. This problem appears per se in the basic particle filter implementation (SIS), and can be solved using resampling stages.
- Sample impoverishment: the population does not represent accurately the posterior distribution (for instance, all the particles in the population are the same). This problem is caused by suboptimal resampling strategies, and can only be mitigated changing the resampling stage.

III. Current Research Lines

In this section, state-of-the-art particle-filtering techniques will be reviewed. These techniques can be divided into two main groups: algorithmic improvements over the basic particle filter, and implementation improvements (i.e. changes in the technology used to implement the particle filters). This research work is more focused on algorithmic improvements based on evolutionary computation, which has also been included as a modification of the basic particle filter architecture in the literature [5]. Refer to the following chapter for further information on this topic.

The most remarkable implementation improvements regarding the scope of this thesis are the ones related to hardware implementations. Several approaches have been presented and can be found in the literature. The first one that will be cited is [6]. In this paper, the authors present a hardware implementation of the Bootstrap Filter [4]. The design is described using VHDL and the target platform is a FPGA (specifically a Xilinx Virtex-2). FPGA parallel processing capabilities are used in [7], where the same particle filter is implemented in order to process data concurrently. Every stage performs its computation while the others are working with other valid data, i.e. the processing is carried out in parallel. This parallel implementation also leads to a significant decrease in terms of resource consumption (on a Xilinx Virtex-5 FPGA). A different approach is presented in [8], since the implementation is not done only in hardware. On the contrary, a System on Programmable Chip (SoPC) approach is analyzed. An embedded processor carries out the weight computations, whereas the

particle update, which is a repetitive process, is speeded up using hardware accelerators. In addition, in this paper some elements from evolutionary computation, such as tournament selection algorithms in the resampling stage, are introduced. The platform used in [8] is an Altera Cyclone II FPGA, as opposed to the previous examples, in which only Xilinx devices were used. Another modified version of the particle filter that is implemented in a FPGA is presented in [9], this time using color histogram enhancements. The authors take advantage (again) of the parallel processing capabilities of the configurable device (another Xilinx Virtex-5 FPGA). Both weight and histogram calculations are carried out using these capabilities.

In the last few years, there has been an increase in the amount of works related to performance improvement in particle filter implementations. Complex particle filtering algorithms (e.g. with huge number of particles, high dimensional problems, etc.) require more and more computational resources. In order to enhance the system, parallel computing has appeared as a feasible alternative to the classical approach. In [10] the authors compare the results from a classic CPU implementation with the ones obtained from a GPU implementation (using CUDA). Their results show that the more parallel the approach is, the faster the processing can be done. However, some increase in the average error appears, due to the fact that each parallel block resamples from a small number of individuals and not from the whole population. Another example can be found in [11], where different parallel implementations (GPGPUs and multicore CPUs) are evaluated with a system that uses over one million particles to perform the computations. In this paper, an extensive sensitivity analysis is carried out, scaling parameters such as the number of particles per filter, the number of sub-filters, and even the state dimensions. The authors have also extended their research to real-time control applications of distributed computing approaches to particle filtering, as in [12].

IV. References

[1] Kalman RE. A New Approach to Linear Filtering and Prediction Problems. *J. Basic Eng.*. 1960;82(1):35-45

[2] Greg Welch and Gary Bishop. 1995. *An Introduction to the Kalman Filter*. Technical Report. University of North Carolina at Chapel Hill, Chapel Hill, NC, USA

[3] A. Doucet and A. Johansen *A tutorial on particle filtering and smoothing: Fifteen years later,*, 2008

[4] Gordon, N.J.; Salmond, D.J.; Smith, A. F M, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *Radar and Signal Processing, IEE Proceedings F* , vol.140, no.2, pp.107,113, Apr 1993

[5] Ziyu, Li; Yan, Liu; Lei, Song; Ying, Cheng, "Particle Filter Based on Pseudo Parallel Genetic Algorithm," *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on* , vol., no., pp.195,198, 21-23 June 2013

[6] Jung Uk Cho; Seung-Hun Jin; Xuan Dai Pham; Jae Wook Jeon; Jong-Eun Byun; Hoon Kang, "A Real-Time Object Tracking System Using a Particle Filter," *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* , vol., no., pp.2822,2827, 9-15 Oct. 2006

[7] El-Halym, H.A.A.; Mahmoud, I.I.; Habib, S. E -D, "Efficient hardware architecture for Particle Filter based object tracking," *Image Processing (ICIP), 2010 17th IEEE International Conference on* , vol., no., pp.4497,4500, 26-29 Sept. 2010

[8] Shih-An Li; Chen-Chien Hsu; Wen-Ling Lin; Jui-Pin Wang, "Hardware/software co-design of particle filter and its application in object tracking," *System Science and Engineering (ICSSE), 2011 International Conference on* , vol., no., pp.87,91, 8-10 June 2011

[9] Agrawal, S.; Engineer, P.; Velmurugan, R.; Patkar, S., "FPGA Implementation of Particle Filter Based Object Tracking in Video," *Electronic System Design (ISED), 2012 International Symposium on* , vol., no., pp.82,86, 19-22 Dec. 2012

[10] Jilkov, Vesselin P.; Wu, Jiande; Chen, Huimin, "Performance comparison of GPU-accelerated particle flow and particle filters," *Information Fusion (FUSION), 2013 16th International Conference on* , vol., no., pp.1095,1102, 9-12 July 2013

[11] Chitchian, M.; van Amesfoort, A.S.; Simonetto, A.; Keviczky, T.; Sips, H.J., "Adapting Particle Filter Algorithms to Many-Core Architectures," *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* , vol., no., pp.427,438, 20-24 May 2013

[12] Chitchian, M.; Simonetto, A.; van Amesfoort, A.S.; Keviczky, T., "Distributed Computation Particle Filters on GPU Architectures for Real-Time Control Applications," *Control Systems Technology, IEEE Transactions on* , vol.21, no.6, pp.2224,2238, Nov. 2013

Evolutionary Computation

I. Introduction

In 1859, Charles Darwin's *The Origin of Species* was published. In that book, the concept of natural selection is introduced as the main reason for a given population to evolve. In any real environment there are limited resources, thus having the individuals to compete for them. Natural selection is the phenomenon in which only those individuals that achieve high levels of adaptation to a specific environment survive. Therefore, natural selection can be expressed as the survival of the fittest.

Evolutionary progress is based on two basic elements: on the one hand, the aforementioned natural selection, or competition-based selection; on the other hand, genetic transference throughout generations of those characteristics (or traits) which make each individual better than the rest of the population.

Natural selection can be analyzed on a microscopic basis by means of molecular genetics. Genetics states that each individual has external characteristics (phenotype) that can be represented at a low level (genotype), i.e. each individual's phenotype is encoded by its genotype. Therefore, the phenotype can be built using the genotype.

In each generation, new individuals are generated. In biological environments, these individuals can be identical to their parents (e.g. mitosis) or inherit different traits from each parent (e.g. meiosis). In addition to that, some random changes tend to appear between generations (the so-called mutations) and contribute to have new individuals to evaluate. These genetic operations can be seen in Fig. I.1.

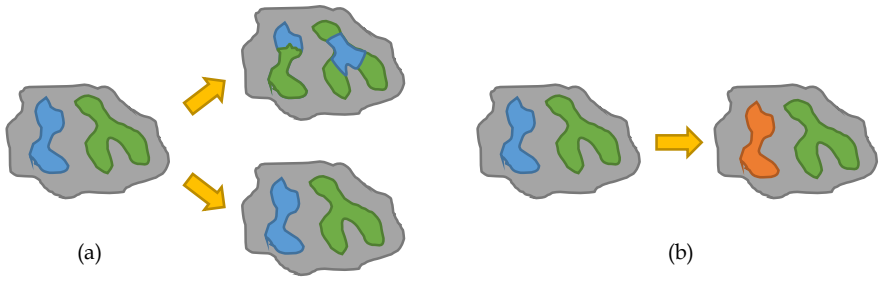


Fig. I.1. Genetic operations: (a) Meiosis (up) and mitosis (down). (b) Mutation

Evolutionary computation tries to apply these biological concepts to automated problem solving. It is commonly assumed that evolutionary computing began back in 1948, when Alan Turing, who is considered to be the father of computer science and artificial intelligence, wrote an essay while he was working on the Automatic Computing Engine. In his work, Turing stated that *"If we are trying to produce an*

intelligent machine, and are following the human model as closely as we can we should begin with a machine with very little capacity to carry out elaborate operations or to react in a disciplined manner to orders (taking the form of interference). Then by applying appropriate interference, mimicking education, we should hope to modify the machine until it could be relied on to produce definite reactions to certain commands. This would be the beginning of the process." [1], thus introducing the idea that artificial evolution can be performed on a machine. However, it was in the 1960s when this concept was deeply explored. There were three different research lines, clearly separated: evolutionary programming (Fogel, Owens and Walsh), genetic algorithms (Holland) and evolution strategies (Rechenberg and Schwefel). Both evolutionary programming and genetic algorithms were developed in the USA, whereas evolution strategies were developed in Germany. Another research line called genetic programming appeared in the 1990s. In the last few years, all these algorithms have been considered subareas of what is known as evolutionary computing or evolutionary algorithms, thus ending the traditional separation between them.

II. Evolutionary Algorithms

An evolutionary algorithm can be seen not only as an optimization algorithm (in which every new solution is closer to the optimal one), but also as a process of adaptation (the environment selects the best solutions. i.e. the ones that are best adapted to its conditions). In order to measure this, it is absolutely necessary to define what is called fitness function, which gives an idea of how good a solution is.

The underlying theory can be expressed as follows: given a fixed population (individuals), a new set of candidates (i.e. possible solutions) is generated from some of the best elements by recombination and/or mutation. Afterwards, the whole population is evaluated in terms of the fitness function, and then the best individuals are allowed to pass to the next generation. The pseudocode of a generic evolutionary algorithm is shown in Table II.1.

```
INITIALIZE_POPULATION();
EVALUATE_FITNESS();
while(CONTINUE_ITERATING)
{
  PARENT_SELECTION();
  RECOMBINATION();
  MUTATION();
  EVALUATE_FITNESS();
  SURVIVOR_SELECTION();
  set CONTINUE_ITERATING;
}
```

Table II.1. Generic evolutionary algorithm pseudocode

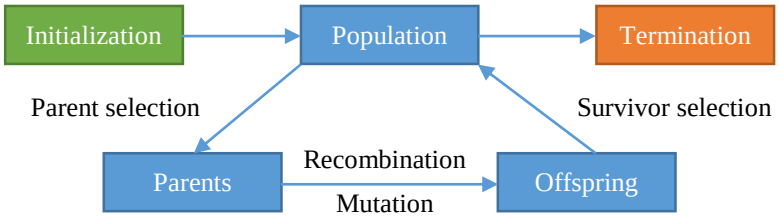


Fig. II.1. Evolutionary algorithm flowchart

In Fig. II.1, the generic evolutionary algorithm flowchart is shown, emphasizing the iterative process that takes place.

The initialization process is usually random, i.e. the first population is generated randomly. The termination condition can be set according to different criteria: limit of generations reached, fitness variation below user-defined bounds (diversity loss), fitness close to an acceptable value, etc.

II.1 Components

There are some elements that must be taken into account when dealing with evolutionary algorithms. These are the main components:

- Representation
- Fitness function
- Genetic operators
- Selection operators

II.1.1 Representation

This is a very important element in any evolutionary algorithm definition. Each individual has to be uniquely defined by its representation. Looking back to the biological systems, an analogy can be established between genotype and representation. Every candidate solution would be determined by a set of genes (the same way phenotype was determined by genotype).

Representation				**Individual**
Genotype				Phenotype
1	0	0	1	9
1	0	0	1	-7

Table II.2. Representation examples

Two representation examples have been provided in Table II.2. Although both have the same genotype, the phenotype is completely different. In the first example the

genotype encodes an unsigned integer, whereas in the second it is used to encode a two's complement signed integer. Therefore, it is extremely important to adopt a good representation scheme, as well as to maintain it throughout the whole algorithm, because the rest of the operations will be based on it.

In evolutionary algorithms terminology, the genotype or representation is usually referred to as chromosome, i.e. a set of characteristics (genes) that define the individual.

II.1.2 Fitness Function

The fitness function is also called evaluation function, and it is an expression used to measure the quality of a solution, i.e. how close to the theoretical best achievable solution the current one is.

Given that evolutionary algorithms are often used to solve optimization problems, sometimes the fitness function is referred to as objective function. It is important to keep in mind that the definition of a good fitness function, along with a good representation scheme, is the root of a well-designed and effective evolutionary algorithm. For instance, if the problem is to find the value within an interval whose square value is higher, a good choice for the fitness function would be $f(x) = x^2$.

II.1.3 Genetic Operators

A genetic operator can be defined as a mechanism whose function is to introduce diversity in any given population, generating new candidate solutions. Trying to replicate natural (biological) systems, two different operations have been proposed: on the one hand crossover; on the other hand mutation. The former approach uses two individuals (the so-called parents) and generates two children by recombining genetic characteristics from each parent. Therefore, any children has features inherited from both its parents, and the best genes can be transferred from a generation to the next one. The latter uses only one element to generate a new individual (another child) by changing one or even more of the genetic traits of the parent. With this mechanism new genes, which could provide better adaptation levels, appear.

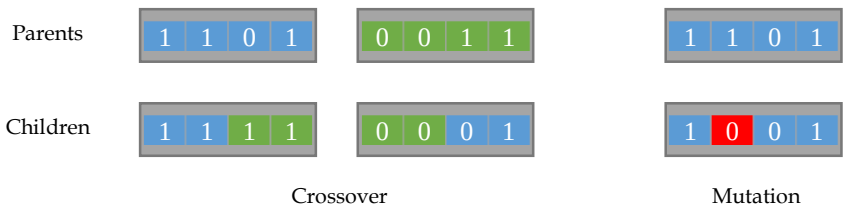


Fig. II.2. Genetic operators over a binary representation (chromosome)

Differences between both genetic operators can be seen in Fig. II.2. On the left side, a crossover operation is performed. Note that each child has half the genes from each parent, thus merging their genetic information. However, only one parent is necessary when the genetic operator is mutation, as it is shown on the right side, where only one gene is changed within the whole chromosome (this particular gene has been highlighted in red in order to emphasize the change).

One of the most important characteristics of this genetic operators is randomness, i.e. they are stochastic processes. Mutation has to be a random process in order to introduce only non-biased changes in the population. Crossover depends strongly on random drawings to decide which part of the parents would be recombined and in which way.

Depending on the specific evolutionary algorithm, these genetic operators may or may not appear. It is a task for the designer to choose whether to use crossover, mutation or both operations. Further information regarding the different proposed algorithms would be provided in the following sections.

II.1.4 Selection Operators

In each iteration (generation), two selection processes take place. The first one is used so that the mating pool, i.e. the individuals that would be the parents, can be selected. The second selection process is in charge of discarding those individuals that are not well-suited for the environment in which they are (this takes us back to the concept of natural selection).

Parent selection is usually done using a stochastic process, in order to allow that even weaker individuals can be promoted to the parent status. The reason for this process to be based upon random drawings is simple: it prevents from getting stuck at a local optimum. Therefore, every individual has a chance to become a parent, even though the probabilities are not the same (stronger individuals, i.e. with higher fitness values, are more likely to be chosen than weaker individuals).

As opposed to parent selection (stochastic process), survivor selection is usually a deterministic process. There are a lot of different possible implementations such as ordering the individuals according to their fitness values and selecting as survivors the top segment, or using age criteria: only the children survive, individuals whose fitness is weighted with the number of generations they have been alive, etc.

Selection operators are the algorithmic component that allows good features, which have been generated through genetic operations, to be transferred from one generation to the next ones (note that genetic transference was one of the basic foundations of evolutionary progress, and that it is indeed carried out in this particular step in the evolutionary algorithm).

II.2 Characteristics

Evolutionary algorithms provide us with good problem-solving capabilities. Moreover, the solutions obtained using these techniques are (in most of the cases, but not in all of them) by far better than the ones which could result from a random search, even though evolutionary algorithms are highly dependent on random processes.

Evolutionary algorithms are also commonly thought of as generic problem solvers. The same algorithm could be used to solve problems that are not related at all. The reason is simple (and has already been explained): the actual solutions (phenotype) are encoded in each chromosome (genotype). Therefore, if two non-related problems can be expressed using the same representation and their fitness can be evaluated in terms of the same fitness function, the algorithm is well-suited for both computations. However, these algorithms have a drawback that is worth noticing: the more generic the problem solver is (i.e. the wider the range of problems it can be applied to), the worse the solutions are. This means that evolutionary algorithms would lead to good solutions, but specific problem solvers would end up finding better results. Therefore, a tradeoff between the number of problems the algorithm can solve and the quality of the solutions appears. This tradeoff is shown in Fig. II.3. Nevertheless, decreases in the quality of the solution caused by evolutionary computing are negligible comparing with the flexibility it entails, and this is the reason why evolutionary computing is widely spread nowadays.

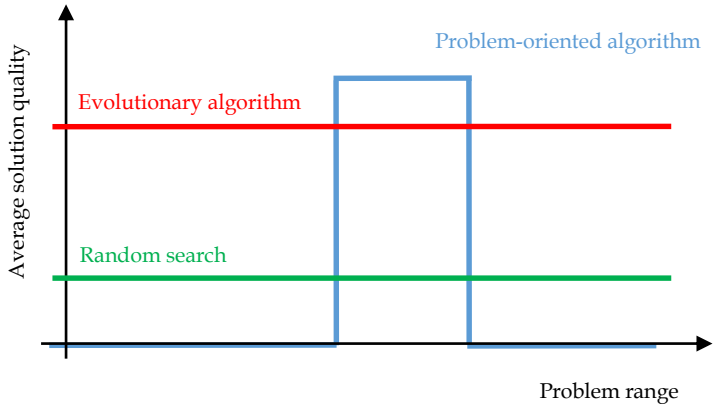


Fig. II.3. Comparison between different problem-solving algorithms

How does an evolutionary algorithm work? Generally speaking, the working cycle of any of the different approaches consists of two stages. In the first stage, the algorithm searches the solution space, starting from the random initialization values. Once the algorithm has found what could be considered as a good solution, the second stage starts. In this step, the algorithm tries to improve the current solution. These two stages are clearly differentiated if analyzed on a time vs. improvement basis. The first stage

shows large variations in fitness values in a few generations, whereas in the second stage the changes are almost insignificant. This temporal analysis becomes particularly useful when defining some termination conditions, e.g. a good termination condition based on the execution time would be to stop when the optimization process has reached this second stage.

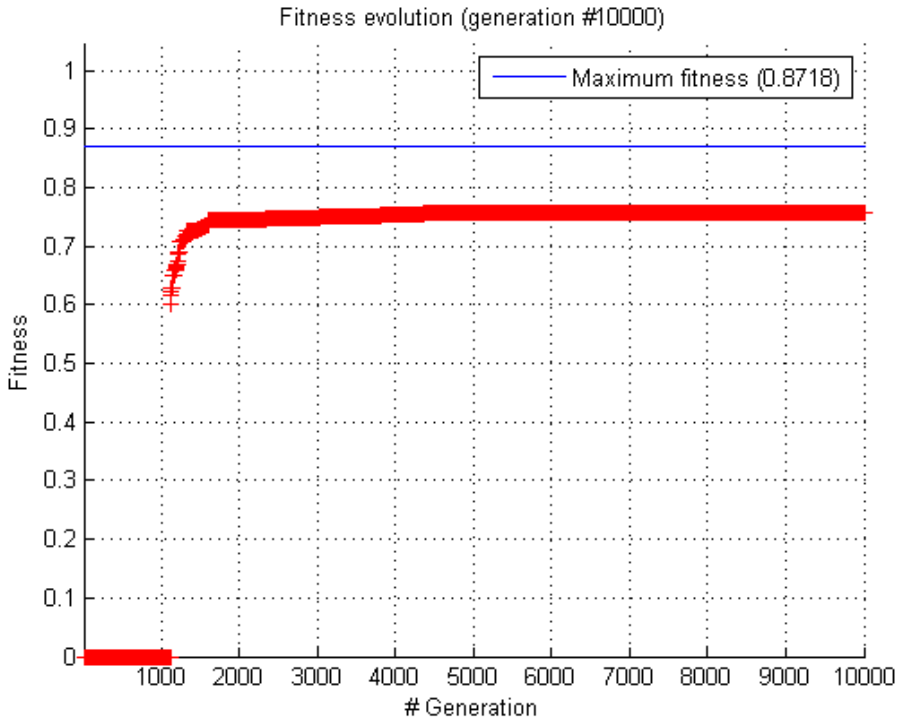


Fig. II.4. Fitness evolution vs. number of generations

Fig. II.4 shows a real example of an optimization process using a genetic algorithm. There is a huge increase in the fitness value from generation #1200 to approximately generation #2000. This corresponds to the aforementioned first stage of the evolutionary search. After generation #2000, the algorithm seems to stop its search (the fitness gets stuck in a plateau, especially after generation #5000), due to the fact that better solutions are more and more unlikely to appear.

In the forthcoming sections, some of the most used examples of evolutionary algorithms would be explored. First, the four traditional lines (i.e. genetic algorithms, evolution strategies, evolutionary programming and genetic programming) and then, a set of new ideas that have been developed all over the years. The main lines would be presented in their most common approaches, which means that subtle variations might be found in the literature.

II.3 Genetic Algorithms

Genetic algorithms are said to be the most extended form of evolutionary algorithms. Commonly used in optimization problems, these algorithms use numeric strings as representation, usually in the form of binary or integer arrays. Although both crossover and mutation are used, the former is the main operator. The parent selection scheme uses a fitness-biased random approach. On the contrary, the survivor selection scheme is usually generational (only the offspring survives, thus replacing the whole previous population) or deterministic (always taking into account fitness criteria).

II.4 Evolution Strategies

Evolution strategies try to take advantage of a very important feature: self-adaptive capabilities. In order to achieve this goal, some algorithmic parameters are evolved along with the solutions (these parameters could even be included as part of the genes that constitute a chromosome). Each chromosome is usually represented as an array of real numbers. Evolution is mutation-based in almost every case, but recombination can also appear as intermediate recombination (i.e. averaging the genes of the parents), or discrete recombination (i.e. selecting randomly as a child one of the parents). Each individual within the population has the same likelihood of being selected as a parent (no fitness-biased criteria appear in this approach), and the surviving population can be generated (always as a deterministic process) using only the offspring or including the previous population to the offspring. The former strategy seems to provide better results, because it avoids the memory effect, allowing transitions from local optima.

II.5 Evolutionary Programming

Evolutionary programming is really useful when the target problem is to optimize a fixed program structure which has some parameters that can be changed. Originally developed to generate artificial intelligence (emulating learning processes), evolutionary programming considers adaptation and environment prediction must-have features. This algorithm is only based on mutation, generating one child from each of the individuals, i.e. every individual is considered a parent. Survivors are selected randomly from the initial population plus the offspring.

The traditional approach used in evolutionary programming, which illustrates accurately the underlying concepts of this field, was to evolve a predictor system represented as a finite state machine or FSM. As stated before, the architectural characteristics of the systems are fixed, but some parameters can be changed: number of states, number of inputs, number of outputs, adding or deleting transitions between states, changing the initial state, etc. However, this classical example is no longer considered as standard evolutionary programming, due to the fact that this algorithm has been mostly used to optimize real-valued parameter vectors since the 1990s.

II.6 Genetic Programming

Also focused on program optimization, the main difference between evolutionary programming and genetic programming is that the latter represents each program as a tree and changes its whole structure (as opposed to the former, in which only some parameters were changed). Both genetic operators can be used in order to modify branches in the trees (recombining by swapping, or mutating by adding or deleting some of the leaves). Parent selection follows the same scheme as in genetic algorithms (fitness-biased random selection), and survivors are selected using generational criteria (i.e. only the offspring survives).

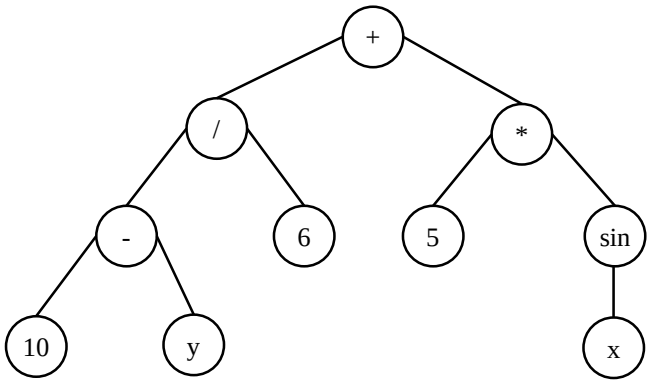


Fig. II.5. Genetic programming representation: tree structure

In Fig. II.5, a typical chromosome in genetic programming is shown. The genotype encodes the function $f(x, y) = \left(\frac{10-y}{6}\right) + (5 \cdot \sin(x))$.

II.7 Other Approaches

Evolutionary computing has been a developing field ever since it appeared. There is a vast range of different implementations and algorithms, but for the sake of convenience only a few of them will be presented here.

When dealing with complex problems which can be divided in simpler subproblems, memetic algorithms may be a wise choice. These algorithms are hybrid, for they combine evolutionary processes and problem-solving knowledge (heuristics). Given that a new component is introduced, a new stage is also necessary: the learning stage, in which the algorithm gathers every piece of knowledge that is available. Memetic algorithms can be found in the literature as hybrid genetic algorithms, Baldwinian evolutionary algorithms or Lamarckian evolutionary algorithms, but all of them have in common the addition of one or more local search (i.e. better solutions in the neighborhood of a known one) stages to the traditional evolutionary algorithm.

Coevolution seems to be the most appealing concept regarding evolutionary computing, and can be implemented as a cooperative algorithm or as a competitive algorithm. Either it is one or the other, the population has different species. In cooperative coevolution, each species represents a part of the problem and cooperate in order to come with a solution of a larger problem. As opposed to that, competitive coevolution is based on individuals gaining fitness at each other's expense, i.e. the species fight against each other.

The last example of evolutionary computation techniques is called interactive evolution, and turns out to be very useful when dealing with problems in which there is no such thing as a clearly defined fitness function. Subjective opinions have a strong influence in the selection process. Therefore, biased external interference is what best defines interactive evolution, even though this interference might or might not be direct (e.g. deciding whether an individual can survive or not).

III. Current Research Lines

Evolutionary computation is an enormous research field. Therefore, we would only present those works closely related to particle filtering (i.e. particle filters which have been enhanced with evolutionary computing).

Particle filtering and evolutionary algorithms, especially genetic algorithms, have conceptual similarities. These similarities have been studied and documented for a long time. For instance, in [2] a modified particle filter is presented. The author uses genetic operators such as crossover and mutation to implement the prediction stage of the filter, describing the whole algorithmic implementation of what he calls Genetic Algorithm Filter. The connection between particle filtering (Monte Carlo simulations) and Bayesian inference and their application to evolutionary environments has also been studied. In [3], the foundations for Bayesian evolutionary computation are presented. The idea is to guide the evolutionary process using Bayes' rule, since it is stated in the paper that the most probable solution could be considered the best one. However, this is not state-of-the-art research.

In the last few years there has been a significantly higher interest in using evolutionary computing concepts in particle filtering. All the conducted research is focused on reducing one of the two main problems a suboptimal resampling strategy generates: sample impoverishment (refer to the Particle Filtering chapter for more information on this particular issue). Evolutionary algorithms, as stated in previous sections, use genetic operators in order to transfer good genes and to introduce genetic diversity. This last feature is the key to understand why evolutionary computing is suitable for solving sample impoverishment (i.e. diversity loss) problems. Hence, in [4] another evolutionary approach is introduced in order to mitigate those effects. At the very beginning of the paper, the authors state that previous works in the field had not fully exploited the advantages of evolutionary computing. Therefore, they propose

introducing genetic operators right after performing the importance sampling stage and immediately before the resampling stage, which uses a parameter (effective number of particles) in order to decide whether to perform resampling or not. In this paper it is also shown that the mutation operator provides better dynamic response when the state jumps abruptly.

Other research lines use parallel distributed filters, i.e. with several subpopulations evolving at the same time. In [5], the authors use these subpopulations to perform genetic operations and, from time to time, migrate the best individuals from one subpopulation to the others so that the best genetic traits can be shared. This also leads to an improvement in global optimum search. Moreover, the concept of genetic resampling stage is introduced in this paper for the first time. Nevertheless, the paper provides only simulation results.

More complex examples can also be found in the literature: a hybrid evolutionary particle filter is presented in [6]. This hybrid approach tries to take advantage of both genetic algorithms (to maintain particle diversity) and particle swarm optimization (to optimize the final particle distribution). Furthermore, the algorithm presented has parallel features, thus reducing computation times. The strategy presented divides the population in two groups, and then performs the specific operations that are required: in one group, a genetic algorithm; in the other, particle swarm optimization. Before the next time step, a migration operation is performed (to share genetic information, as in previous examples).

Real-world applications include object tracking, as in [7]. Another evolutionary approach is presented in order to deal with sample impoverishment. In this case, the evolutionary resampling stage may or may not take part in the estimation loop. The decision is made based on the aforementioned parameter, i.e. the effective number of particles. The algorithm presented provides good results but it is not accurate in occlusion tracking sequences.

Recent studies with differential evolution have been conducted in order to reduce significantly sample sizes [8]. The differential evolution algorithm divides the particles in three different groups: the first group would undergo crossover; the second group would undergo mutation; the last group would not suffer any modification. One important fact about this work is that the evolutionary stage takes place at the very beginning of the process, and it is immediately followed by the importance sampling stage.

In conclusion, evolutionary computation has found a vast field of application in which optimization is not the main concern. The characteristic properties of the genetic operators make evolutionary algorithms a potential tool in particle filtering, since they are able to reduce to almost negligible values both problems: particle degeneracy (it is avoided performing resampling) and sample impoverishment (it is avoided introducing genetic diversity in the particle population).

IV. References

[1] Turing, A.: Intelligent Machinery. In: D. Ince (ed.) Collected Works of A. M. Turing: Mechanical Intelligence. Elsevier Science (1992)

[2] T. Higuchi, "Monte Carlo filter using the genetic algorithm operators," *Journal of Statistical Computation and Simulation*, vol. 59, pp. 1-23, 1997

[3] Byoung-Tak Zhang, "A Bayesian framework for evolutionary computation," *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on* , vol.1, no., pp.,728 Vol. 1, 1999

[4] Seongkeun Park; Jae Pil Hwang; Euntai Kim; Hyung-Jin Kang, "A New Evolutionary Particle Filter for the Prevention of Sample Impoverishment," *Evolutionary Computation, IEEE Transactions on* , vol.13, no.4, pp.801,809, Aug. 2009

[5] Cong Li; Qin Honglei; Xing Juhong, "Distributed genetic resampling particle filter," *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on* , vol.2, no., pp.V2-32,V2-37, 20-22 Aug. 2010

[6] Jialong Zhang; Tien-Szu Pan; Jeng-Shyang Pan, "A Parallel Hybrid Evolutionary Particle Filter for Nonlinear State Estimation," *Robot, Vision and Signal Processing (RVSP), 2011 First International Conference on* , vol., no., pp.308,312, 21-23 Nov. 2011

[7] Wei Leong Khong; Wei Yeang Kow; Yit Kwong Chin; Mei Yeen Choong; Teo, K.T.K., "Enhancement of Particle Filter Resampling in Vehicle Tracking Via Genetic Algorithm," *Computer Modeling and Simulation (EMS), 2012 Sixth UKSim/AMSS European Symposium on* , vol., no., pp.243,248, 14-16 Nov. 2012

[8] Nyirarugira, C.; Tae Yong Kim, "Adaptive evolutional strategy of particle filter for real time object tracking," *Consumer Electronics (ICCE), 2013 IEEE International Conference on* , vol., no., pp.35,36, 11-14 Jan. 2013

Evolutionary Particle Filter

I. Introduction

In this chapter, the proposed architecture is presented. The Evolutionary Particle Filter has been designed for movement estimation applications. In this application, the state variables are four: x-axis position, y-axis position, x-axis velocity and y-axis velocity.

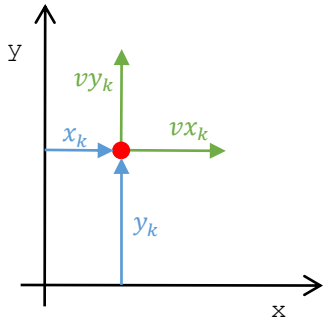


Fig. I.1. State variables

As far as this implementation is concerned, it has been assumed that the movement does not have any acceleration at all. The only allowed changes in both velocity components are the ones caused by the addition of random noise. The motion equations can be then expressed as follows:

$$\begin{cases} x_k = x_{k-1} + T \cdot vx_{k-1} + w_k^x \\ y_k = y_{k-1} + T \cdot vy_{k-1} + w_k^y \\ vx_k = vx_{k-1} + w_k^{vx} \\ vy_k = vy_{k-1} + w_k^{vy} \end{cases}$$

This motion model can also be expressed in matrix-based notation:

$$\boldsymbol{x_k} = A \cdot \boldsymbol{x_{k-1}} + \boldsymbol{w_{k-1}}$$

$$A = \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{x_k} = \begin{bmatrix} x_k \\ y_k \\ vx_k \\ vy_k \end{bmatrix} \qquad \boldsymbol{w_k} = \begin{bmatrix} w_k^x \\ w_k^y \\ w_k^{vx} \\ w_k^{vy} \end{bmatrix} \sim \begin{bmatrix} N(\mu_x, \sigma_x) \\ N(\mu_y, \sigma_y) \\ N(\mu_{vx}, \sigma_{vx}) \\ N(\mu_{vy}, \sigma_{vy}) \end{bmatrix}$$

As opposed to [1], this model does not perform any estimations regarding the current velocity in each axis. Therefore, a significant reduction in terms of memory resources is achieved, since it is no longer necessary to store previous position values.

However, the state space is not observable. The only state variables that are measurable are both x-axis and y-axis positions. Therefore, the measurement model can be represented (in matrix notation) as:

$$\boldsymbol{z_k} = H \cdot \boldsymbol{x_k}$$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad \boldsymbol{z_k} = \begin{bmatrix} zx_k \\ zy_k \end{bmatrix}$$

The most important advantage this proposal has is that it is highly application-independent. This estimation engine can be used to track a wide range of different elements, as long as their state can be expressed as the aforementioned $\boldsymbol{x_k}$ array. This feature makes the system incredibly suitable for a reconfigurable platform. By using different (reconfigurable) preprocessing stages, the Evolutionary Particle Filter can track colored objects, corners, shapes, etc.

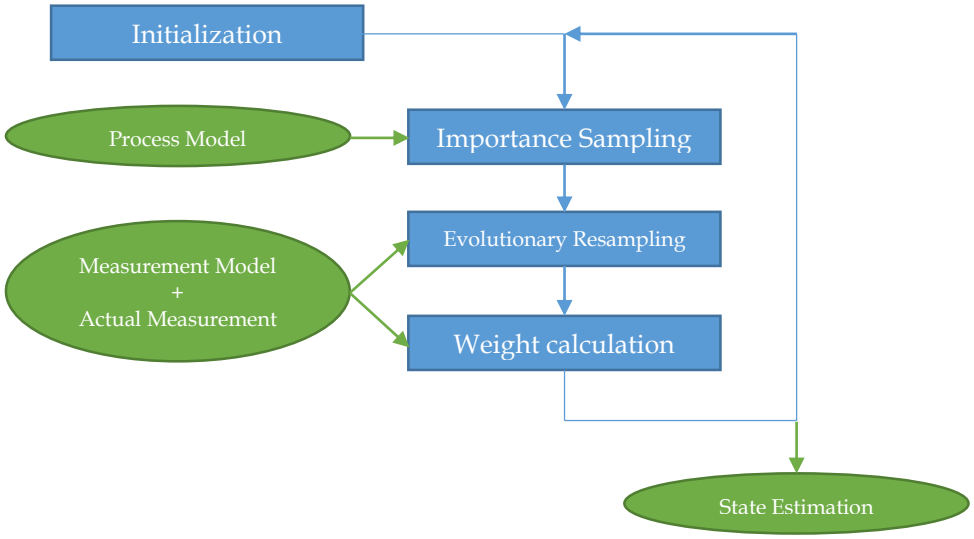


Fig. I.2. Evolutionary Particle Filter flowchart

The modified flowchart of the Evolutionary Particle Filter is shown in Fig. I.2. Note that this flowchart is based on the basic particle filter but adding an extra stage, which is in charge of performing resampling using an evolutionary algorithm.

The rest of this chapter is organized as follows: first, the algorithmic implementation of the evolutionary resampling stage will be presented and discussed. Then, the hardware-based modules will be explained in detail.

II. Evolutionary Resampling Stage

Placed right after the Importance Sampling process, i.e. the process model update, and immediately before weight calculation and state estimation, the evolutionary resampling stage constitutes itself the distinctive element of the EPF algorithm. Although some approaches can be found in the literature (as in the aforementioned proposal of [2]), some modifications have been introduced and will be discussed in the forthcoming paragraphs.

A genetic algorithm has been selected, due to the flexibility of introducing both genetic operators (crossover and mutation) and to the fact that chromosomes will be represented as integer arrays. The target application has some timing constraints (e.g. frames per second in the camera); therefore, the termination condition is set to be reaching a fixed number of generations.

II.1 Crossover

Arithmetic crossover will be used as recombination operator. Crossover takes place whenever a random number $p \sim U(0,1)$ satisfies $p < p_c$, being p_c the crossover probability. Arithmetic crossover can be expressed in terms of the following equations:

$$\begin{cases} x_k^a = \alpha \cdot x_k^i + (1-\alpha) \cdot x_k^j \\ x_k^b = \alpha \cdot x_k^j + (1-\alpha) \cdot x_k^i \end{cases}$$

where x_k^a and x_k^b are the children, i.e. the offspring, and can be obtained weighting the states of the parents x_k^i and x_k^j. The subscripts (k) represent the current time step in which the system is, whereas the superscripts are the individual identifiers (*a, b* for the children; *i, j* for the parents). The weight factor is a random number drawn from a uniform distribution $\alpha \sim U(0,1)$.

II.2 Mutation

Mutations should occur whenever a random number $p \sim U(0,1)$ satisfies $p < p_m$, being p_m the mutation probability. In this proposal, two mutation operations can be performed. If $p \geq p_m \cdot r_m$, with $r_m = \frac{p_{random\ placement}}{p_{local\ search}}$ being the mutation ratio, a local search mutation operator is used. Otherwise, the mutation operator generates a random child over the whole search space.

Local search mutation adds Gaussian noise, i.e. random numbers, to the parents' state variables, as can be seen in the following equation:

$$x_k^c = x_k^i + \delta_k$$

Random placement mutation generates a random child using this expression:

$$x_k^c = x_{min} + \beta \cdot (x_{max} - x_{min})$$

In these expressions x_k^c is the child, x_k^i the parent (only in local search mutation), $\delta_k \sim N(\mu, \sigma)$ (only in local search mutation), x_{min} and x_{max} the limits of the search space (only in random placement mutation), and $\beta \sim U(0,1)$ (only in random placement mutation).

Each of these two mutation operations has a defined purpose. On the one hand, local search helps reaching close optimal points. On the other hand, random placement promotes global optimization (not just local), making possible for the population to jump to unexplored regions in the state space and evolve there. Hence, better solutions might be obtained.

II.3 Selection

The chosen selection algorithm is the so-called Stochastic Universal Sampling (SUS). This algorithm appears as a development of Fitness Proportionate Selection (FPS), also known as roulette wheel algorithm, even though the drawing of random numbers is carried out in a different manner. Considering that n_{sel} elements have to be selected, FPS draws n_{sel} different uniform random numbers, whereas SUS only draws one using a uniform distribution.

$$r \sim U\left(0, \frac{\sum_i f_i}{n_{sel}}\right)$$

In the last expression, f_i is the fitness value of the individual x_k^i, n_{sel} is the number of individuals that have to be selected, and r is the value which is drawn.

How does Stochastic Universal Sampling work? First, individuals have to be sorted according to their fitness value.

$$f = [f_{n-1} \quad f_{n-2} \quad \cdots \quad f_1 \quad f_0]$$
$$f_0 \geq f_1 \geq \cdots \geq f_{n-2} \geq f_{n-1}$$

Then, the cumulative fitness vector is computed starting with the individual that has the highest fitness value.

$$q_k = \sum_{i=0}^{k} f_i$$

Once the random number r has been drawn, the individual k is selected if $r < q_k$. Otherwise, r is incremented as follows:

$$r = r + \frac{\sum_i f_i}{n_{sel}}$$

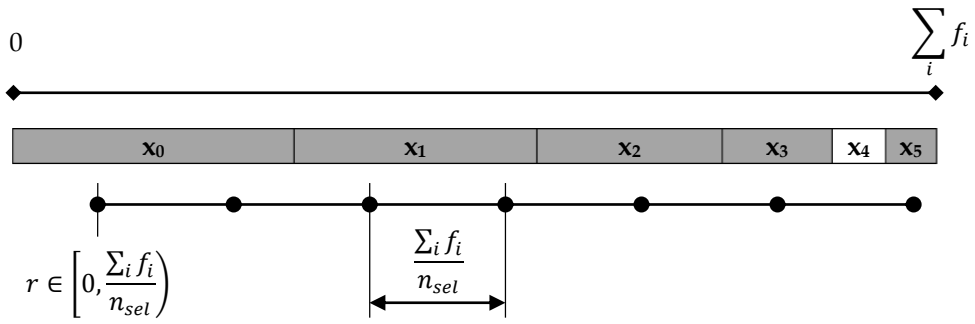


Fig. II.1. Stochastic universal sampling

Therefore, the idea is that every selected individual is equidistant with the previous and the following element. A good example has been provided in Fig. II.1, where $n_{sel} = 7$. The first element is selected using the random number r, and then the following elements are obtained adding a fixed increment, which depends on both the number of desired selected individuals and the maximum value of the cumulative fitness vector. In this particular example, the seven selected individuals are $[x_0, x_0, x_1, x_1, x_2, x_3, x_5]$. This selection algorithm has very important features, such as being unbiased. However, the most important characteristic is that with this selection strategy, even the weakest members of the population can be selected, as in the previous example. This also prevents the evolutionary process from remaining stuck at a local maximum.

```
i = 0;
j = 0;
inc = q[N-1]/N_SEL;
r = inc*(rand()/RAND_MAX);
while(i<N_SEL)
{
  if(r<=q[j])
  {
    selected[i] = j;
    r += inc;
    i++;
  }
  else
  {
    j++:
  }
}
```

Table II.1. Example code for Stochastic Universal Sampling selection

This selection scheme has been adopted in this work as both parent and survivor selection algorithm. The main reason for this decision is that implementing more than one selection algorithm might increase significantly resource utilization rates. Therefore, survivor selection will be stochastic in this proposal, rather than deterministic as in many examples in the literature.

To summarize, the proposed evolutionary algorithm is a genetic algorithm with Stochastic Universal Sampling selection, arithmetic recombination and both local and global mutation schemes, and its flowchart can be seen in Fig. II.2.

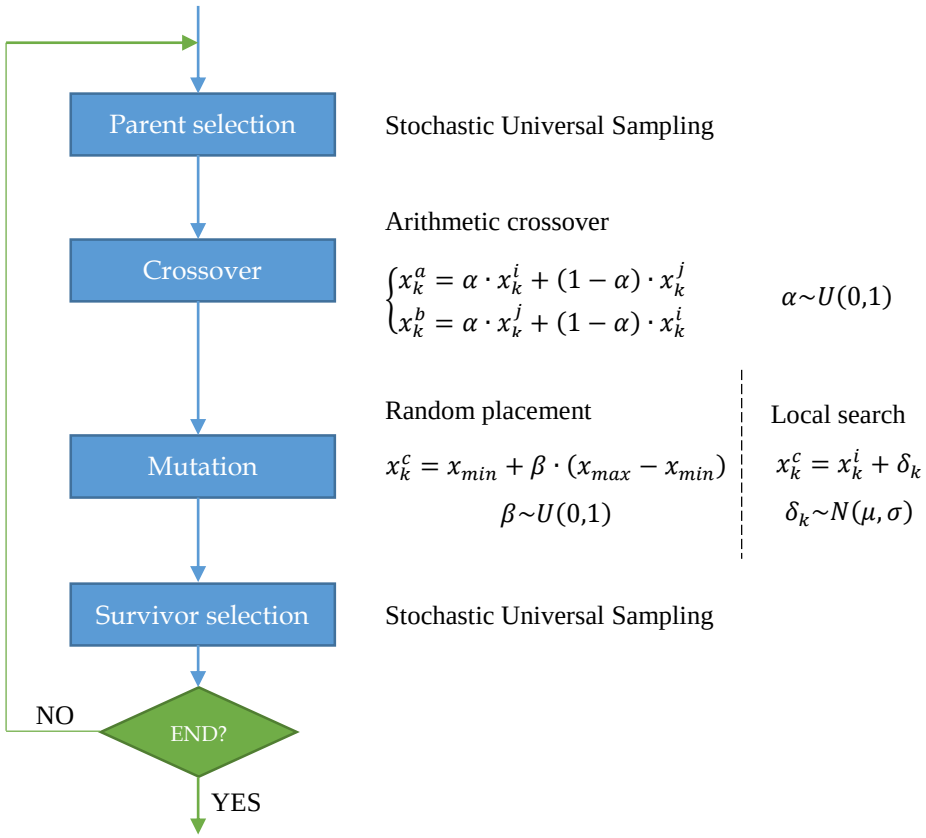


Fig. II.2. Evolutionary resampling flowchart

III. Hardware Architecture

The Evolutionary Particle Filter has been implemented as an IP (Intellectual Property) core, i.e. a hardware module. A modular approach has been adopted so as to make the design stage easier and to favor reusability. Moreover, it could allow future reconfiguration within the core itself.

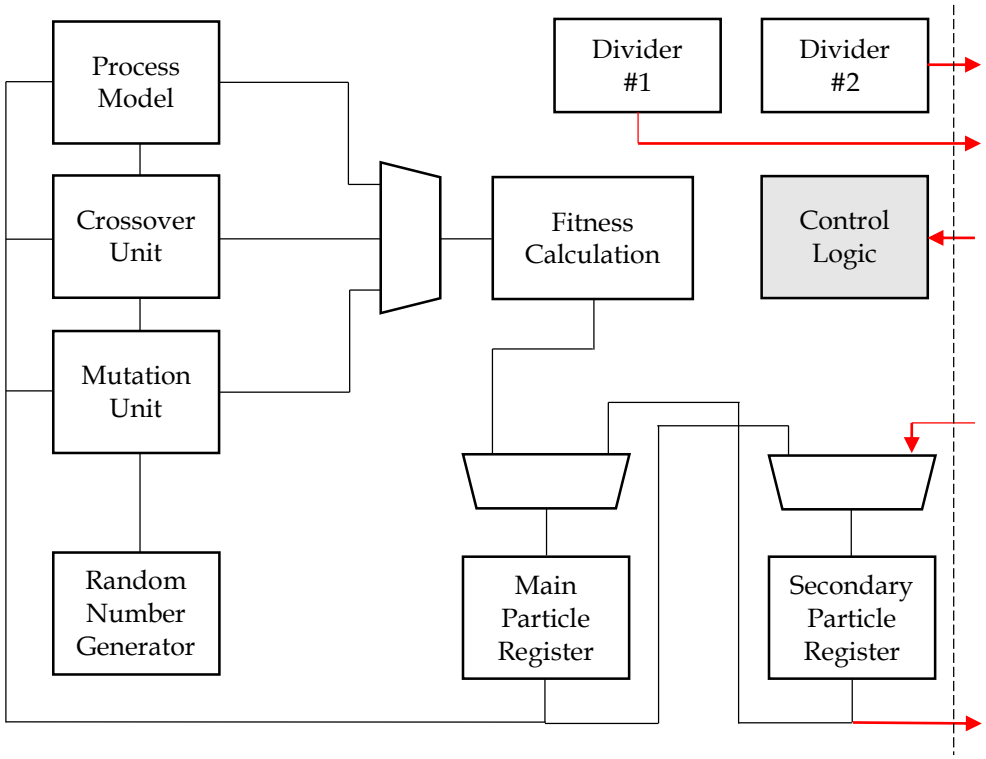


Fig. III.1. Evolutionary Particle Filter block diagram

The proposed modular architecture can be seen in Fig. III.1. Some control signals and data connections have been simplified in the diagram in order to avoid excessive complexity. The signals highlighted in red represent the external connections of the hardware module, i.e. the interface with the top system. The main modules of the Evolutionary Particle Filter system are:

- Random number generator.
- Fitness calculation.
- Particle registers.
- Process model.
- Crossover unit.
- Mutation unit.
- Dividers.
- Control logic.

These blocks will be presented in detail in the forthcoming sections of this chapter.

Hardware designs have some limitations in terms of arithmetic operations. Designers have to decide whether to use integer, fixed-point or even floating-point arithmetic. There is an obvious tradeoff between precision and resource consumption (or time): the more precise the operations have to be, the more complex the resulting system is. Focusing on the Evolutionary Particle Filter, it seems clear that estimation and prediction tasks require accurate computations, at least to some extent. However, extra complexity in the design is not desirable. For this reason, fixed-point arithmetic has been selected as the most convenient implementation strategy in this thesis.

Fixed-point arithmetic uses real number representations in which the decimal part, i.e. those digits after the radix point, and the integer part have a fixed size. For instance, in Fig. III.2, n_d would represent the size, in bits, of the decimal part, and n_i would represent the size of the integer part.

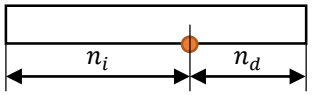


Fig. III.2. Fixed-point number representation

Having a fixed number of bits for both integer and decimal part generates some problems. The first problem a designer might face is overflow or underflow phenomena, which occur when the result of an operation cannot be represented with the given resolution, i.e. the number of bits, the system has. For example, the result of $24 + 12$ with an unsigned integer resolution of 5 bits would be 4 instead of 36.

Another important problem of using fixed-point arithmetic is that not all the numbers within an interval can be represented, since the less significant bit determines the minimum increment. Therefore, the resolution is finite. For example, if the decimal part resolution were 4 bits, the minimum increment would be 0.625.

In this design, numbers are represented as 19-bit integers, unless otherwise specified: 8-bit decimal part, 10-bit integer part and 1 bit to represent the sign (2's complement). This gives an overall representation range from -1024.0 to 1023.99609375.

III.1 Random Number Generator

Stochastic processes play a very important role in the Evolutionary Particle Filter. On the one hand, normal-distributed random numbers are used in some stages (e.g. process model update, or importance sampling). On the other hand, the evolutionary algorithm uses uniform random numbers to decide whether to perform a genetic operation or not. Therefore, the random number generator is an essential part of the design.

The target application is not as demanding as a cyphering system would be. Therefore, a pseudorandom number generator (PRNG) is enough to meet the requirements. Moreover, designing a true random number generator (TRNG) would be a hard task, since they have strong dependencies on physical phenomena that are really hard to validate through simulation.

Uniform distributions of pseudorandom numbers can be generated with a linear-feedback shift register, also known as LFSR. LFSRs are built using a shift register in which the input bit is a linear combination of some of the register bits, i.e. a linear combination of the register state. This linear combination can be done, for example, with a XOR gate (actually, this is by far the most used alternative), and can be expressed in terms of a feedback polynomial.

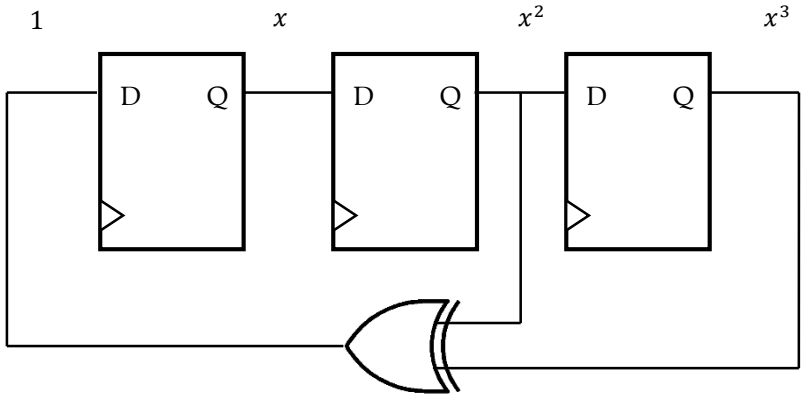


Fig. III.3. 3-bit LFSR

A 3-bit LFSR is shown in Fig. III.3. The feedback polynomial for this specific LFSR is $p(x) = x^3 + x^2 + 1$. Note that in the feedback polynomial the bits that are combined appear as x^t, being t the so-called tap.

LFSRs can be maximal-length only if there is an even number of taps, and if the whole set of taps is relatively prime, i.e. no common divisor to all taps exists. If these requirements are met, the LFSR will go through all possible states, except that in which all the bits are equal to zero. In maximal-length LFSRs, there are $2^n - 1$ possible states, being n the register size (i.e. the number of bits). If n is small, the sequence cannot be considered as random. However, if n is set to be relatively large, the outgoing sequence can be thought of as a random sequence. In [3], maximal-length LFSRs and their taps can be found. The LFSR used in this thesis is based upon that application note.

The initial value in a LFSR is called seed. This seed cannot be set to zero, since that is a non-return state (as stated in the previous paragraph). Once the LFSR has gone through all possible states, i.e. its period, the seed will appear again, and the whole sequence will be repeated.

Once the uniform-distributed random number generation has been solved using a LFSR, the next problem to address is the normal-distributed random number generation. Some hardware implementations have been proposed in the literature, for example in [4]. In this particular example, the Box-Muller method is used in order to generate the random samples. Complex mathematical operations, such as natural logarithms and square roots, are performed to obtain two different normal-distributed samples from two different uniform-distributed values. As a result, many resources are necessary to implement this solution.

In this thesis, an alternative solution is presented. This solution is based upon the central limit theorem and the outcome is a very simple architecture. In probability theory, the central limit theorem states that, if some conditions are met, the arithmetic average of a sufficiently large set of different independent random variables, each of them with known mean and standard deviation values, will be approximately normally distributed. A slight modification of this statement can be expressed as follows:

$$\chi_i \sim pdf(\mu, \sigma)$$

$$\sum_{i=1}^{n} \chi_i \sim N\left(n \cdot \mu, \sqrt{n} \cdot \sigma\right)$$

The random distributions χ_i can be arbitrarily chosen. Since the LFSR will produce a uniform-distributed random variable at its output port, it seems reasonable to choose uniform distributions. A uniform distribution in the interval (a, b) has the probability density function that is shown in Fig. III.4.

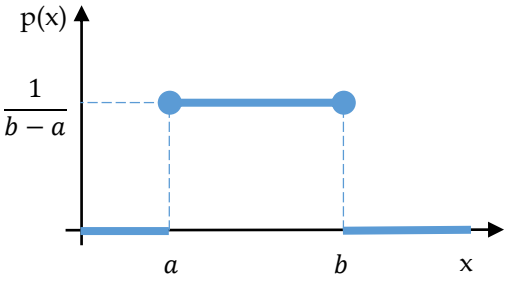


Fig. III.4. Uniform distribution probability density function

Moreover, the mean and the standard deviation can be computed using the following equations:

$$\chi_i \sim U(a, b)$$

$$\mu = \frac{a+b}{2} \qquad \sigma = \frac{b-a}{\sqrt{12}}$$

It is possible then to select $n = 12$ in order to save some computations, because the square roots disappear in the expression of the central limit theorem.

$$\sum_{i=1}^{12} \chi_i \sim N\left(12 \cdot \mu, \sqrt{12} \cdot \sigma\right) = N(6 \cdot (a+b), b-a)$$

Since twelve different uniform-distributed variables have to be added, a new problem arises: how to perform this mathematical operation.

- Parallel solution: using twelve different LFSRs and adding their output values. This solution is not feasible due to excessive resource consumption.
- Serial solution: using only one LFSR and registering twelve output values. Once the values have been registered, the addition is carried out. This solution is better than the previous one, but normal-distributed samples have a latency of twelve clock cycles.
- Correlated solution: this was the first proposal that reduced the latency to one clock cycle. However, it is not a good alternative, because the output values cannot be considered white noise, since they are correlated. The architecture used one LFSR and twelve different accumulators, each with an increasing delay of a clock cycle, as can be seen in Fig. III.5.

LFSR	0	1	2	3	4	5	6	7	8	9	10	11	12	13	…
Acc1	0	1	2	3	4	5	6	7	8	9	10	11			
Acc2		0	1	2	3	4	5	6	7	8	9	10	11		
Acc3			0	1	2	3	4	5	6	7	8	9	10	11	
…															

Fig. III.5. Correlated random number generation scheme (source of correlation has been highlighted)

All these solutions were discarded in the design stage because of the aforementioned reasons.

The proposed solution uses only a large LFSR and twelve adders. In the LFSR, the constraint $tap_{min} \geq 12 \cdot n$, where n is the size in bits of the uniform variables that are added, has to be met. This condition is expressed graphically in Fig. III.6.

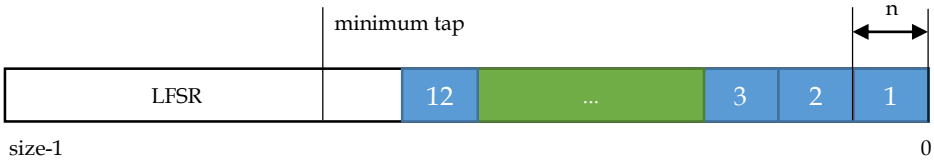


Fig. III.6. Normal-distributed random number generator proposal

With this structure, twelve different independent and identically distributed random numbers can be generated in only one clock cycle.

The final implementation consists of a maximal-length 111-bit LFSR, whose taps are 111 and 101. The VHDL module symbol is shown in Fig. III.7, and it has some parameters that are configurable.

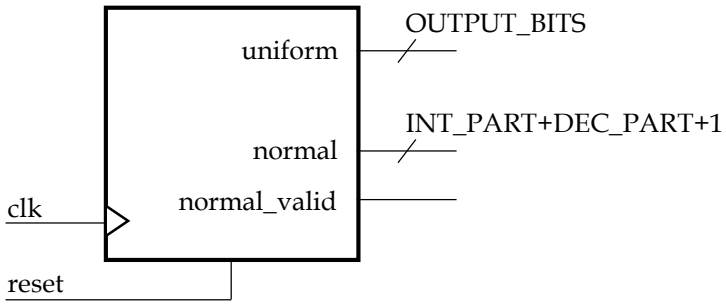


Fig. III.7. Random number generator symbol

- SEED: the initial value of the 111-bit LFSR.
- MEAN: mean value of the normal distribution.
- STD_DEV: standard deviation of the normal distribution.
- INT_PART: integer part size in bits.
- DEC_PART: decimal part size in bits.
- OUTPUT_BITS: uniform output size in bits.

The size of the twelve uniform-distributed numbers is DEC_PART. This size, along with the uniform output size appears in Fig. III.8. This architecture is compliant with the tap requirement, since $12 \cdot DEC\_PART = 12 \cdot 8 = 96 \leq 100 = tap_{min}$.

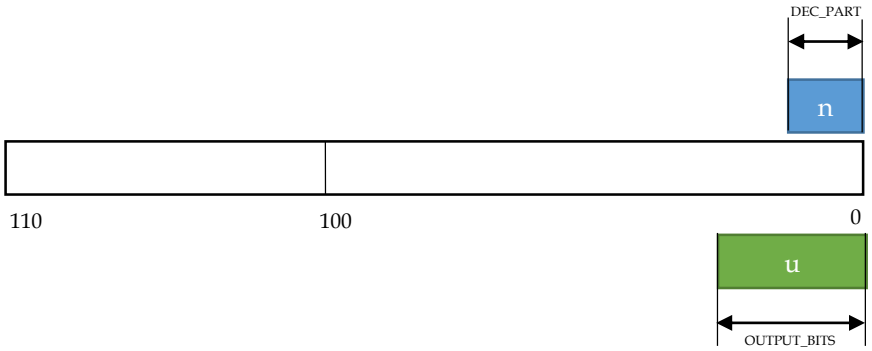


Fig. III.8.LFSR architecture

Given that the output values are not one-bit-sized, the LFSR operation has suffered some small variations that are shown in Fig. III.9.

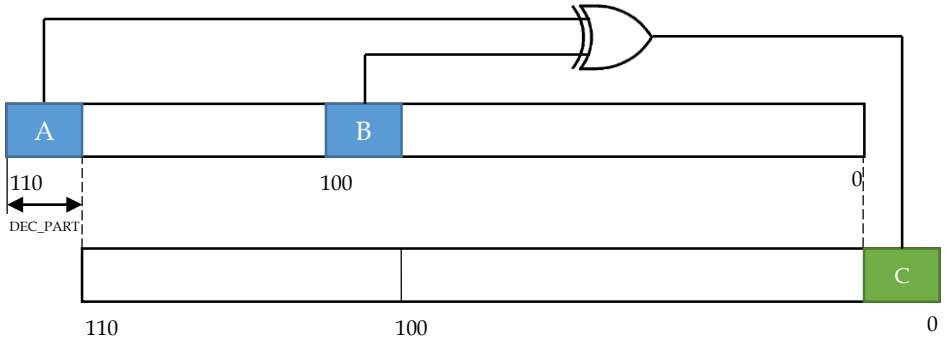


Fig. III.9. LFSR operation

The random number generator module provides three different outputs.

- **uniform**: uniform-distributed number with a resolution of OUTPUT_BITS bits. It is an unsigned integer that represents a real number in the interval (0,1).
- **normal**: normal-distributed number $n \sim N(MEAN, STD\_DEV)$ with a resolution of INT_PART+DEC_PART+1 bits. It is a signed (2's complement) integer that codes a fixed-point number with INT_PART bits representing the integer part and DEC_PART bits representing the decimal part. The extra bit is used to code the sign.
- **normal_valid**: control signal. It is one if the normal output represents a valid sample and zero if it does not. This signal has been included in the design in order to make it compatible with the other proposed random number generators (in which the latency was over one clock cycle), even though its value will always be one, since the latency of the proposed design is only one clock cycle.

The effective numeric values of the signals in this module, as well as their characteristics (i.e. attributes) are shown in Table III.1.

Name	Mode	Attributes	Minimum	Maximum	Increment
clk	in	rising	0	1	-
reset	in	high	0	1	-
uniform	out	unsigned	0	$1 - 2^{-OUTPUT\_BITS}$	$2^{-OUTPUT\_BITS}$
normal	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
normal_valid	out	-	0	1	-

Table III.1. RNG signal characteristics

III.2 Fitness Calculation

Weight calculation is a very important stage in particle filtering, and fitness calculation plays a similar role in evolutionary computation. In this thesis, these two operations have been combined into one single unit. The fitness calculation unit obtains the fitness, i.e. the weight, of any particle.

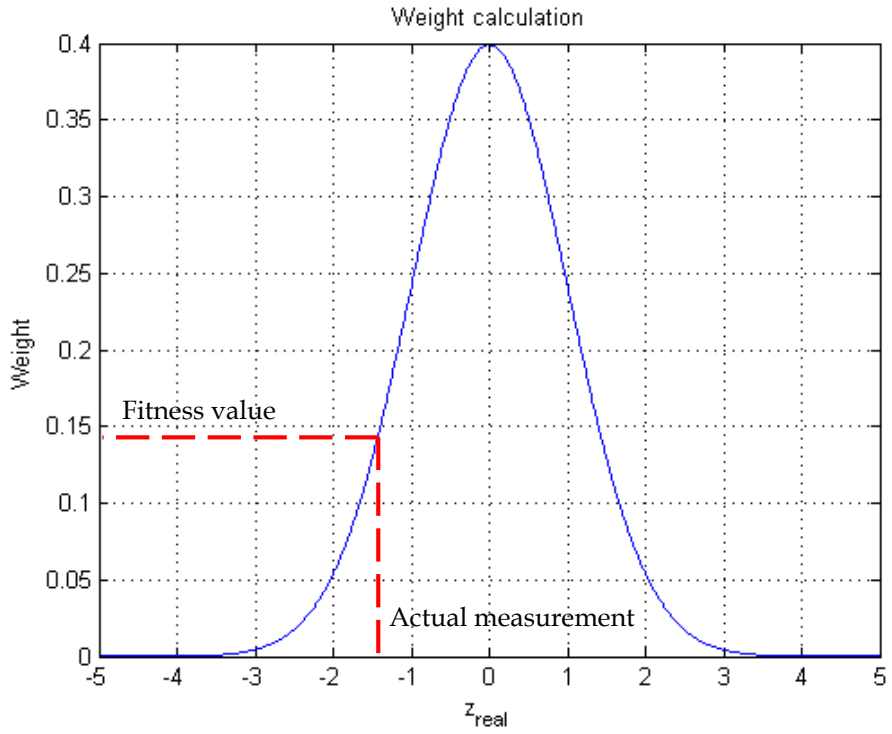


Fig. III.10. Fitness graphic calculation

A simple example on how to compute the weight of a particle has been provided in Fig. III.10. The measurement noise standard deviation is used to generate a normal distribution whose mean value is the predicted measurement, which is obtained after applying the measurement model to the particle current state. The particle weight is then calculated as the probability of obtaining the actual measurement in that normal distribution.

Since the proposed measurement model provides two-element vectors, the normal distribution around the mean values has two dimensions, i.e. it is multivariate.

$$w_k^i = f_i \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \qquad \boldsymbol{\mu} = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix} = \begin{bmatrix} \widehat{zx}_k^i \\ \widehat{zy}_k^i \end{bmatrix} = H \cdot \hat{\boldsymbol{x}}_{\boldsymbol{k}}^{\boldsymbol{i}} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \sigma_x & \sigma_{xy} \\ \sigma_{xy} & \sigma_y \end{bmatrix}$$

In the previous expressions, it is possible to see that the mean values are $\mu_x = \widehat{zx}_k^i$ and $\mu_y = \widehat{zy}_k^i$, which are the predicted measurements, i.e. the expected measurement values, obtained from each particle after applying the measurement model to the population ($\hat{\boldsymbol{x}}_{\boldsymbol{k}}^{\boldsymbol{i}}$ represents the predicted state of the particle *i* at time *k*). The covariance matrix $\boldsymbol{\Sigma}$ has constant values, and the state variables are considered independent, i.e. uncorrelated. Therefore, the covariance $\sigma_{xy} = 0$.

In order to compute the probability of a given state defined by its x-axis and y-axis state variables *x* and *y*, the following formula is used:

$$f(x,y) = \frac{1}{2 \cdot \pi \cdot \sigma_x \cdot \sigma_y \cdot \sqrt{1-\rho^2}} \cdot e^{-\frac{1}{2 \cdot (1-\rho^2)} \cdot \left(\frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu_y)^2}{\sigma_y^2} - \frac{2 \cdot \rho \cdot (x-\mu_x) \cdot (y-\mu_y)}{\sigma_x \cdot \sigma_y} \right)}$$

In the previous equation, μ_x and μ_y are the mean values of x-axis and y-axis state variables respectively; σ_x and σ_y are the standard deviations of x-axis and y-axis state variables respectively; ρ is the correlation between x-axis and y-axis state variables. Taking into account the aforementioned condition of uncorrelated state variables, i.e. $\sigma_{xy} = 0$, the equation is reduced to:

$$f(x,y) = \frac{1}{2 \cdot \pi \cdot \sigma_x \cdot \sigma_y} \cdot e^{-\frac{1}{2} \cdot \left(\frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu_y)^2}{\sigma_y^2} \right)}$$

Implementing a hardware module in order to perform this operation is almost impossible, since it is very complex. Therefore, another approach has been used: this module has been implemented as a Look-Up Table (LUT), i.e. an array that replaces complex computations with array indexing operations.

Before explaining how the LUT is generated, the hardware module architecture will be presented. The VHDL module symbol is shown in Fig. III.11, and it has some parameters that are configurable. Note that it is a combinational logic block, and therefore it has no clock input. Moreover, it has no reset port either.

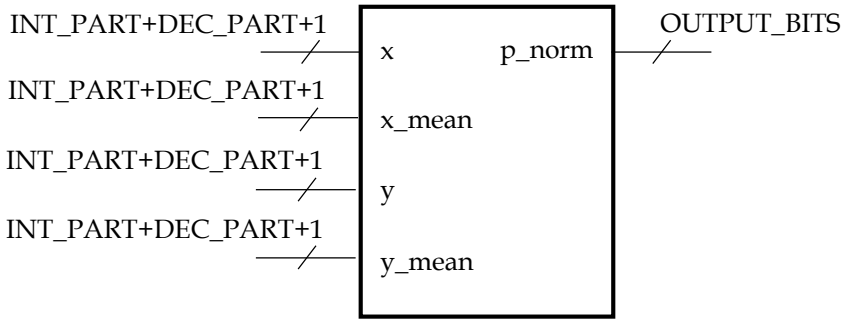


Fig. III.11. Fitness calculation symbol

- INT_PART: input integer part size in bits.
- DEC_PART: input decimal part size in bits.
- OUTPUT_BITS: output size in bits, i.e. fitness/weight resolution.
- MEAN_X: x-axis state variable mean value.
- STD_DEV_X: x-axis state variable standard deviation.
- MEAN_Y: y-axis state variable mean value.
- STD_DEV_Y: y-axis state variable standard deviation.
- VALUES: number of divisions in a single axis. This determines the resolution of the LUT, whose total size can be computed as $VALUES^2$.
- EXT_MEAN: selects whether the mean values are defined by the generic parameters or by the external signals.

The look-up table is generated during the synthesis process, using the computational resources available in the computer. The LUT values are calculated as real numbers and then are truncated in order to represent then in integer precision. When the system is running, particle fitness values are obtained using the LUT and zero-order interpolation. The VHDL coding of the look-up table generation is shown in both Fig. III.12 and Fig. III.13.

```
(...)

architecture rtl of normal_eval is

  -- Type definitions
  type pdf is array (VALUES**2-1 downto 0) of
std_logic_vector(OUTPUT_BITS-1 downto 0);

  -- Function definitions
  impure function generate_pdf return pdf is

  (...)

  end generate_pdf;

  -- Constant definitions
  constant pdf_values : pdf := generate_pdf;

begin

(...)
```

Fig. III.12. LUT generation in VHDL coding

```
  -- Function definitions
  impure function generate_pdf return pdf is
    -- Constants
    constant min     : real := 0.0;
    constant max     : real := real(2**INT_PART
    constant inc     : real := (max-min)/real(VALUES);
    constant mean_x  : real := 0.0;
    constant sigma_x : real := real(STD_DEV_X);
    constant mean_y  : real := 0.0;
    constant sigma_y : real := real(STD_DEV_Y);
    -- Variables
    variable pdf_aux : pdf;
    variable calc    : real := 0.0;
    variable norm    : real := 0.0;
    variable point_x : real := min;
    variable point_y : real := min;
  begin

    -- Calculate normalizing constant
    norm := 1.0/(2.0*MATH_PI*sigma_x*sigma_y)*exp(-1.0/2.0*(((mean_x-
mean_x)**2)/(sigma_x**2) + ((mean_y-mean_y)**2)/(sigma_y**2)));

    -- Generate LUT with PDF
    x_loop:
    for i in 0 to VALUES-1 loop

      y_loop:
      for j in 0 to VALUES-1 loop

        -- Compute Probability Density Function
        calc := 1.0/(2.0*MATH_PI*sigma_x*sigma_y)*exp(-
1.0/2.0*(((point_x-mean_x)**2)/(sigma_x**2) + ((point_y-
mean_y)**2)/(sigma_y**2)));

        -- Store value
        pdf_aux(i*VALUES+j) :=
std_logic_vector(to_unsigned(integer(calc*(2.0**(OUTPUT_BITS)-
1.0)/norm),OUTPUT_BITS));

        -- Increment point y
        point_y := point_y + inc;

      end loop;

      -- Increment point_x
      point_x := point_x + inc;
      -- Reset point y
      point_y := min;

    end loop;

    -- Return solution
    return pdf_aux;

  end generate_pdf;
```

Fig. III.13. Normal distribution probability density function generation in VHDL

In order to reduce resource utilization rates and maximize the LUT resolution, some enhancements have been implemented in this unit:

- Since 99.7% of the normal distributed values can be found within three standard deviations of the mean, it is possible to define the effective size as the minimum bit size that allows the representation of the number $3 \cdot \sigma_{max}$, where $\sigma_{max} = max(STD\_DEV\_X, STD\_DEV\_Y)$. The LUT is built for that particular bit size instead of the whole dynamic range defined by INT_PART. Any value out of the effective size range has a zero value at the output port.
- Considering that, the resulting multivariate normal distribution is symmetric. Therefore, it is only necessary to store the values in one of the four quadrants (rectangular coordinate system).

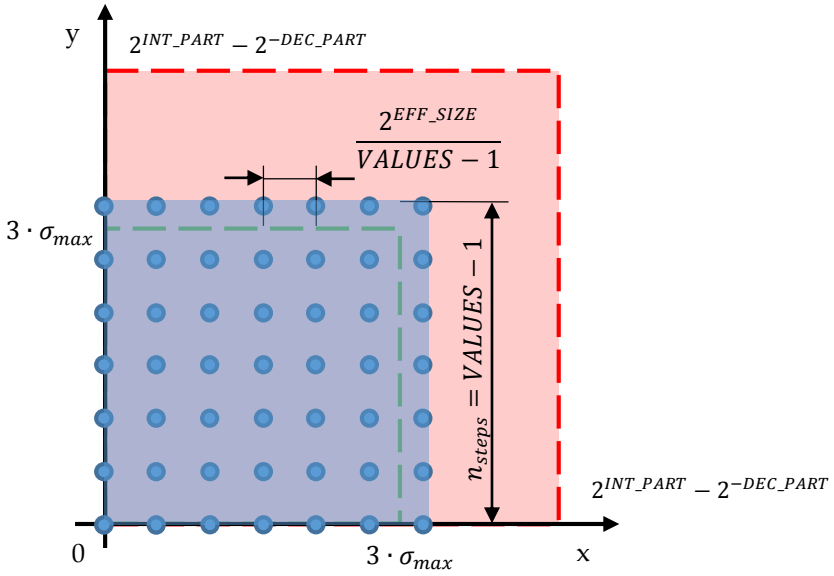


Fig. III.14. LUT enhancements

These modifications are shown in Fig. III.14. The green dotted line represents the limit $3 \cdot \sigma_{max}$. The blue dots represent the calculated values in the LUT. Any indexed point within the blue area would directly access the LUT, whereas if the value is within the red zone, the output will always be zero.

LUT access (blue area) is carried out following these equations:

$$\begin{cases} id_x = \left\lfloor \frac{VALUES-1}{2^{EFF\_SIZE}} \cdot (x - x_{mean}) \right\rfloor \\ id_y = \left\lfloor \frac{VALUES-1}{2^{EFF\_SIZE}} \cdot (y - y_{mean}) \right\rfloor \end{cases} \qquad id = VALUES \cdot id_x + id_y$$

The effective numeric values of the signals in this module, as well as their characteristics (i.e. attributes) are shown in Table III.2.

Name	Mode	Attributes	Minimum	Maximum	Increment
x	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
x_mean	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_mean	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
p_norm	out	unsigned	0	2^{OUTPUT_BITS}	1

Table III.2. Fitness calculation signal characteristics

III.3 Particle Registers

Particle states have to be stored during the filtering operation. Therefore, it is necessary to introduce memory elements in the design. How can these memory elements be implemented?

- Registers: flip-flops store the state variables. One of the most important advantages this approach has is that the memory access can be done in parallel (i.e. more than one particle state can be reached in one clock cycle), even though it can also be sequential (i.e. one particle state per clock cycle). However, resource utilization rates are high, thus having large area overheads.
- RAM memory: random access memories can be used to store particle states. Data access can only be performed in a sequential manner (if the RAM memory has only one port), but the area overhead is smaller compared with the previous alternative.

In the end, a RAM implementation was chosen, mainly due to the fact that RAM memories are easy to implement as BRAM (Block RAM) in Xilinx FPGAs. Therefore, this is the only module in the whole system that is technology dependent. BRAM inference in Xilinx tools requires specific syntactic constructions, especially when trying to describe dual port RAMs with two read/write ports in VHDL.

VHDL coding of the state register module has been provided in both Fig. III.15 (entity definitions) and Fig. III.16 (architecture description). Note that in the entity, an attribute is defined in order to tell the synthesis tool that the module has to be inferred as a block RAM (in case it is not done automatically). Also, note that in the architecture description a shared variable is used. This is, generally speaking, a type of variable that cannot be synthesized. However, the VHDL code has to be written exactly like that so that the synthesis tool detects that a block RAM is being defined. This is the reason why this module is technology dependent (it depends on Xilinx FPGAs and synthesis tool, XST).

```
-- Particle state memory
-- Read-first Dual Port RAM
-- Xilinx FPGA implementation target

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.all;

entity state_register is

  generic
  (
    -- Number of elements
    ELEMENTS    : natural := 1024;
    -- Address port width in bits
    ADDR_WIDTH  : natural := 32;
    -- Data resolution in bits
    DATA_WIDTH  : natural := 32
  );

  port
  (
    -- CLK signals
    clk_a  : in std_logic;
    clk_b  : in std_logic;
    -- Control signals
    en_a   : in std_logic;
    en_b   : in std_logic;
    we_a   : in std_logic;
    we_b   : in std_logic;
    -- Address ports
    addr_a : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    addr_b : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    -- Input data ports
    din_a  : in std_logic_vector(DATA_WIDTH-1 downto 0);
    din_b  : in std_logic_vector(DATA_WIDTH-1 downto 0);
    -- Output data ports
    dout_a : out std_logic_vector(DATA_WIDTH-1 downto 0);
    dout_b : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );

  -- BRAM definitions
  attribute ram_style : string;
  attribute ram_style of state_register : entity is "block";

end state_register;
```

Fig. III.15. State register VHDL entity

```
architecture rtl of state_register is
  -- Type definitions
  type RAM is array (ELEMENTS-1 downto 0) of
std_logic_vector(DATA_WIDTH-1 downto 0);
  -- Shared variable definitions (TECHNOLOGY DEPENDENT)
  shared variable state : RAM := (others => (others => '0'));
begin

  -- Particle state storage (Dual port RAM): PORTA
  port_a: process(clk_a)
    -- Variable definitions
    variable index_a : integer range 0 to ELEMENTS-1 := 0;
  begin
    if clk_a'event and clk_a = '1' then
      -- Enable memory
      if en_a = '1' then
        -- Index calculation
        index_a := to_integer(unsigned(addr_a));
        -- Read operation PORTA
        dout_a <= state(index_a);
        -- Write operation PORTA
        if we_a = '1' then
          state(index_a) := din_a;
        end if;
      end if;
    end if;
  end process;

  -- Particle state storage (Dual port RAM): PORTB
  port_b: process(clk_b)
    -- Variable definitions
    variable index_b : integer range 0 to ELEMENTS-1 := 0;
  begin
    if clk_b'event and clk_b = '1' then
      -- Enable memory
      if en_b = '1' then
        -- Index calculation
        index_b := to_integer(unsigned(addr_b));
        -- Read operation PORTA
        dout_b <= state(index_b);
        -- Write operation PORTA
        if we_b = '1' then
          state(index_b) := din_b
        end if;
      end if;
    end if;
  end process;

end rtl;
```

Fig. III.16. State register VHDL architecture

The VHDL module symbol is shown in Fig. III.17, and it has some parameters that are configurable.

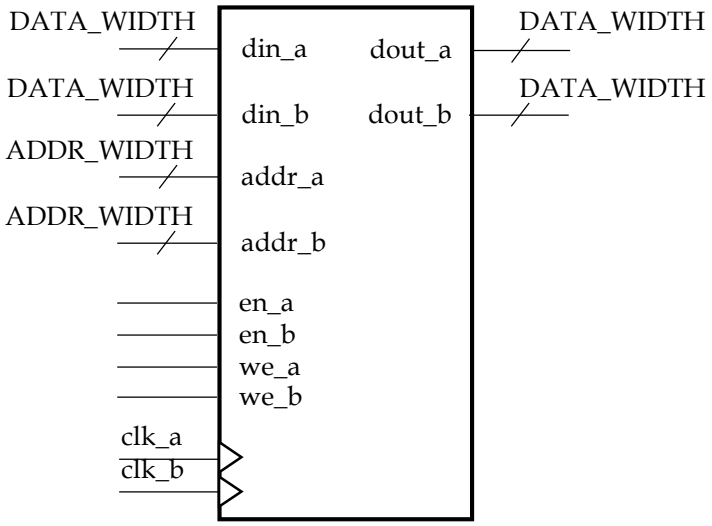


Fig. III.17. State register symbol

- ELEMENTS: number of available memory elements.
- ADDR_WIDTH: address size in bits.
- DATA_WIDTH: data size in bits.

Since each particle has four different state variables plus its weight/fitness value, a higher-level module has been created using five instances of the state register. Thus, all particle information is stored within a single VHDL module.

There are two particle registers in the Evolutionary Particle Filter, because at the end of each generation, during the survivor selection process, it is necessary to store a copy of each particle so as not to overwrite existing information. This will be explained with further details in forthcoming sections of this chapter.

III.4 Process Model

Importance sampling is carried out within this unit. As stated in previous sections, the importance sampling function is the process model. Particle state is updated from a time step to the following one using the dynamic model presented in the introductory section of this chapter. This update process generates a new set of predicted states (as in the prediction stage of the Kalman filter).

$$\boldsymbol{x_k} = A \cdot \boldsymbol{x_{k-1}} + \boldsymbol{w_{k-1}}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad \boldsymbol{x_k} = \begin{bmatrix} x_k \\ y_k \\ vx_k \\ vy_k \end{bmatrix} \qquad \boldsymbol{w_k} = \begin{bmatrix} w_k^x \\ w_k^y \\ w_k^{vx} \\ w_k^{vy} \end{bmatrix} \sim \begin{bmatrix} N(\mu_x, \sigma_x) \\ N(\mu_y, \sigma_y) \\ N(\mu_{vx}, \sigma_{vx}) \\ N(\mu_{vy}, \sigma_{vy}) \end{bmatrix}$$

In order to implement this matrix-multiplication dynamic model in hardware, there were two main different alternatives:

- Parallel implementation: all four state variables are updated in the same clock cycle, using 16 multipliers.
- Resource sharing implementation: using multiplexers and only four multipliers, but consuming more clock cycles.

The target platform is a XUPV5 board, which features a Xilinx Virtex-5 XC5VLX110T FPGA. This FPGA has a limited number of hardware multiplier resources, namely, only 64 DSP slices. Hence, every module with multiplications in this thesis has been implemented using the second approach, i.e. sharing hardware resources.

The process model module architecture has been optimized in order to reduce the maximum delay, i.e. to increase the maximum allowable frequency, and has been presented in Fig. III.18. In addition, the finite state machine that controls data transitions and generates control signals has been included in Fig. III.19.

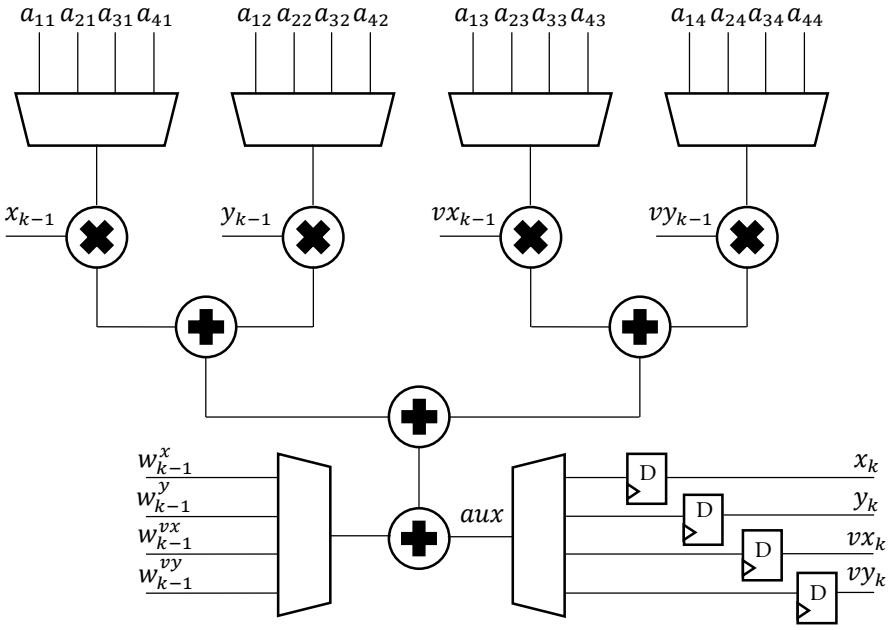


Fig. III.18. Process model hardware architecture

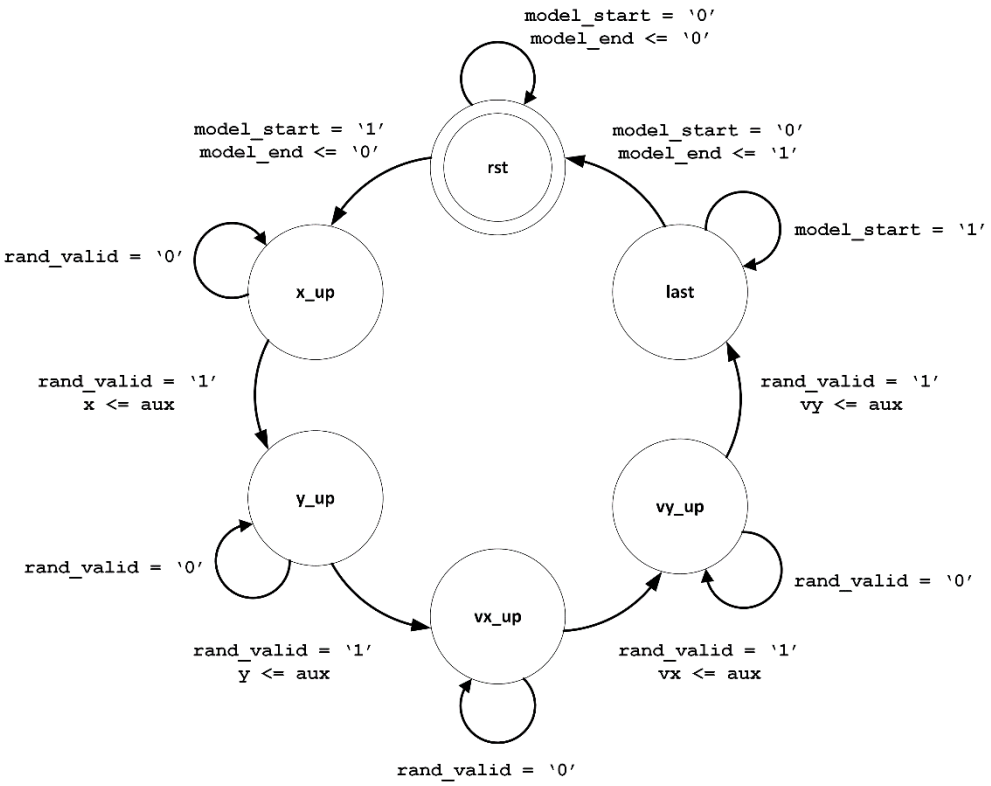


Fig. III.19. Process model control FSM

The FSM works as follows: first, it remains waiting at *rst* state. When a rising edge appears in the control signal *model_start*, the FSM goes through all updating states, i.e. *x_up*, *y_up*, *vx_up*, and *vy_up*. To change from one state variable to the next one that has to be updated, it is absolutely necessary that a valid normal-distributed random number is available. Therefore, transitions between states depend on the signal *randn_valid* being high. If this condition is not satisfied, the FSM stops and remains in the same state. There is another extra state, called *last*, in which the FSM stops unless the input control signal *model_start* is low. This prevents the system from updating the same particle state twice instead of only once. In the final state transition, the output control signal *model_end* is set to high only one clock cycle.

One of the most important disadvantages of this module is the latency. Since the state variable update process depends on valid random samples, this latency can have a huge impact in overall performance if it is not taken into account. In this thesis it has been, since the random number generator has no latency between normal samples.

The VHDL module symbol is shown in Fig. III.20, and it has some parameters that are configurable.

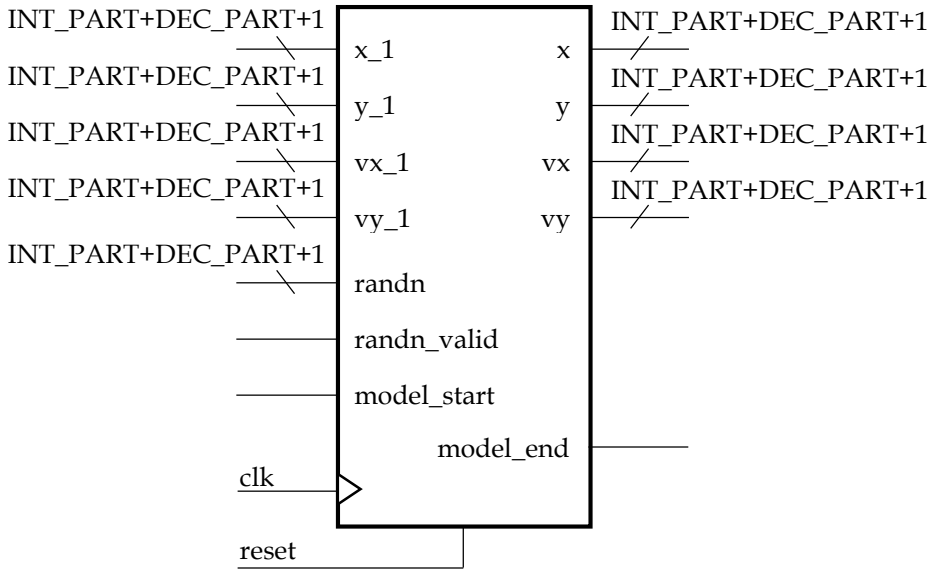


Fig. III.20. Process model symbol

- INT_PART: integer part size in bits.
- DEC_PART: decimal part size in bits.
- SIGMA_RATIO: ratio between the standard deviation in position and in velocity.
- T: sampling time in the dynamic model.

The dynamic matrix coefficients and the number of bits for the right shift operation are computed during the synthesis process (following the same procedure used in the fitness calculation unit). VHDL coding is shown in Fig. III.21 and in Fig. III.22.

```
-- Get shift value from sigma ratio
function get_ratio(ratio : real) return natural is
  -- Variable definitions
  variable ratio_log : natural;
begin
  -- Compute value
  ratio_log := integer(log2(ratio) + 1.0);
  -- Return value
  return ratio_log;
end function;
```

Fig. III.21. Bit shift computation in VHDL

```
  -- Type definitions
  type dyn_matrix is array(3 downto 0, 3 downto 0) of
std_logic_vector(INT_PART+DEC_PART+1-1 downto 0);

  -- Function definitions
  -- Obtain std_logic_vector matrix from real matrix
  function generate_matrix (T : real) return dyn_matrix is
    -- Constant definitions
    type dyn_matrix_real is array (3 downto 0, 3 downto 0) of real;
    constant a : dyn_matrix_real :=
      (
        (1.0, 0.0,   T, 0.0),
        (0.0, 1.0, 0.0,   T),
        (0.0, 0.0, 1.0, 0.0),
        (0.0, 0.0, 0.0, 1.0)
      );
    variable a_int : dyn_matrix;
  begin
    for i in 3 downto 0 loop
      for j in 3 downto 0 loop
        a_int(i,j) :=
std_logic_vector(to_signed(integer(real(2**DEC_PART)*a(i,j)),a_int(i,j)'
length));
      end loop;
    end loop;
    -- Return value
    return a_int;
  end function;
```

Fig. III.22. Dynamic matrix generation in VHDL

The effective numeric values of the signals in this module, as well as their characteristics (i.e. attributes) are shown in Table III.3.

Name	Mode	Attributes	Minimum	Maximum	Increment
clk	in	rising	0	1	-
reset	in	high	0	1	-
x_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
randn	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
randn_valid	in	-	0	1	-
model_start	in	-	0	1	-
x	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
model_end	out	-	0	1	-

Table III.3. Process model signal characteristics

III.5 Crossover Unit

This hardware module performs the recombination operation in the evolutionary resampling stage. Crossover equations have been provided as a reminder.

$$\begin{cases} x_k^a = \alpha \cdot x_k^i + (1-\alpha) \cdot x_k^j \\ x_k^b = \alpha \cdot x_k^j + (1-\alpha) \cdot x_k^i \end{cases}$$

The implementation guidelines are exactly the same that were discussed in the previous section. In order to avoid excessive resource consumption rates, the module has been described as a resource-sharing architecture. The hardware architecture can be seen in Fig. III.23. Note that the same multiplexed inputs (op_1 and op_2) are used to generate two children (aux_a and aux_b). The control FSM for this particular module appears in Fig. III.24.

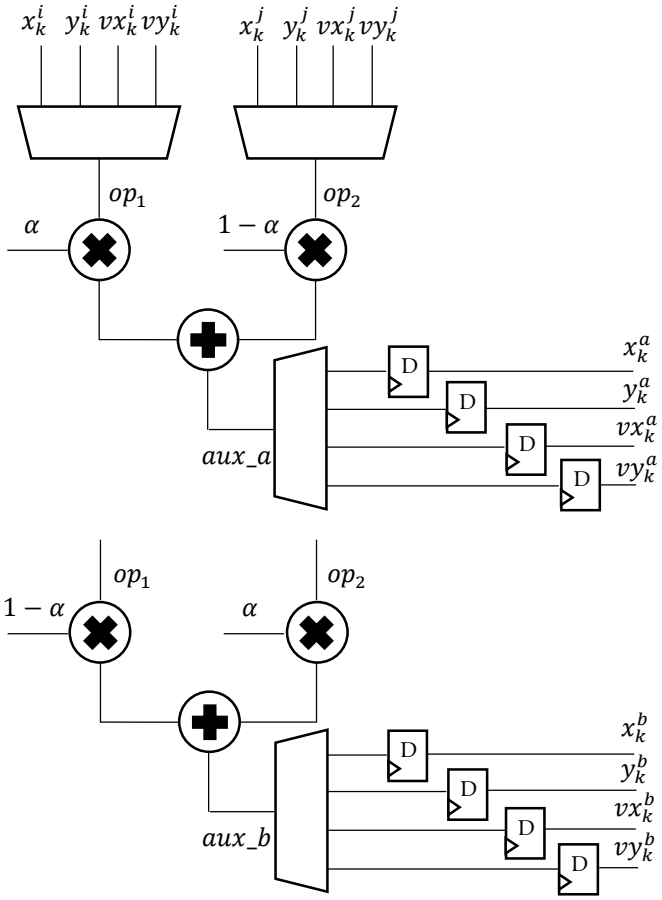


Fig. III.23. Crossover unit hardware architecture

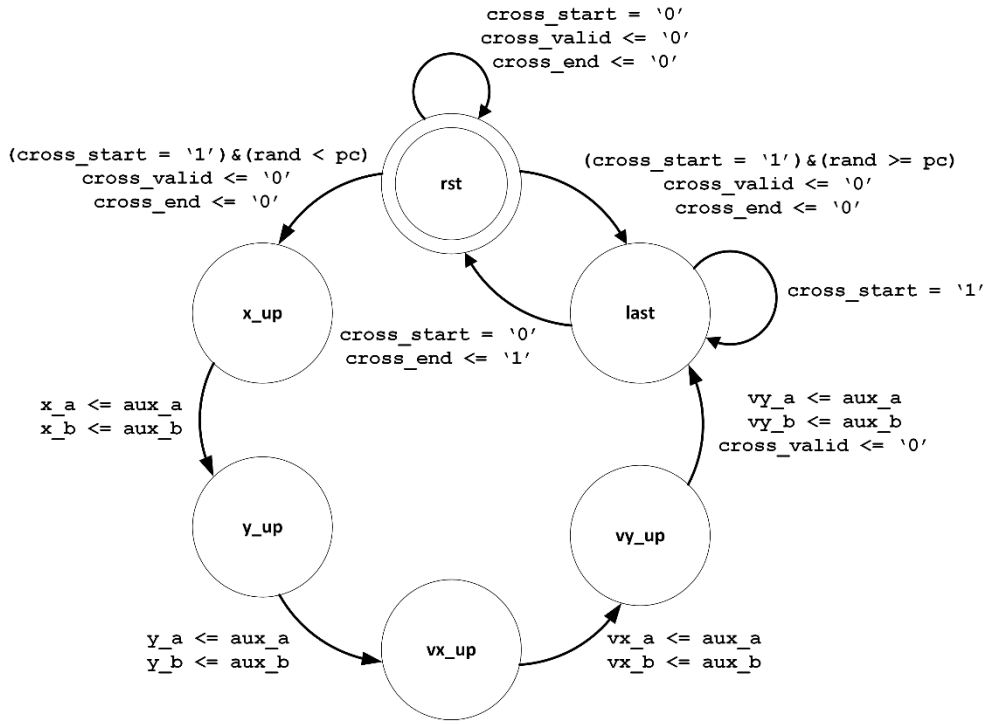


Fig. III.24. Crossover unit control FSM

This module operates similarly to the process model unit. Unless the input control signal, in this case *cross_start* goes high, the module does nothing. Once a rising edge has been detected in that control signal, there are two possible state transitions. If the uniform-distributed random number which is drawn is greater than the fixed crossover probability threshold, i.e. p_c, the system goes to the state *last*. Otherwise, the next state is *x_up*, and clock cycle after clock cycle, the system performs the recombination of the four state variables. Once this has been finished, the system will step to the state *last*, setting the control signal *cross_valid* to high. Independently on how the *last* state is reached, the system will stop until the input control signal goes low. Only then will the system go to the initial state, setting the other output control signal, *cross_end*, to high for only one clock cycle. Once the system is in *rst* state, both output control signals are set to low. These two outgoing control signals let the rest of the system know when the recombination has finished and whether there has been offspring generation or not. Refer to the following sections for further information.

Although the system takes more than one clock cycle to finish the recombination algorithm, the latency is fixed, since it does not depend on the random number generation process (it is assumed that uniform random numbers are generated with a LFSR, thus having one sample per clock cycle).

The VHDL module symbol is shown in Fig. III.25, and it has some parameters that are configurable.

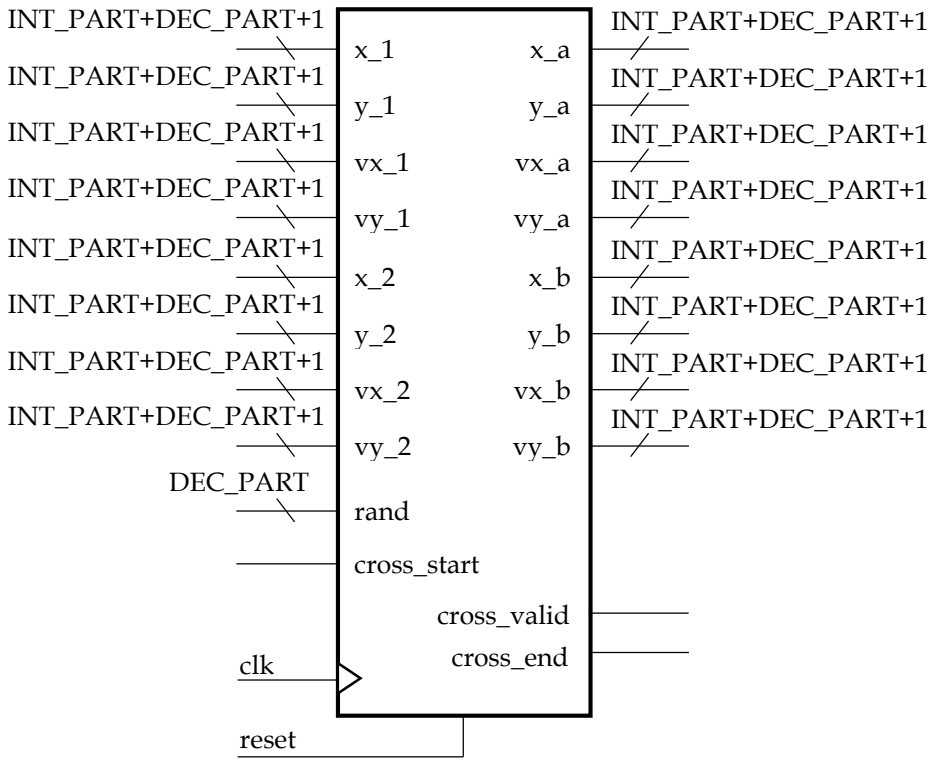


Fig. III.25. Crossover unit symbol

- INT_PART: integer part size in bits.
- DEC_PART: decimal part size in bits.
- P_CROSS: crossover probability p_c.

The generic parameter P_CROSS is defined as a real number to represent a real value ranging from 0.0 to 1.0. However, real numbers cannot be synthesized. In order to overcome this problem, the crossover probability is computed (again) during the synthesis process, expressing it with the resolution given by the other generic parameters, specifically DEC_PART. This conversion can be seen in Fig. III.26.

```
-- Constant definitions
constant pc : integer := integer(real(2**DEC_PART)*P_CROSS);
```

Fig. III.26. Real to integer conversion in crossover probability threshold

The effective numeric values of the signals in this module, as well as their characteristics (i.e. attributes) are shown in Table III.4.

Name	Mode	Attributes	Minimum	Maximum	Increment
clk	in	rising	0	1	-
reset	in	high	0	1	-
x_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_1	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
x_2	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_2	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_2	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_2	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
rand	in	unsigned	0	$1 - 2^{-DEC\_PART}$	2^{-DEC_PART}
cross_start	in	-	0	1	-
x_a	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_a	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_a	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_a	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
x_b	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_b	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_b	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_b	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
cross_valid	out	-	0	1	-
cross_end	out	-	0	1	-

Table III.4. Crossover unit signal characteristics

III.6 Mutation Unit

The implementation of the second genetic operator in the evolutionary resampling stage will be explained in this section. Since there was not only one but two different operations (namely, random placement and local search), in this module there are two different data paths.

The first data path generates a child using random placement; therefore, the following equation has been implemented in hardware:

$$x_k^c = x_{min} + \beta \cdot (x_{max} - x_{min})$$

The second data path creates a child using local placement, i.e. placing this child in a close environment (in the state space) of the parent. This operation is represented as the following equation:

$$x_k^c = x_k^i + \delta_k$$

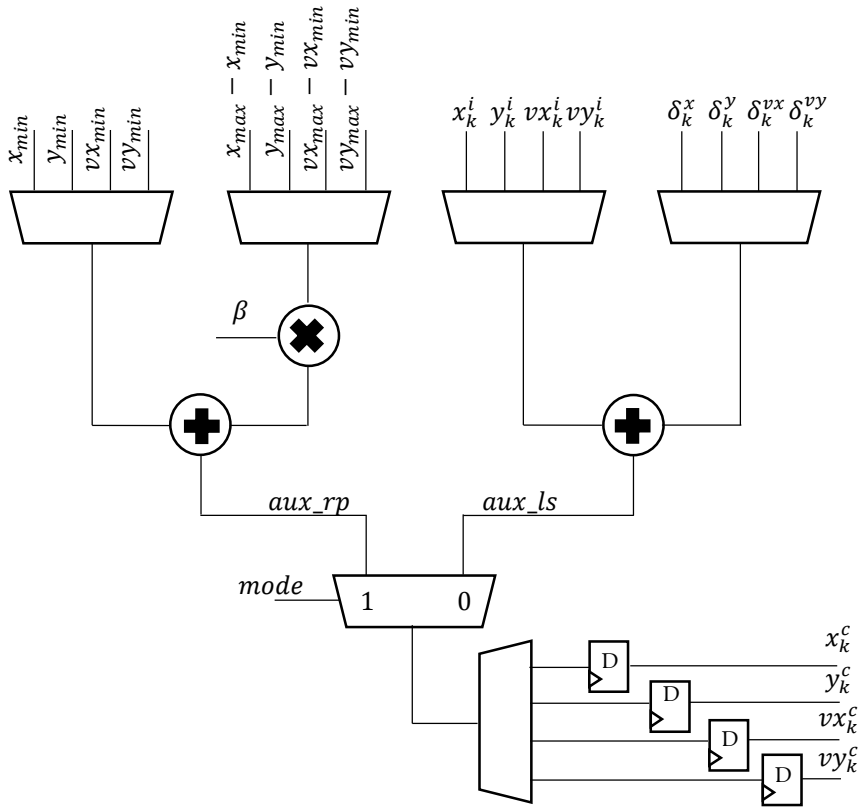


Fig. III.27. Mutation unit hardware architecture

In order to perform both operations, the architecture that is shown in Fig. III.27 has been proposed. By taking a look at this figure, it is possible to see the two different data paths that have been explained in the previous paragraphs. The random placement data path is the one on the left side, whereas the local search data path is located on the right side. The control logic decides whether to use the random placement child (aux_rp) or the local search one (aux_ls) with a control signal ($mode$).

The control FSM can be seen in Fig. III.28. It is more complex than the control FSMs from both process model and crossover units. However, it shares common features with them. For instance, the capability of deciding whether the child is valid or not, based upon a uniform random number comparison with a fixed threshold (exactly the same that happened in the crossover module), or the need for valid normal random numbers (as in the process model module).

Note that, although the control logic is by far more complex than in the aforementioned modules, the data logic is simpler, since only one multiplier is needed, and only in one of the paths (random placement of the particle in the state space).

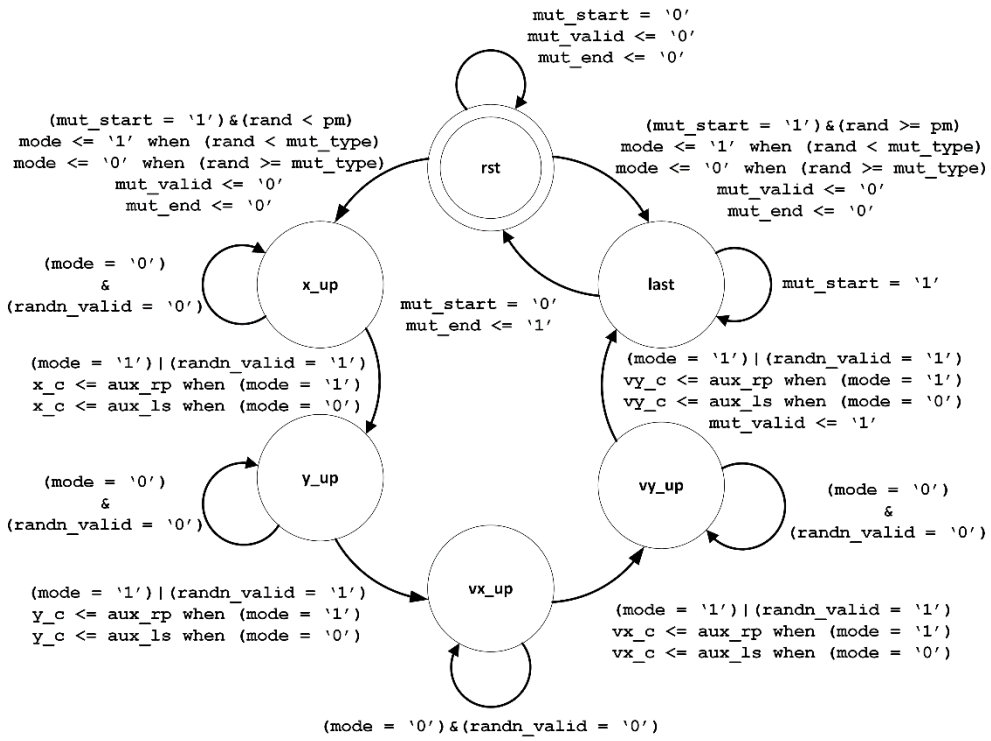


Fig. III.28. Mutation unit control FSM

The control strategy in this module can be analyzed as follows. As in the two previous modules, the mutation unit waits for a rising edge in the input control signal (in this case, it is the signal *mut_start*). If this condition is met, a uniform random number is drawn. Two possibilities can arise: on the one hand, the random number is less than the threshold; on the other hand, it is greater than the threshold. If it is greater than the threshold, the procedure is the same as in the crossover unit: the system goes to the state *last*. However, if the random number is less than the threshold, the internal control signal *mode* is set to high or low depending on the comparison of that random number and a second threshold. This comparison decides the type of mutation that will be performed on the parent. Then, the state machine starts generating each state variable offspring. Note that if the current mode is local search, i.e. $mode = '0'$, the system will have to wait for a valid normal-distributed random number to be able to get to the next state. On the contrary, if the mode is random placement, i.e. $mode = '1'$, the system does not have to wait for anything, and therefore the transition takes only one clock cycle to happen. Independently of the current mode, once the system has generated valid offspring for each state variable, the control signal *mut_valid* is set to high and the state *last* is reached. When the input control signal goes low, the system returns to the idle state, *rst*, setting the other control signal to high (one clock cycle).

The VHDL module symbol is shown in Fig. III.29, and it has some parameters that are configurable.

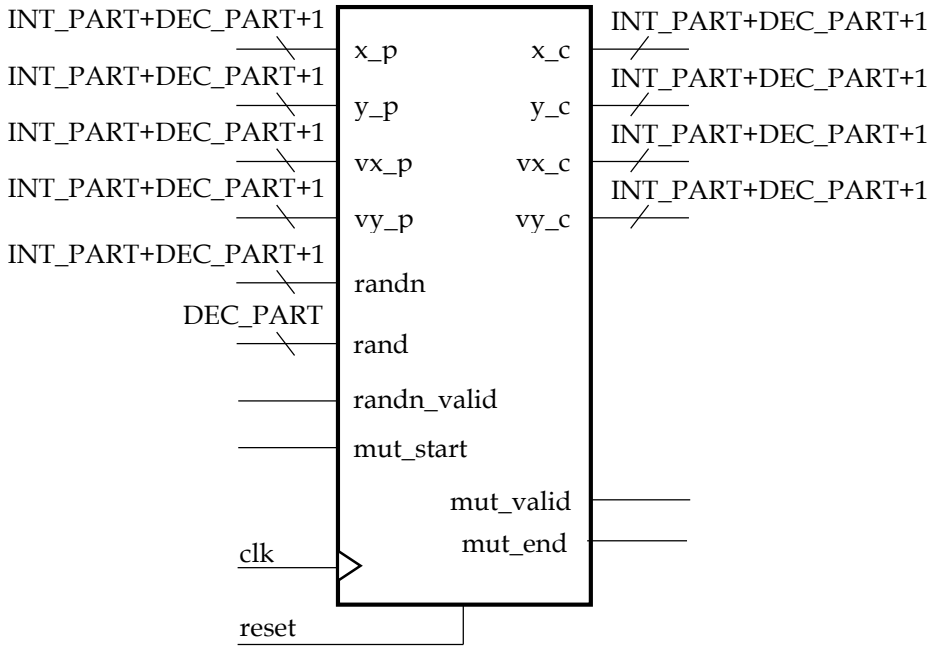


Fig. III.29. Mutation unit symbol

- INT_PART: integer part size in bits.
- DEC_PART: decimal part size in bits.
- SIGMA_RATIO: standard deviation ratio $r_\sigma = \frac{\sigma_{position}}{\sigma_{local\ search}}$.
- P_MUT: mutation probability p_m.
- MUT_RATIO: mutation ratio r_m.

Following the same reasoning process that in the previous section, the probability thresholds are computed during the synthesis process, using the expressions provided in Fig. III.30. In addition to that, since there are normal random numbers involved, the number of bits that suffer the right shift operation is computed the same way it was calculated in the process model unit (see Fig. III.21).

```
-- Constant definitions
constant pm       : integer := integer(real(2**DEC_PART)*P_MUT);
constant mut_type : integer := integer(real(2**DEC_PART)*P_MUT*MUT_RATIO);
```

Fig. III.30. Real to integer conversion in mutation unit thresholds

The effective numeric values of the signals in this module, as well as their characteristics (i.e. attributes) are shown in Table III.5.

Name	Mode	Attributes	Minimum	Maximum	Increment
clk	in	rising	0	1	-
reset	in	high	0	1	-
x_p	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_p	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_p	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_p	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
randn	in	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
randn_valid	in	-	0	1	-
rand	in	unsigned	0	$1 - 2^{-DEC\_PART}$	2^{-DEC_PART}
mut_start	in	-	0	1	-
x_c	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
y_c	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vx_c	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
vy_c	out	signed	-2^{INT_PART}	$2^{INT\_PART} - 2^{-DEC\_PART}$	2^{-DEC_PART}
mut_valid	out	-	0	1	-
mut_end	out	-	0	1	-

Table III.5. Mutation unit signal characteristics

III.7 Dividers

At the very end of the particle filtering process, an estimation of the system state is generated. In order to compute each state variable estimated value, the following equation is used:

$$\widehat{x_k} = \sum_i w_i \cdot x_k^i$$

In this equation, x_k^i represents the state variables of the particle *i* at time *k*, whereas w_i are the normalized weights of those particles. In order to avoid unnecessary computations of normalized values, this process is carried out at the estimation stage of the Evolutionary Particle Filter.

$$\widehat{x_k} = \sum_i w_i \cdot x_k^i = \frac{\sum_i f_i \cdot x_k^i}{\sum_i f_i}$$

This last equation represents the new problem that needs to be solved: a division. Note that each normalized weight has been expressed as $w_i = \frac{f_i}{\sum_i f_i}$, where f_i represents the output values of the fitness calculation module. After searching through the literature, the division algorithm that has been selected and implemented is the radix-2 non-restoring division algorithm that was found in [5].

Radix-2 non restoring division algorithm can be explained as follows. First, consider the operation we are trying to solve: the division.

$$num = q \cdot den + r$$

The last formula represents a common division, where *num* represents the numerator, *den* is the denominator, q is the quotient and r is the remainder.

The hardware architecture that has been used can be seen in Fig. III.31. It requires one register to store the denominator, which is called *inc*, a shift register, which is named *op_register*, and an adder/substractor.

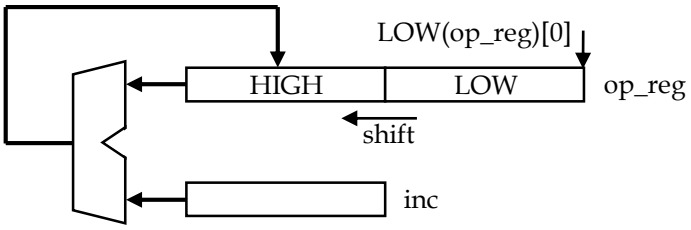


Fig. III.31. Divider hardware architecture

Having said that, let us focus on the algorithm itself, which has been illustrated in the following figure (Fig. III.32).

```
LOW(op_reg) = num;
HIGH(op_reg) = 0;
inc = den;

for(int i = 0; i < N_BITS; i++)
{
  shift_left(op_reg,1);
  (HIGH(op_reg) < 0) ? (HIGH(op_reg) += inc) : (HIGH(op_reg) -= inc);
  (HIGH(op_reg) < 0) ? (LOW(op_reg)[0] = 0) : (LOW(op_reg)[0] = 1);
}

q = LOW(op_reg);
(HIGH(op_reg) < 0) ? (r = HIGH(op_reg) + inc) : (r = HIGH(op_reg));
```

Fig. III.32. Radix-2 non-restoring division algorithm

This algorithm works well if both *num* and *den* are unsigned integers, but some modifications have to be made in order to adapt the algorithm to signed integers (the Evolutionary Particle Filter state variables are signed integer, therefore this is mandatory). These changes have been included within the system control FSM, which can be seen in Fig. III.33. The signs are computed before the aforementioned algorithm starts working with the absolute values of the operands. Once the algorithm has finished its execution, the quotient and remainder signs are modified with the previous knowledge of the operand signs.

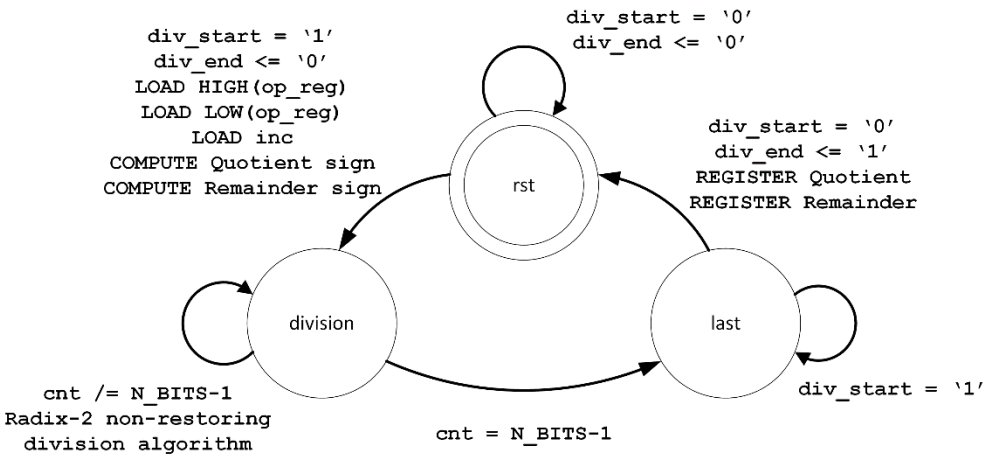


Fig. III.33. Divider control FSM

The finite state machine used to control the system has only three states. The first one represents the idle mode of the divider unit. When there is a rising edge in the control signal *div_start*, the signs of the quotient and the remainder are calculated, and the input numbers are stored in its respective registers (*inc, op_reg*). Then, the algorithm computes the division while the FSM remains in the state *division*. This process takes N_BITS cycles, being N_BITS the number of bits each operand has. Then, the system goes to a waiting state until the input control signal goes low. At this moment, the output control signal is set to high for one clock cycle, and both the quotient and the remainder are registered as output values.

The VHDL module symbol is shown in Fig. III.34, and it has some parameters that are configurable.

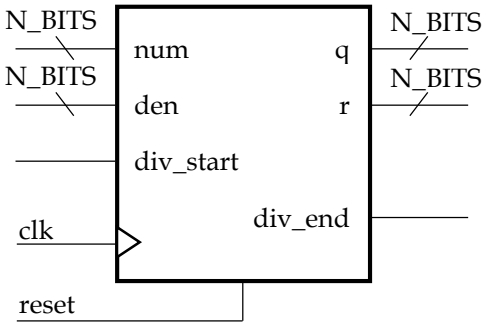


Fig. III.34. Divider symbol

- N_BITS: data signal resolution in bits.

The effective numeric values of the signals in this module, as well as their characteristics (i.e. attributes) are shown in Table III.6.

Name	Mode	Attributes	Minimum	Maximum	Increment
clk	in	rising	0	1	-
reset	in	high	0	1	-
num	in	signed	-2^{N_BITS-1}	$2^{N\_BITS-1}-1$	1
den	in	signed	-2^{N_BITS-1}	$2^{N\_BITS-1}-1$	1
div_start	in	-	0	1	-
q	out	signed	-2^{N_BITS-1}	$2^{N\_BITS-1}-1$	1
r	out	signed	-2^{N_BITS-1}	$2^{N\_BITS-1}-1$	1
div_end	out	-	0	1	-

Table III.6. Divider signal characteristics

III.8 Additional Logic

In the previous section, the divider unit was explained. Nevertheless, how are the inputs of that unit generated? Additional hardware resources used in this thesis will be presented in this section. In particular, multiply and accumulate (MAC) units and accumulators will be reviewed.

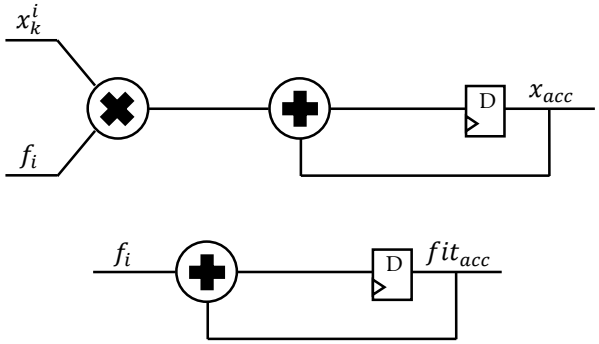


Fig. III.35. State variable MAC unit and fitness accumulator

In Fig. III.35, the hardware that has been implemented to perform the operations of multiply and accumulate on the state variables and accumulate on the fitness is shown.

$$\widehat{x_k} = \sum_i w_i \cdot x_k^i = \frac{\sum_i f_i \cdot x_k^i}{\sum_i f_i}$$

Therefore, the upper component performs the operation $\sum_i f_i \cdot x_k^i$, whereas the lower component is in charge of computing $\sum_i f_i$. Since these operations use registers, it is important to select a particle number that makes the module feasible. Excessive area overhead could be generated if this is not taken into account.

III.9 Process Scheduling and System Control

Complex systems such as the Evolutionary Particle Filtering require large control FSMs in order to work properly. In this last section, the control logic that connects and rules the operating conditions is presented.

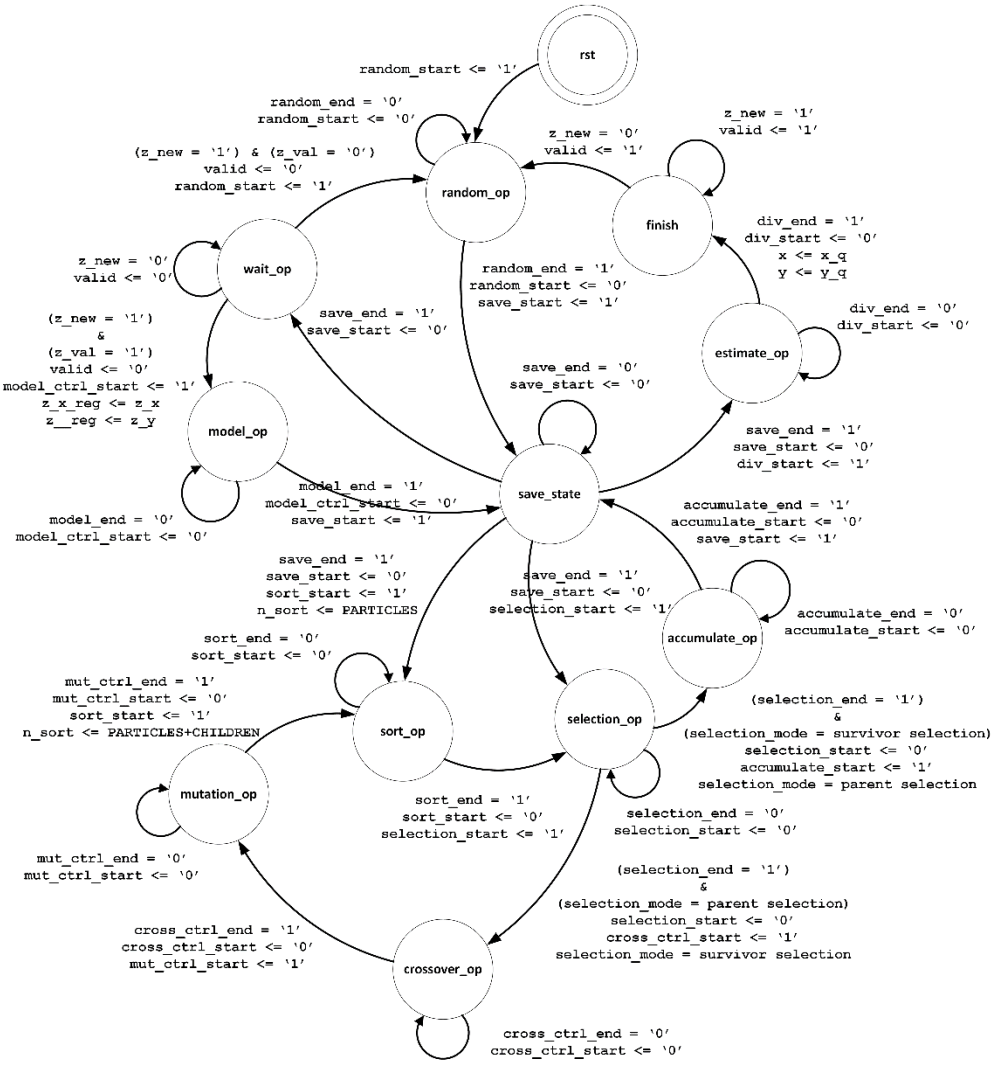


Fig. III.36. Process Scheduler

The main control is represented in Fig. III.36. In this figure, some control signals have been omitted for the sake of simplicity. However, all possible state transitions, as well as their conditions have been included.

Since the system is very complex, local control strategies have also been implemented. There are eight different control submodules, each one with a specific functionality:

- Initial particle random placement.
- Process model control: interface with the process model unit.
- Particle sort algorithm: bubble sort algorithm to sort particles according to their fitness values. This algorithm is very slow with large population sizes, and in future works might be revised in order to increase overall system performance.
- Selection algorithm: performs Stochastic Universal Sampling.
- Crossover control: interface with the crossover unit.
- Mutation control: interface with the mutation unit.
- Accumulation: controls the MAC an accumulators, presented in the previous section.
- Save particle state: stores intermediate particle states (after importance sampling, between generations, etc.) for representation purposes.

The FSM in Fig. III.36 generates the necessary control signals for the control submodules, whereas more specific control signals (such as memory access signals) are generated inside the control submodules. Since the hardware modules can operate only with one particle (or two, in the case of the crossover unit), the control submodules are in charge of requesting data from the particle registers and then passing those data so that processing can be carried out. For the sake of brevity, these control submodules, i.e. local FSMs, have not been included in this document.

In the Particle Registers section of this chapter, some references were made to the fact that at least two memories for each state variable. There are two obvious reasons for this: on the one hand, the aforementioned overwriting problem when passing from one generation to the next one. With the implemented control strategy, particle states would suffer from data consistency errors were it not for the additional memories. On the other hand, these additional memories are important because they store the intermediate particle states. Moreover, since the particle registers are described as dual port RAMs, there is an extra interface available for data transactions. This advantage will be used in one of the testing platforms. Refer to the following chapter for further details on the usage of additional particle registers.

As far as this thesis is concerned, the process scheduling has been designed as sequential, i.e. each unit performs its function and then the next one, and so on. Therefore, full parallelism has not been accomplished (e.g. process model, crossover and mutation units processing data in parallel). This will be documented in the future lines sections of the following chapter.

IV. References

[1] Zhuohua Duan; Zixing Cai, "Adaptive Evolutionary Particle Filter Based Object Tracking with Occlusion Handling," *Natural Computation, 2009. ICNC '09. Fifth International Conference on* , vol.5, no., pp.358,361, 14-16 Aug. 2009

[2] Cong Li; Qin Honglei; Xing Juhong, "Distributed genetic resampling particle filter," *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on* , vol.2, no., pp.V2-32,V2-37, 20-22 Aug. 2010

[3] Xilinx, Inc., "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators," *XAPP 052,* July 7, 1996 (Version 1.1)

[4] Lee, D.-U.; Villasenor, J.D.; Luk, W.; Leong, P.H.W., "A hardware Gaussian noise generator using the Box-Muller method and its error analysis," *Computers, IEEE Transactions on* , vol.55, no.6, pp.659,671, June 2006

[5] Galal, S.; Pham, D., "Division Algorithms and Hardware Implementations," http://www.seas.ucla.edu/~ingrid/ee213a/lectures/division_presentV2.pdf

Results and Conclusions

I. Evolutionary Resampling Stage

Since the evolutionary resampling stage is one of the main contributions of this thesis, in this section a thorough analysis of the algorithm itself will be presented.

In order to validate the evolutionary resampling algorithm, it has been implemented in MATLAB. It is common to see validation examples in the literature in which the univariate non-stationary growth model is used as the process model equation. This is mainly due to the fact that it is highly non-linear, therefore making it suitable for testing any estimation tool which handles this specific type of systems (i.e. non-linear systems). Moreover, the measurement model is quadratic, thus conferring more complexity and uncertainty to the filtering process.

$$x_k = x_{k-1} + \frac{12 \cdot x_{k-1}}{1 + x_{k-1}^2} + 7 \cdot \cos\big(1.2 \cdot (k-1)\big) + \omega_{k-1}$$

$$z_k = \frac{x_k^2}{20} + \varphi_k$$

$$\omega_k \sim N(0, \sigma_x) \qquad \varphi_k \sim N(0, \sigma_z)$$

With this model, different tests have been proposed and performed. First, the Evolutionary Particle Filter was compared with the basic particle filter: the Bootstrap Filter. The aim of this test was to evaluate whether the proposed algorithm outperformed the Bootstrap Filter or not. The results of this test can be seen in both Fig. I.1 and Fig. I.2. In the first figure, the tracking performance is shown. Note that both algorithms provide accurate estimations. However, the Evolutionary Particle Filter provides, generally speaking, less estimation errors (computed as mean square errors), as it is shown in the second figure. In some time steps, especially in those of the beginning, the EPF estimation error has larger spikes than the Bootstrap Filter. Nevertheless, the rest of the spikes are, in average, smaller.

Another important test was carried out in order to measure the robustness of the system. Introducing large modifications in the dynamic evolution of the real system, the Bootstrap Filter lost the target trajectory, never to return, whereas the EPF lost the target and then returned to the real estimation value (since mutation operations generated children in the environment of the real state, therefore evolving the population to that state space region). The robustness of the system can be seen in Fig. I.3, where from $t = 20$ to $t = 40$, a linear model equation ($x_k = x_{k-1} + 7$) is used instead of the univariate non-stationary growth model. In a real-world situation, this divergence from the theoretical model could be due to inaccurate modeling strategies, i.e. the process model does not represent the actual behavior of the real system.

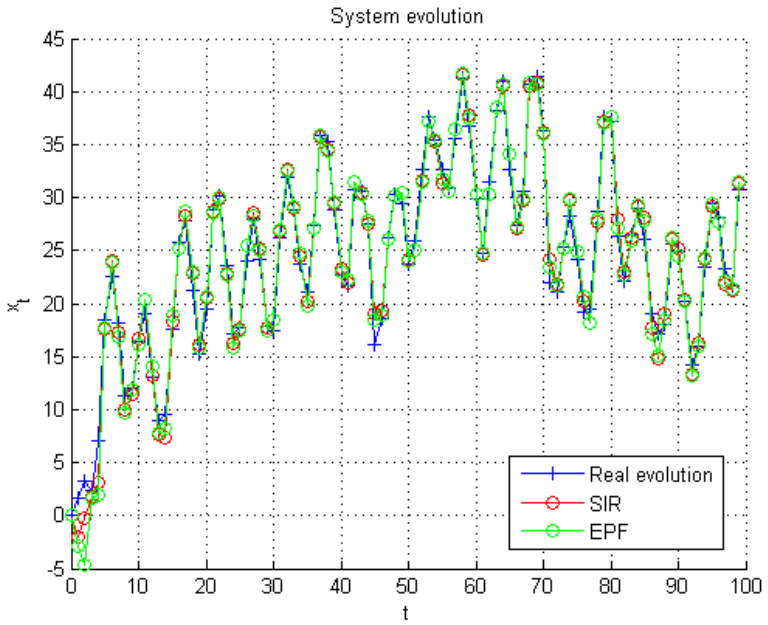


Fig. I.1. EPF vs. Bootstrap Filter: tracking performance

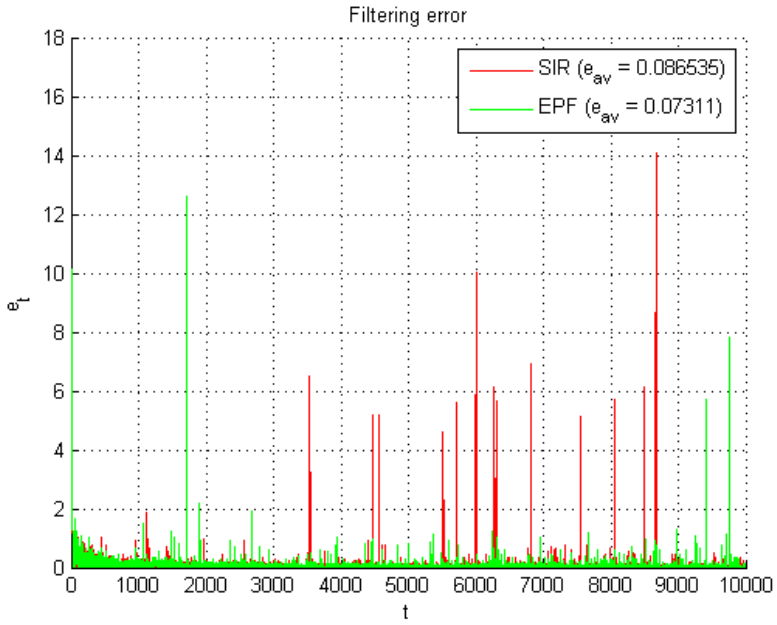


Fig. I.2. EPF vs. Bootstrap Filter: estimation error

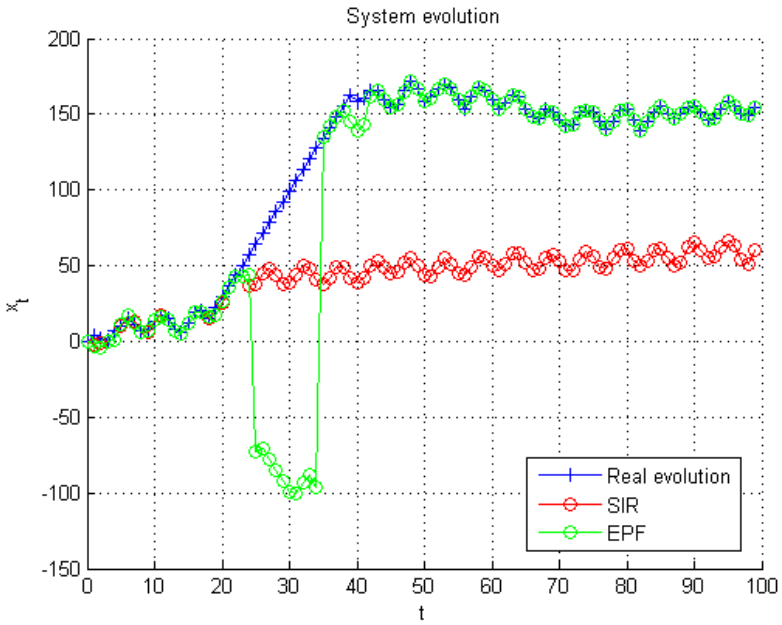


Fig. I.3. EPF vs. Bootstrap Filter: inaccurate process modeling

The last test that has been designed is focused on checking whether this resampling strategy can be considered optimal, rather than suboptimal. If the resampling stage algorithm is optimal, the sample impoverishment problem will be mitigated. Moreover, the other main problem of particle filtering, i.e. particle degeneracy, should not be present, since there is a resampling stage.

As in the previous tests, a comparison between the Bootstrap Filter and the proposed Evolutionary Particle Filter has been made. The effects of suboptimal resampling strategies, which have already been discussed in this thesis, are shown in Fig. I.4. The resampling stage in the Bootstrap Filter decreases particle diversity. However, the results of the EPF (shown in Fig. I.5) prove that genetic operations help keeping the population diversity. The posterior distribution still looks like a probability density function after resampling has been performed, as opposed to the Bootstrap Filter. The idea is that the particles "migrate" towards higher probability regions, instead of just being replaced or copied (which is basically what the suboptimal resampling stage in the Bootstrap Filter does).

The Evolutionary Particle Filter used in all these tests has been configured with 200 particles, $\sigma_x = \sigma_z = 2$, and with a limit of only two generations in the evolutionary algorithm. Each generation, new children are generated using 10 parents, and with $p_c = 0.8$ and $p_m = 0.1$ being the genetic operations probabilities. The generation limit has been established in order to further mitigate sample impoverishment.

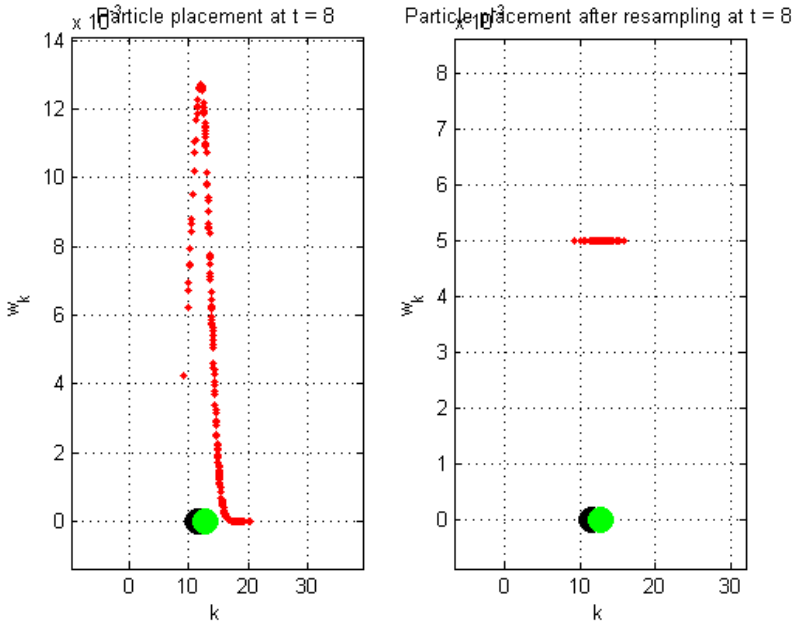


Fig. I.4. Bootstrap Filter: diversity loss due to suboptimal resampling

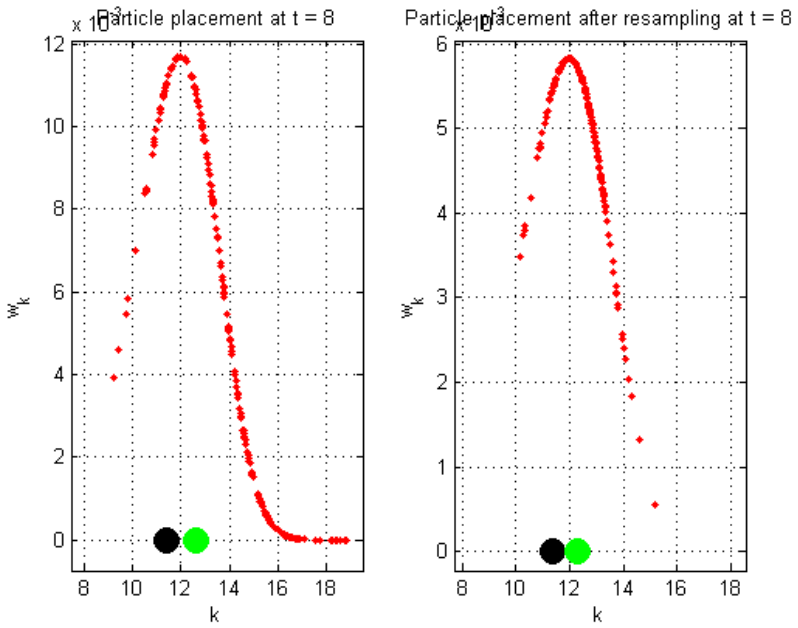


Fig. I.5. EPF: genetic operators keep population diversity

II. Random Number Generator

The proposed random number generator architecture has to be validated, especially the normal-distributed output. In this section, statistical analysis (e.g. histograms, autocorrelation and partial correlation functions) are performed on the output signals in order to check the functionality of the hardware module. The module configuration that has been used is the following one:

- Number of samples: statistical analysis require large sampling sizes in order not to be biased. Therefore, in these tests the total number of samples is set to one million.
- Data resolution: the uniform-distributed data output has a resolution of 8 bits (unsigned integer, 8-bit decimal part), whereas the normal-distributed output has a resolution of 19 bits (2's complement signed integer, 10-bit integer part, 8-bit decimal part).
- Probability density function parameters: based upon the definitions of the previous paragraph, the uniform-distributed output will generate a uniform distribution $U(0,1)$. The normal distribution parameters are specified in the generic section of the hardware module. For testing purposes only, these parameters have been set as follows: $\mu = 0; \sigma = 20$.

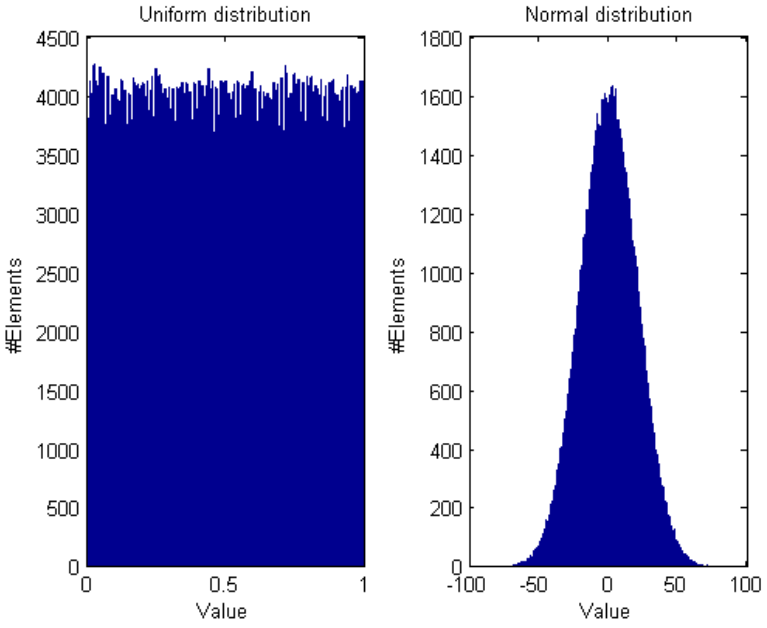


Fig. II.1. RNG output histograms

The first step in the validation process requires the histograms from both outputs to be computed. Therefore, after the simulation has finished, the data sets are processed with MATLAB in order to obtain their histograms. The outcome of these operations is shown in Fig. II.1.

Let us focus on the normal-distributed output first. Using a distribution-fitting tool, e.g. MATLAB integrated statistics toolbox, the aim is to find the normal distribution that best fits (i.e. represents) the data set. In Fig. II.2 it is possible to see that the data set looks like a normal distribution, and that the distribution-fitting tool provides a normal distribution that almost fits this data set. Note that some values are over the fitted normal distribution, whereas others are below. This phenomenon is due to the limited precision in output data signals.

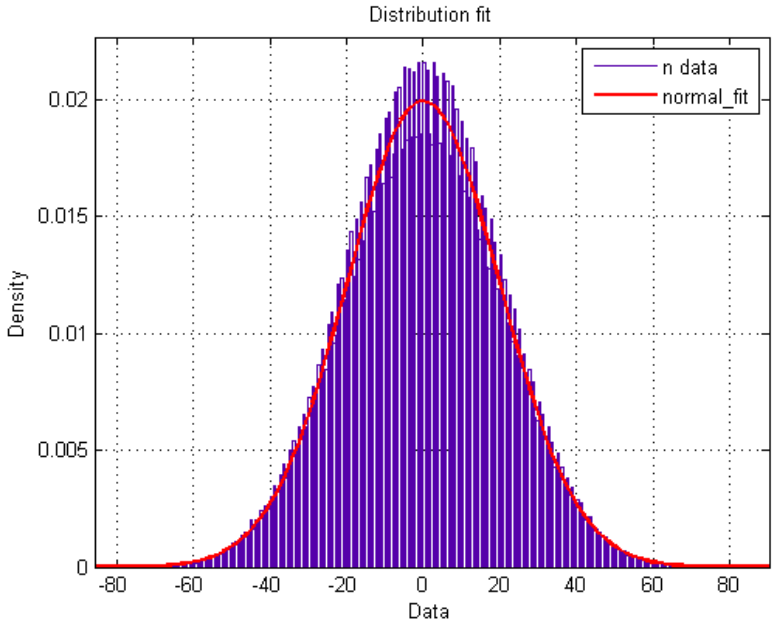


Fig. II.2. RNG normal output distribution fitting

In Fig. II.3, the parameters of the estimated fitting normal distribution are shown. Note that these values correspond with the specifications that were set at the beginning of this section.

Parameter	Estimate	Std. Err.
mu	-0.00698601	0.020004
sigma	20.004	0.014145

Fig. II.3. RNG normal output distribution fitting values

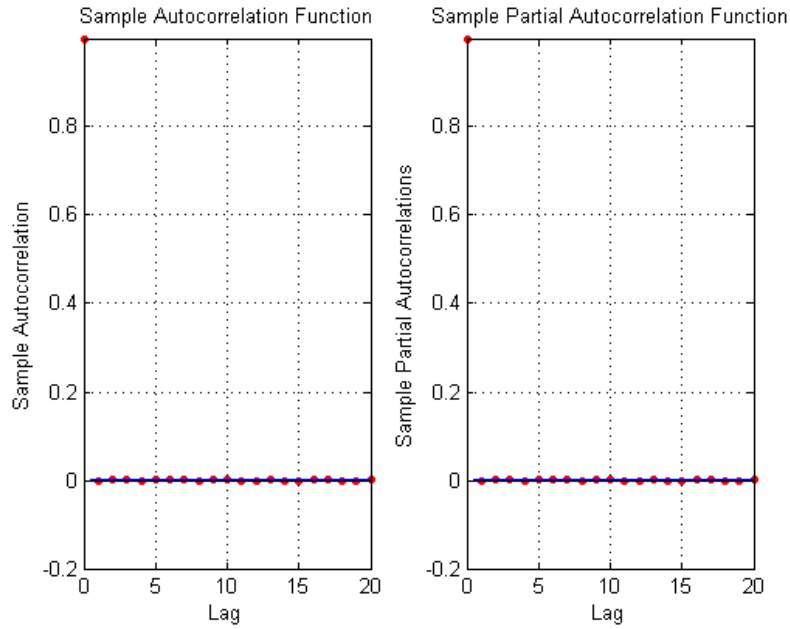


Fig. II.4. RNG normal output autocorrelation (left) and partial correlation (right) functions

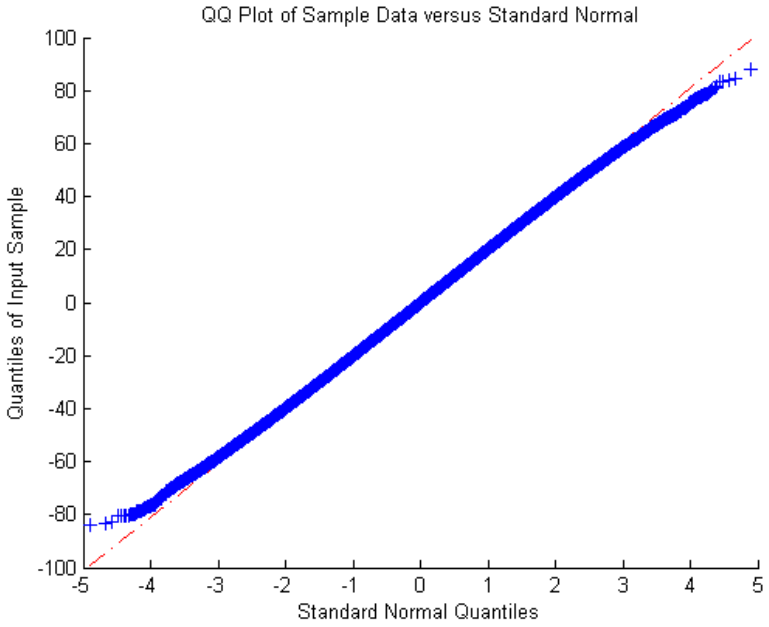


Fig. II.5. RNG normal output QQ Plot

So far, the configurable random number generator generates normal-distributed random samples with user-defined mean and standard deviation values. However, are these random samples good enough? The Evolutionary Particle Filter requires white noise signals, i.e. a stream of uncorrelated random samples with zero mean and finite standard deviation. Therefore, some tests have to be carried out in order to establish whether the normal output provides white noise signals or not. Correlation between samples can be analyzed using autocorrelation and partial correlation functions. This is what has been presented in Fig. II.4. This output has been confirmed as Gaussian (Fig. II.2). However, further tests can be done. For instance, in Fig. II.5, the QQ plot of the data set is shown. This analysis compares two distributions; in this case, the one defined by a data set, and the reference normal distribution (computed with the distribution fitting tool). Note that both distributions are very similar, except in the edges. Again, this is a natural consequence of the limited data precision in the system, which is due to the hardware implementation.

Correlation analysis have also been performed on the other output, the uniform distributed data output. Results from this test have been provided in Fig. II.6. As it can be seen, random samples drawn from this output are not correlated at all.

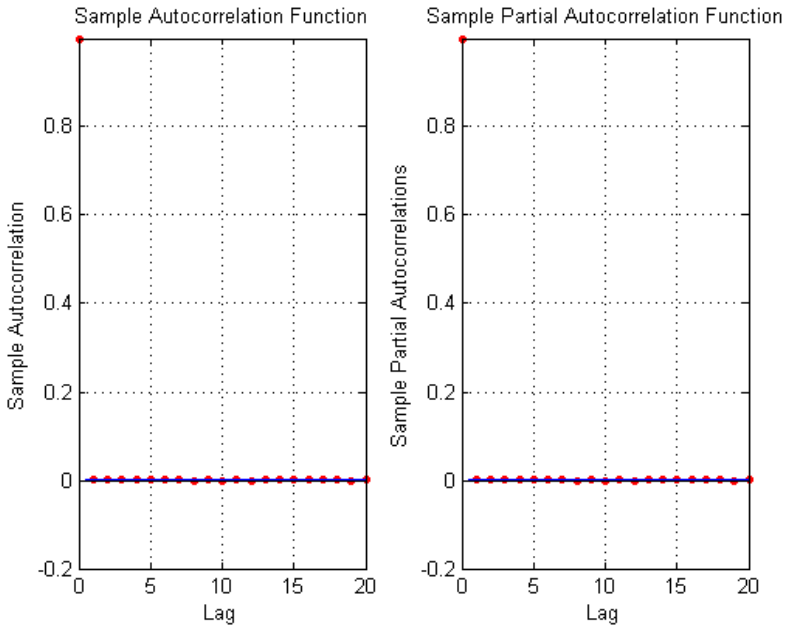


Fig. II.6. RNG uniform output autocorrelation (left) and partial correlation (right) functions

All this validation tests have been carried out through simulation, where the data were written to text files from the testbench itself. These text files were then loaded in MATLAB in order to perform the required computations.

III. Fitness Calculation

In this section, the functional validation of the fitness calculation unit is presented. The module configuration that has been used is the following one:

- Input data resolution: 19 bits (2's complement signed integer, 10-bit integer part, 8-bit decimal part).
- Mean values are introduced into the module using external signals, i.e. the specific input ports *x_mean* and *y_mean*.
- Standard deviations: $\sigma_x = \sigma_y = 10$.
- Look-up table elements: the LUT has been built using a 32x32 matrix (Fig. III.1).

The resulting LUT mapping can be seen in Fig. III.2. Note that the resolution of the look-up table has been increased, since only first-quadrant points are computed, as discussed in the previous chapter. With this LUT, and setting the external signals so that $\mu_x = 10$ and $\mu_y = -50$, the fitness values of the input signals can be seen in Fig. III.3. Two-dimensional representations can also be found in Fig. III.4 and Fig. III.5. Note that he peak fitness values are obtained when $x = \mu_x$ and $y = \mu_y$. Also notice that all four quadrants have been successfully reconstructed.

Fig. III.1. LUT step values

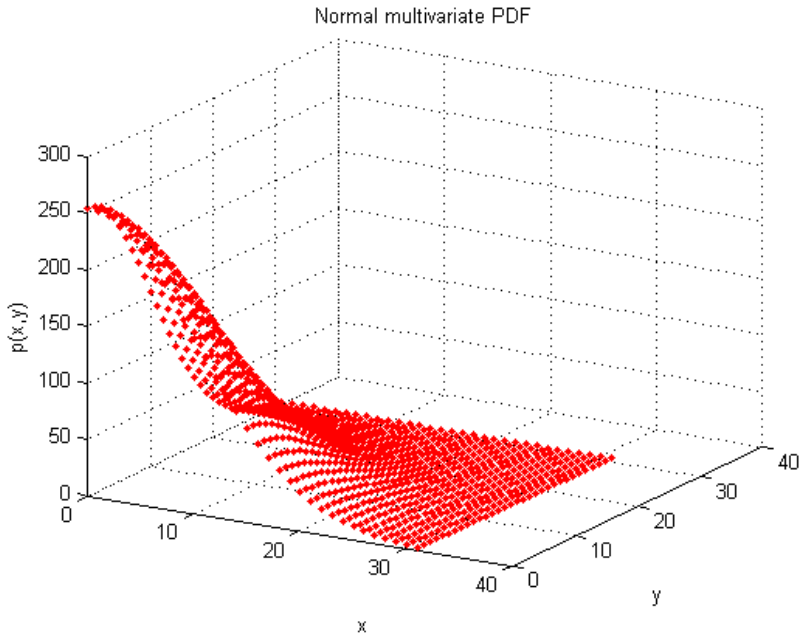


Fig. III.2. Fitness calculation LUT

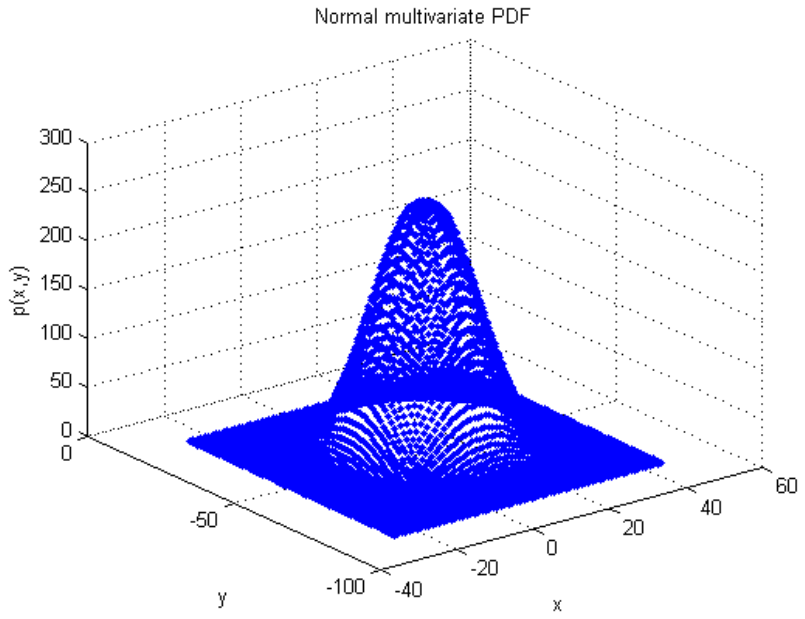


Fig. III.3. Normal PDF 3D computation using the LUT

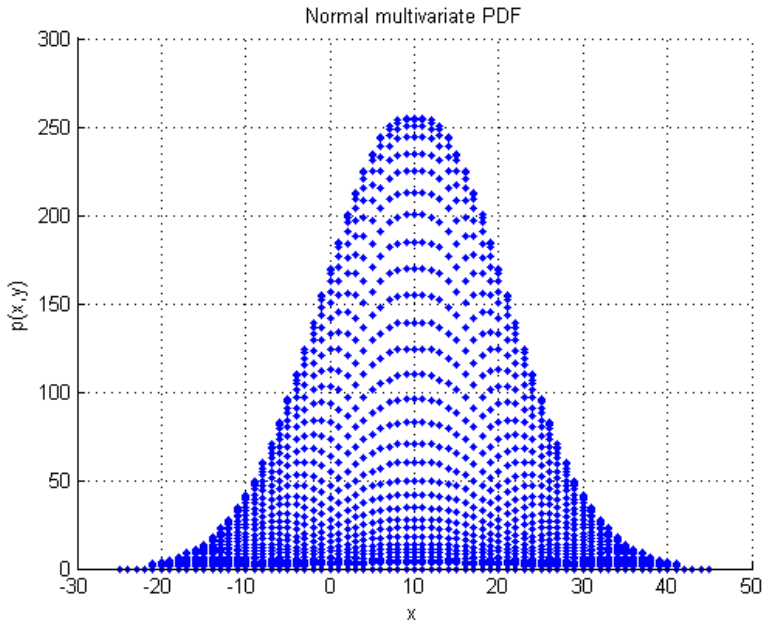


Fig. III.4. Normal PDF computation (xz-axis)

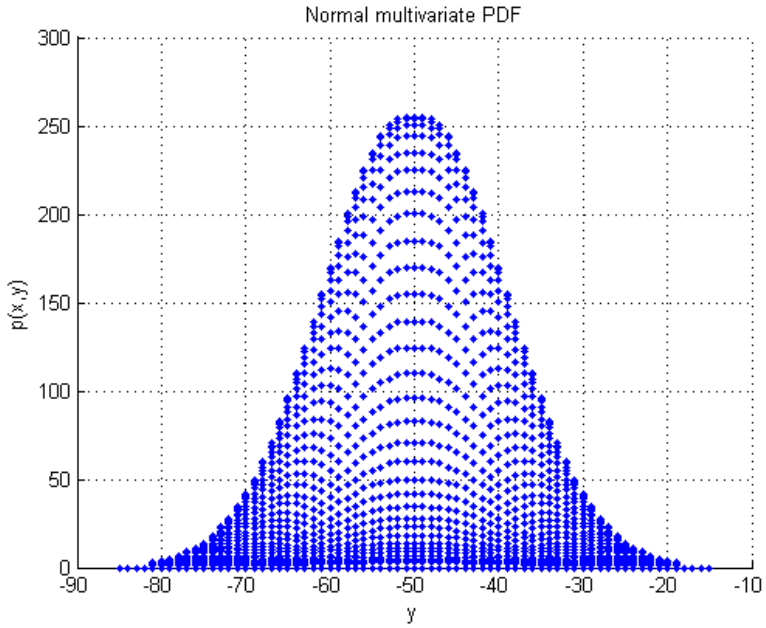


Fig. III.5. Normal PDF computation (yz-axis)

In Fig. III.4 and in Fig. III.5 it is also possible to see the other proposed enhancement: the effective input data size. Every point that falls outside the circle defined by $3 \cdot \sigma_{max}$ has a fitness value of zero.

As in the previous section, this results have been obtained from simulations. Fitness values are written to a text file from the testbench, and then MATLAB computes the figures.

IV. Experimental Methodologies

The Evolutionary Particle Filter has been implemented in hardware using two different methodologies.

- Hardware In the Loop (HIL): the FPGA board is used as a coprocessor. The main processor is inside a personal computer, and performs the preprocessing operations on the input image.
- System on Programmable Chip (SoPC): the EPF is included as a peripheral in an embedded system (which also has a microprocessor) inside the FPGA. In this approach, there is no preprocessing stage, since it has yet to be developed.

These two approaches are described in more detail in the forthcoming sections of this chapter. Furthermore, the reasons for including both strategies will be presented.

IV.1 Hardware In the Loop (HIL)

Although the Evolutionary Particle Filter is a complex system, it is only a module within a larger and even more complex system. However, the design of the rest of the processing system lies out of the scope of this thesis. Therefore, hardware-in-the-loop simulations seem a feasible alternative so that the whole application can be tested and validated.

The HIL validation platform that has been used appears in Fig. IV.1, and consists of a personal computer, a USB webcam and a FPGA board featuring a Xilinx Virtex-5.

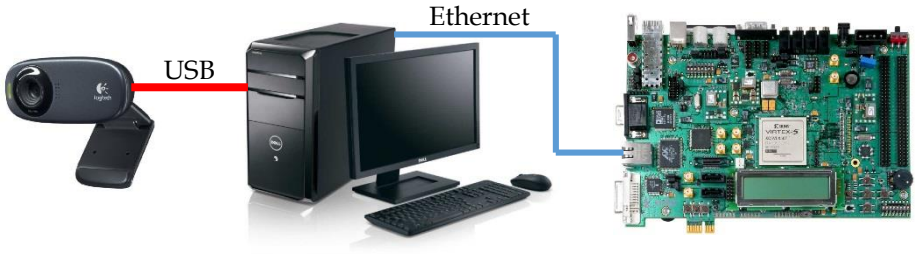


Fig. IV.1. HIL validation platform setup

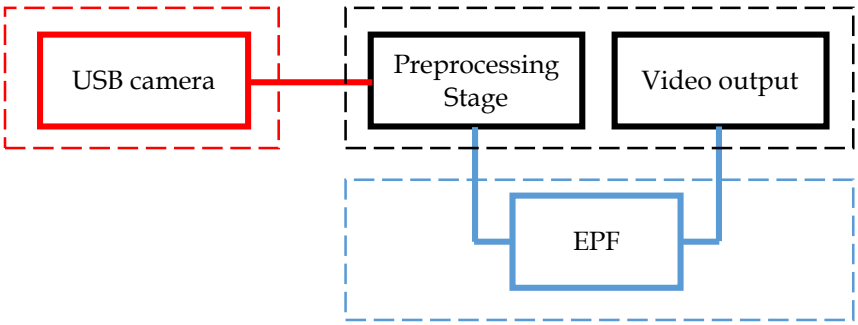


Fig. IV.2. HIL block diagram

In order to implement the system described with the block diagram in Fig. IV.2, both Simulink and Xilinx development tools have been used. Simulink toolboxes provide a wide range of application-specific blocks that are suitable for image processing applications. Furthermore, the HIL simulation platform takes advantage of these processing blocks.

Since a specific target application was required in order to have a functional demonstrator, a simple object tracking application was proposed. In this application, red objects are the target that has to be tracked. With these requirements, the preprocessing stage has to perform the following operations:

1. Read a RGB image from the USB camera (block: From Video Device; blockset: Image Acquisition Toolbox).
2. Transform the RGB representation into YCbCr axes (block: Color Space Conversion; blockset: Computer Vision System Toolbox).
3. Threshold the image to obtain those pixels in which the property $C_r > 0.65$ is satisfied (blocks: Constant, Relational Operator; blockset: Simulink).
4. Perform a closing operation on the binary image in order to remove small holes (block: Closing; blockset: Computer Vision System Toolbox).
5. Perform a blob analysis so that all red elements can be identified. Then, obtain the centroid, bounding box and area of each identified object (block: Blob Analysis; blockset: Computer Vision System Toolbox).
6. Select the largest identified element, and obtain its current position in the image (block: MATLAB Function; blockset: Simulink).

VHDL files have been imported to the Simulink project using Xilinx System Generator. This process has the following steps:

1. Create a top-level VHDL wrapper for the logic, in which there must be a clock enable signal. This port might not be used inside the wrapper.

2. Create a new Simulink model, and add as many input-output gateways as needed, i.e. one gateway per input-output port of the VHDL entity that the user wants to connect to the computer (block: Gateway In, Gateway Out; blockset: Xilinx Blockset).
3. Create a black-box instance, and configure it with the top-level VHDL wrapper. Then, edit the black-box configuration file (.m) including all the low-level VHDL files (this is similar to editing the .pao when designing in Xilinx EDK), and the ports width and data type (block: Black Box; blockset: Xilinx Blockset).
4. Set the compilation target as hardware co-simulation, and generate FPGA programming file (block: System Generator; blockset: Xilinx Blockset).
5. Instantiate the result of the compilation process in the Simulink model in which the preprocessing stage has been implemented.

Two different options for hardware co-simulation are provided within Xilinx System Generator blockset. The first one performs co-simulation through a JTAG interface, i.e. a serial communication; therefore, data transactions create a bottleneck and increase significantly simulation times. The other alternative is to use an Ethernet connection, which provides higher bandwidth (especially if Gigabit Ethernet is used). Although System Generator supports different FPGA development boards, the XUPV5 is not among them. Therefore, both interfaces had to be defined and configured. The JTAG interface configuration is really simple, and thus it will not be explained here. The Ethernet connection, on the other hand, has required much more effort in order to work properly. The configuration files and scripts for a similar board, which featured a different Virtex-5 device, have been modified, introducing slight changes that allowed hardware co-simulation with the XUPV5 as target board.

Synchronization problems in some control signals (e.g. new measurement and valid measurement) forced to create intermediate hardware modules which acted as if they were drivers. These synchronization problems were mostly generated due to serial data transmission within the Ethernet framework. Simulink sends control signals in different Ethernet packets, even if they change at the very same time. Therefore, the hardware experiences these changes in the control signals with the delay that exists between two consecutive Ethernet frames. This has been confirmed using Xilinx ChipScope to monitor these control signals. JTAG co-simulation does not allow signal monitoring, since it also uses JTAG communication. Therefore, Ethernet co-simulation was chosen for this reason, as well as for the reduction in terms of simulation time.

The results of the hardware co-simulation can be seen in Fig. IV.3. In this output image, a red pen is followed through the region of interest. The actual measurement is represented with a green rectangle, whereas the estimation provided by the Evolutionary Particle Filter corresponds to the blue rectangle.

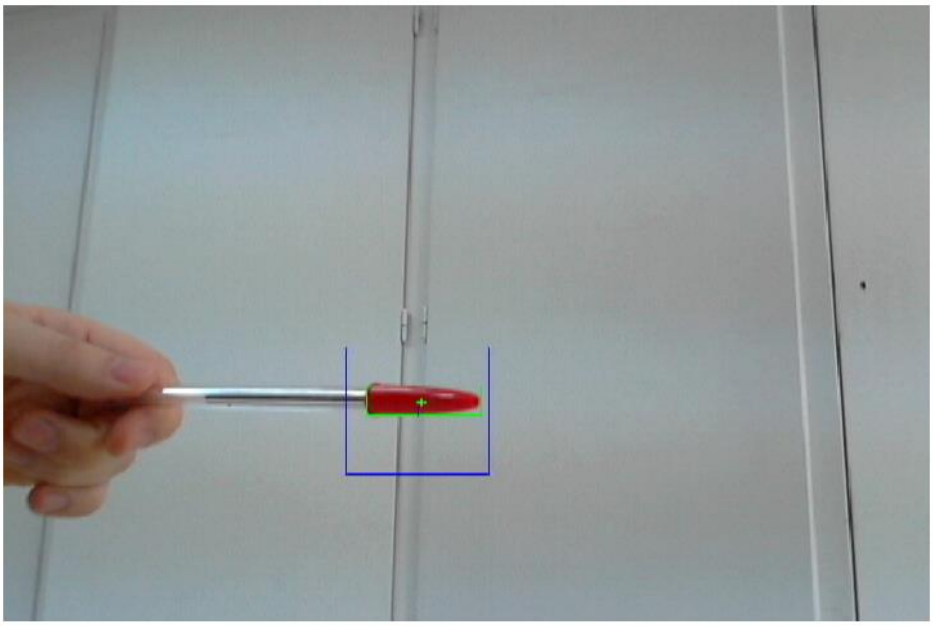


Fig. IV.3. HIL demo

Estimation evolution through time can be seen in the six consecutive frames that have been included in Fig. IV.4.

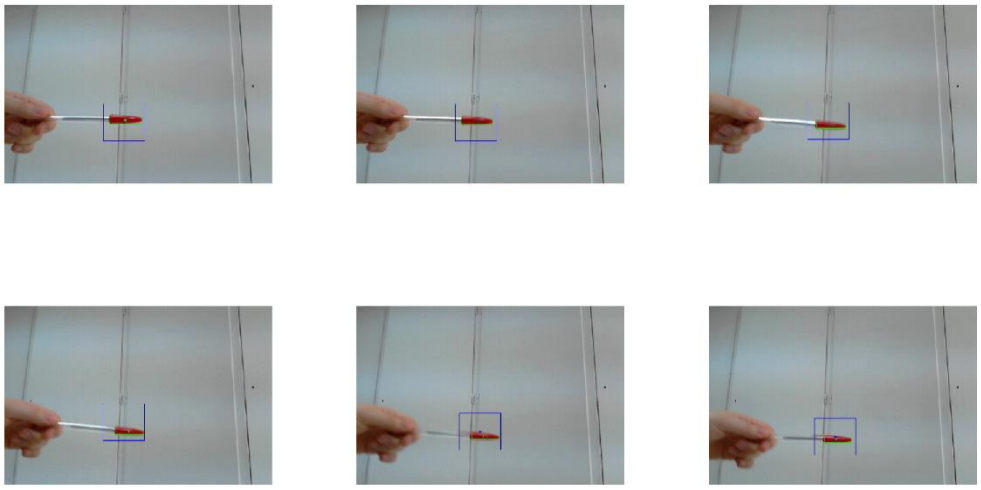


Fig. IV.4. HIL filter estimation evolution

All this operations, toolboxes and functions have been implemented using MATLAB R2013a and Xilinx ISE Design Suite 14.6.

IV.2 System on Programmable Chip (SoPC)

One of the potential target applications of the Evolutionary Particle Filter is autonomous navigation. It is in this specific type of system where embedded systems play an important role. Although HIL simulations are a good validation technique, they are not feasible in embedded-system approaches.

In this section, a SoPC architecture that includes the whole tracking system (i.e. image acquisition module, preprocessing stage, particle filter and video output) within one single chip is presented. This architecture can be seen in Fig. IV.5. Note that there is some part of the system that has been highlighted in red. All components that fall within this red area are out of the scope of this thesis, and therefore they have not been implemented. Refer to the future lines section for further information on this issue.

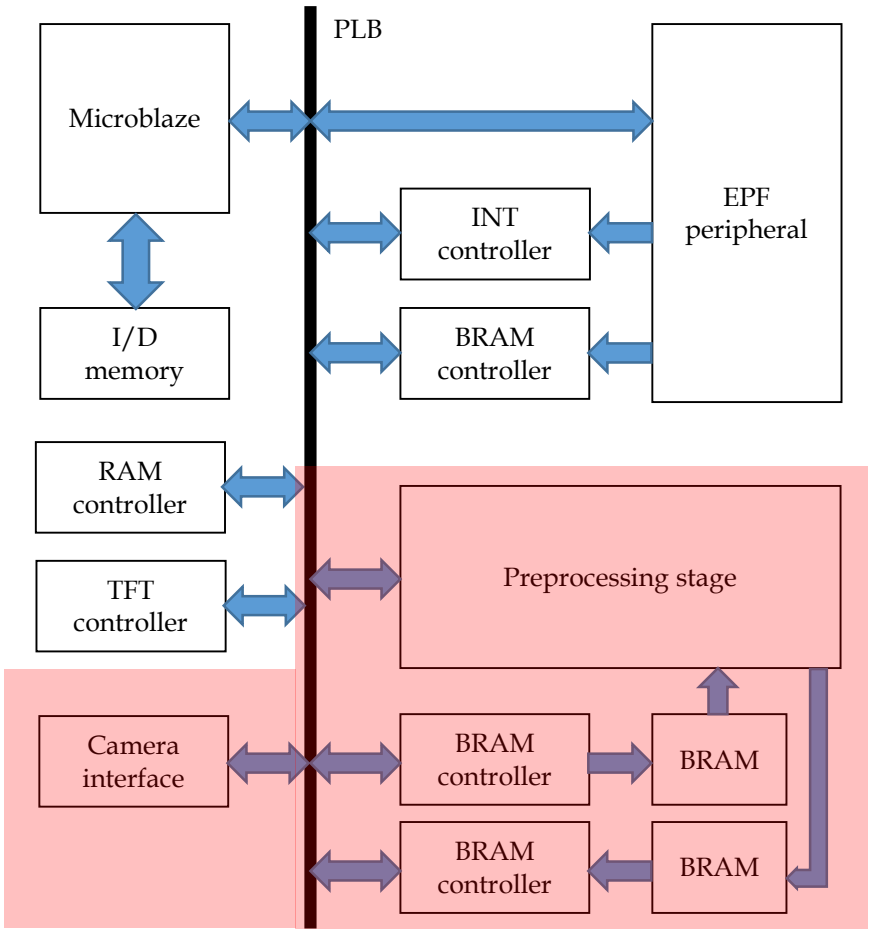


Fig. IV.5. SoPC architecture (block diagram)

The embedded system consists of a microprocessor (Xilinx MicroBlaze soft-core processor), and some peripherals connected through a bus interface (Processor Local Bus or PLB).

- TFT controller: handles the video output signals, and represents them in a VGA screen.
- RAM controller: since the TFT memory map is stored in an external RAM device, it is necessary to include the interface signals with this external memory chip. These signals will be handled within this IP core.
- INT controller: handles interrupt signals coming from different peripherals within the system. In this case, only interrupt signals generated at the EPF peripheral will be valid.
- BRAM controller: the MicroBlaze has access to the BRAM memory connected to this intermediate block.
- Evolutionary Particle Filter peripheral: top-level VHDL wrapper for the EPF. This block contains the architecture proposed in the previous chapter and additional logic (e.g. 32-bit register bank communication with the MicroBlaze, PLB slave interface, interrupt controller interface, etc.). The internal architecture of this IP core is shown in Fig. IV.6. Note that the interfaces have been simplified in this diagram (e.g. the interrupt bus, which has two different interrupt signals, or the z_{ctrl} bus, which contains both new measurement and valid measurement control signals). Also, notice that not all registers in the bank allow write operations from the MicroBlaze.

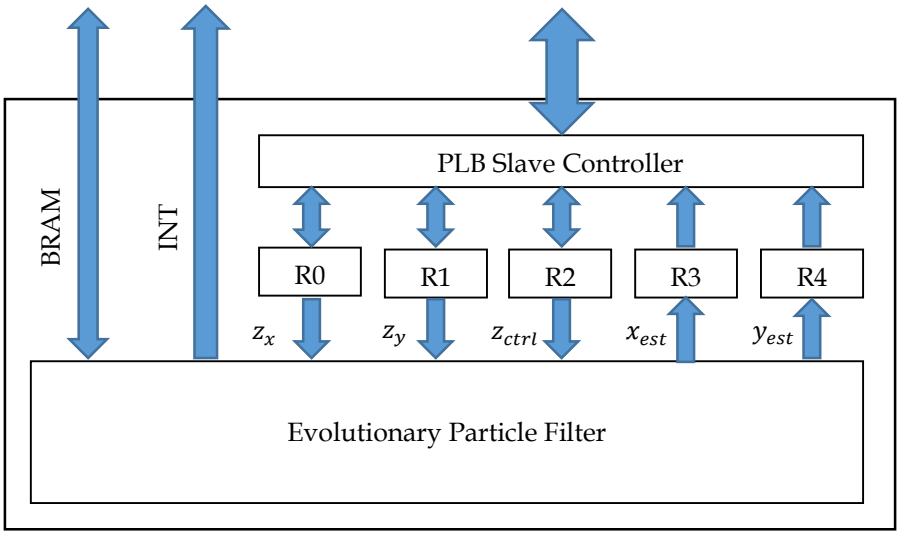


Fig. IV.6. EPF peripheral interfaces

The normal operation cycle of this embedded system would be defined in a software application running on the Microblaze. This application has the following steps:

1. Initialize the system. Set initial configuration in each peripheral.
2. Acquire an image from the video camera and store it in a BRAM memory.
3. Preprocess the image (e.g. edge detection, corner detection, color detection, etc.) in order to obtain a valid pair of XY coordinates, and store the processed image in another BRAM memory.
4. Write these XY values into registers R0 and R1 in EPF peripheral. If this measurement is valid, i.e. it is within the region of interest, write 0x3 into R3; otherwise, write 0x1.
5. Wait for the valid data EPF interruption to occur. Then, read the estimated value of the state variables from registers R3 and R4, and reset register R2 to 0x0.
6. Write all required data in the external RAM in order to show the estimation process in the VGA screen. Then, go back to step 2.

Since the complete tracking system is not implemented yet, only steps 4 to 6 can be performed. The results presented in forthcoming sections are based in this unfinished architecture, controlled with a software application with the limitations above mentioned.

V. Timing and Resource Occupation

Timing and area constraints are very important conditions in digital circuit designs. In this section, the maximum allowable frequency in each hardware module, as well as in the Evolutionary Particle Filter as a system, is presented. In addition, device utilization rates for each hardware component are also included.

The results provided within this section may help developing optimization processes over the basic architecture. For instance, if the maximum allowable frequency in the EPF is lower than in each individual component, the optimization process should be focused on the interconnections and the control logic. Moreover, device utilization rates could help identifying those parts of the digital design that can be simplified, in order to use fewer resources. A good example of this resource optimization strategy can be found in the process model unit. The general approach that has been proposed in this thesis generates four different multipliers (Process Model). However, if there are area limitations, this general structure can be revised in order to obtain a more specific architecture, without multipliers (since the matrix coefficients are 0 and 1).

The configuration parameters that have been used to obtain the results in this section have been included in a VHDL package, and can be found in Fig. V.1.

```
package particle_filter_pkg is

  -- PARTICLE REGISTER PARAMETERS
  -- Number of particles
  constant PARTICLES   : natural := 200;
  -- Data resolution in bits
  constant INT_PART    : natural := 10;
  constant DEC_PART    : natural := 8;
  -- Fitness resolution in bits
  constant FIT_WIDTH   : natural := 8;
  -- Number of parents
  constant PARENTS     : natural := 10;
  -- Number of generations
  constant GENERATIONS : natural := 2;
  -- External BRAM interface
  constant MEM_ADDR_WIDTH : natural := 32;
  constant MEM_DATA_WIDTH : natural := 32;
  -- RANDOM GENERATOR PARAMETERS
  -- LFSR seed
  constant SEED        : std_logic_vector(110 downto 0) := (0 => '0',
others => '1');
  -- Normal distribution mean value
  constant MEAN        : integer := 0;
  -- Normal distribution standard deviation (2**n to reduce the number
of multipliers)
  constant STD_DEV     : natural := 32;
  -- Output bits in uniform distribution
  constant OUTPUT_BITS : natural := DEC_PART;
  -- NORMAL EVALUATION PARAMETERS
  -- Normal distribution parameters
  constant MEAN_X      : integer := 0;
  constant STD_DEV_X   : natural := 10;
  constant MEAN_Y      : integer := 0;
  constant STD_DEV_Y   : natural := 10;
  constant VALUES      : natural := 32;
  -- Use external mean value
  constant EXT_MEAN    : boolean := true;
  -- PROCESS MODEL PARAMETERS
  -- sigma_pos/sigma_vel
  constant SIGMA_RATIO_V : real := real(STD_DEV)/0.01;
  -- Sampling time
  constant T             : real := 0.033;
  -- CROSSOVER PARAMETERS
  -- Crossover probability
  constant P_CROSS : real := 0.6;
  -- MUTATION PARAMETERS
  -- sigma pos/sigma mut
  constant SIGMA_RATIO_M : real := real(STD_DEV)/6.0;--0.5;
  -- Mutation probability
  constant P_MUT         : real := 0.1;
  -- Mutation mode ratio (random placement/local placement)
  constant MUT_RATIO     : real := 0.4;

end particle_filter_pkg;
```

Fig. V.1. VHDL configuration package

V.1 Timing

VHDL module	Maximum frequency (MHz)
RNG	135.080
Fitness calculation	-
State register	-
Process model	95.716
Crossover unit	113.097
Mutation unit	99.885
Divider	227.376
EPF	63.914

Table V.1. Timing results

Table V.1 shows the maximum reachable frequencies for each hardware module. Note that both fitness calculation and state register do not have any value. In the former, this is due to the fact that it is a combinational logic block, whereas in the latter it is because BRAMs are instantiated as a component. Also, note that the integrated Evolutionary Particle Filter limits the maximum operating frequency.

```
Delay:       15.646ns (Levels of Logic = 41)
Source:      mutation_inst/y_c_6 (FF)
Destination: particle_register_inst/fit_state/Mram_state (RAM)
```

As far as this thesis is concerned, this relatively low frequency is not a problem. However, further optimization tasks regarding this parameter must be carried out, e.g. pipelining in order to reduce the critical path, which in this case has 41 levels of logic. Refer to future lines for further information on this topic.

V.2 Resource Occupation

V.2.1 Random Number Generator

```
Macro Statistics
# Adders/Subtractors                                   : 12
 10-bit adder carry out                                : 1
 11-bit adder carry out                                : 1
 14-bit subtractor                                     : 1
 8-bit adder carry out                                 : 6
 9-bit adder carry out                                 : 3
# Registers                                            : 131
 Flip-Flops                                            : 131
# Xors                                                 : 96
 1-bit xor2                                            : 96
```

V.2.2 Fitness Calculation

```
Macro Statistics
# ROMs                                                 : 1
 1024x8-bit ROM                                        : 1
# Multipliers                                          : 2
 15x6-bit multiplier                                   : 2
# Adders/Subtractors                                   : 15
 13-bit subtractor                                     : 2
 15-bit adder                                          : 2
 18-bit adder                                          : 2
 19-bit subtractor                                     : 2
 20-bit adder                                          : 2
 20-bit subtractor                                     : 2
 32-bit adder carry out                                : 1
 5-bit adder                                           : 2
# Comparators                                          : 2
 21-bit comparator greatequal                          : 2
```

V.2.3 State Register

```
Synthesizing (advanced) Unit <state_register>.
INFO:Xst:3226 - The RAM <Mram_state> will be implemented as a BLOCK
RAM, absorbing the following register(s): <dout_a> <dout_b>
    -----------------------------------------------------------------------
    | ram type           | Block                                |          |
    -----------------------------------------------------------------------
    | Port A                                                               |
    |     aspect ratio   | 200-word x 19-bit                    |          |
    |     mode           | read-first                           |          |
    |     clkA           | connected to signal <clk a>          | rise     |
    |     enA            | connected to signal <en a>           | high     |
    |     weA            | connected to signal <we_a>           | high     |
    |     addrA          | connected to signal <addr_a>         |          |
    |     diA            | connected to signal <din a>          |          |
    |     doA            | connected to signal <dout a>         |          |
    -----------------------------------------------------------------------
    | optimization       | speed                                |          |
    -----------------------------------------------------------------------
    | Port B                                                               |
    |     aspect ratio   | 200-word x 19-bit                    |          |
    |     mode           | read-first                           |          |
    |     clkB           | connected to signal <clk_b>          | rise     |
    |     enB            | connected to signal <en_b>           | high     |
    |     weB            | connected to signal <we_b>           | high     |
    |     addrB          | connected to signal <addr b>         |          |
    |     diB            | connected to signal <din b>          |          |
    |     doB            | connected to signal <dout b>         |          |
    -----------------------------------------------------------------------
    | optimization       | speed                                |          |
    -----------------------------------------------------------------------
Unit <state_register> synthesized (advanced).
```

```
Macro Statistics
# RAMs                                  : 1
 200x19-bit dual-port block RAM         : 1
# Registers                             : 19
 Flip-Flops                             : 19
```

V.2.4 Process Model

```
Macro Statistics
# FSMs                                  : 1
# Multipliers                           : 4
 10x19-bit multiplier                   : 3
 19x19-bit multiplier                   : 1
# Adders/Subtractors                    : 4
 38-bit adder                           : 4
# Registers                             : 77
 Flip-Flops                             : 77
```

V.2.5 Crossover Unit

```
Macro Statistics
# FSMs                                  : 1
# Multipliers                           : 4
 10x19-bit multiplier                   : 2
 9x19-bit multiplier                    : 2
# Adders/Subtractors                    : 3
 38-bit adder                           : 2
 9-bit adder                            : 1
# Registers                             : 154
 Flip-Flops                             : 154
# Comparators                           : 1
 8-bit comparator less                  : 1
```

V.2.6 Mutation Unit

```
Macro Statistics
# FSMs                                  : 1
# Multipliers                           : 1
 9x38-bit multiplier                    : 1
# Adders/Subtractors                    : 2
 19-bit adder                           : 1
 76-bit adder                           : 1
# Registers                             : 79
 Flip-Flops                             : 79
# Comparators                           : 2
 8-bit comparator less                  : 2
```

V.2.7 Divider

```
Macro Statistics
# FSMs                                          : 1
# Adders/Subtractors                            : 8
 28-bit adder                                   : 2
 28-bit adder carry out                         : 1
 28-bit addsub                                  : 1
 29-bit adder                                   : 2
 29-bit adder carry out                         : 1
 30-bit adder                                   : 1
# Counters                                      : 1
 6-bit up counter                               : 1
# Registers                                     : 145
 Flip-Flops                                     : 145
# Comparators                                   : 4
 28-bit comparator less                         : 4
```

V.2.8 Evolutionary Particle Filter

```
Macro Statistics
# FSMs                                          : 17
# RAMs                                          : 10
 200x19-bit dual-port block RAM                 : 2
 200x32-bit dual-port block RAM                 : 2
 200x8-bit single-port block RAM                : 1
 300x19-bit dual-port block RAM                 : 4
 300x8-bit dual-port block RAM                  : 1
# ROMs                                          : 1
 1024x8-bit ROM                                 : 1
# Multipliers                                   : 17
 10x19-bit multiplier                           : 5
 15x6-bit multiplier                            : 2
 17x17-bit multiplier                           : 2
 17x9-bit multiplier                            : 1
 19x19-bit multiplier                           : 2
 19x9-bit multiplier                            : 2
 9x19-bit multiplier                            : 2
 9x38-bit multiplier                            : 1
# Adders/Subtractors                            : 73
 10-bit adder carry out                         : 1
 10-bit subtractor                              : 1
 11-bit adder carry out                         : 1
 13-bit subtractor                              : 2
 14-bit subtractor                              : 1
 15-bit adder                                   : 2
```

```
 17-bit adder                                : 3
 18-bit adder                                : 2
 19-bit adder                                : 3
 19-bit subtractor                           : 2
 20-bit adder                                : 2
 20-bit subtractor                           : 2
 28-bit adder                                : 6
 28-bit adder carry out                      : 2
 28-bit addsub                               : 2
 29-bit adder                                : 4
 29-bit adder carry out                      : 2
 30-bit adder                                : 2
 32-bit adder carry out                      : 1
 38-bit adder                                : 6
 5-bit adder                                 : 2
 76-bit adder                                : 1
 8-bit adder carry out                       : 6
 8-bit subtractor                            : 1
 9-bit adder                                 : 9
 9-bit adder carry out                       : 3
 9-bit subtractor                            : 4
# Counters                                   : 16
 2-bit up counter                            : 1
 6-bit up counter                            : 2
 9-bit down counter                          : 1
 9-bit up counter                            : 12
# Registers                                  : 8380
 Flip-Flops                                  : 8380
# Comparators                                : 20
 10-bit comparator equal                     : 1
 10-bit comparator not equal                 : 1
 17-bit comparator greatequal                : 1
 17-bit comparator less                      : 1
 21-bit comparator greatequal                : 2
 28-bit comparator less                      : 8
 8-bit comparator less                       : 4
 9-bit comparator equal                      : 2
# Multiplexers                               : 5
 17-bit 300-to-1 multiplexer                 : 1
 9-bit 10-to-1 multiplexer                   : 3
 9-bit 200-to-1 multiplexer                  : 1
# Xors                                       : 96
 1-bit xor2                                  : 96
```

```
Device utilization summary:
---------------------------

Selected Device : 5vlx110tff1136-1


Slice Logic Utilization:
 Number of Slice Registers:            8432  out of  69120    12%
 Number of Slice LUTs:                13064  out of  69120    18%
    Number used as Logic:             13064  out of  69120    18%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:  15401
   Number with an unused Flip Flop:    6969  out of  15401    45%
   Number with an unused LUT:          2337  out of  15401    15%
   Number of fully used LUT-FF pairs:  6095  out of  15401    39%
   Number of unique control sets:       267

IO Utilization:
 Number of IOs:                        280
 Number of bonded IOBs:                232  out of    640    36%

Specific Feature Utilization:
 Number of Block RAM/FIFO:               9  out of    148     6%
    Number using Block RAM only:         9
 Number of BUFG/BUFGCTRLs:               4  out of     32    12%
 Number of DSP48Es:                     17  out of     64    26%
```

VI. Sensitivity Analysis

The Evolutionary Particle Filter has a large set of configuration parameters. The user can change these parameters only at synthesis time, since most of them set the size of the system (e.g., the size of the internal memories is determined using the number of particles and the number of parents).

In this section, a sensitivity analysis is carried out. The idea is to establish the relationship (if it exists) between the estimation error and one of the system parameters, while the rest of the parameters remain constant. The goal of these tests is to determine which parameters, if any, have larger impact on the system overall performance.

Generally speaking, the system parameters can be divided into two main groups: the particle filter parameters and the evolutionary algorithm parameters.

- Particle filter parameters: σ_{pos}, σ_{vel}, σ_{meas}, sampling time (T), number of LUT values, and number of particles (i.e. population size).
- Evolutionary algorithm parameters: number of particles, number of parents, number of generations, p_{cross}, p_{mut}, r_{mut}, and σ_{mut}.

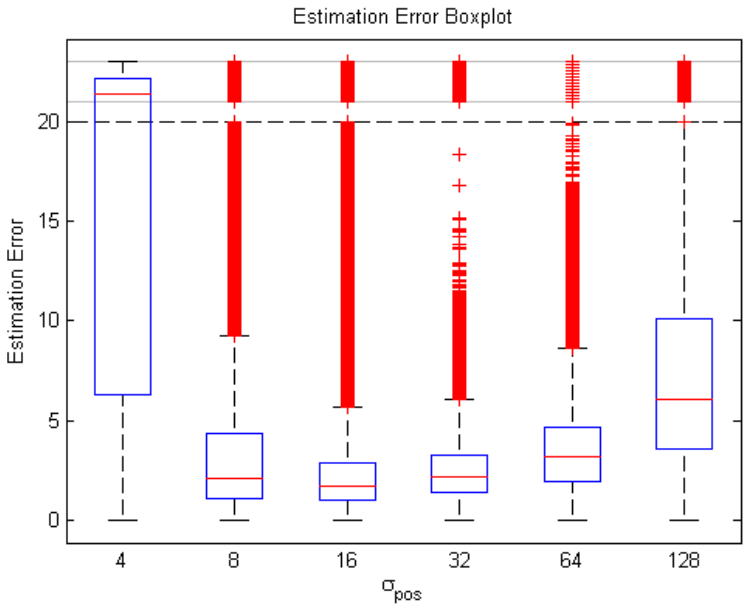


Fig. VI.1. Estimation error vs.σ_{pos}

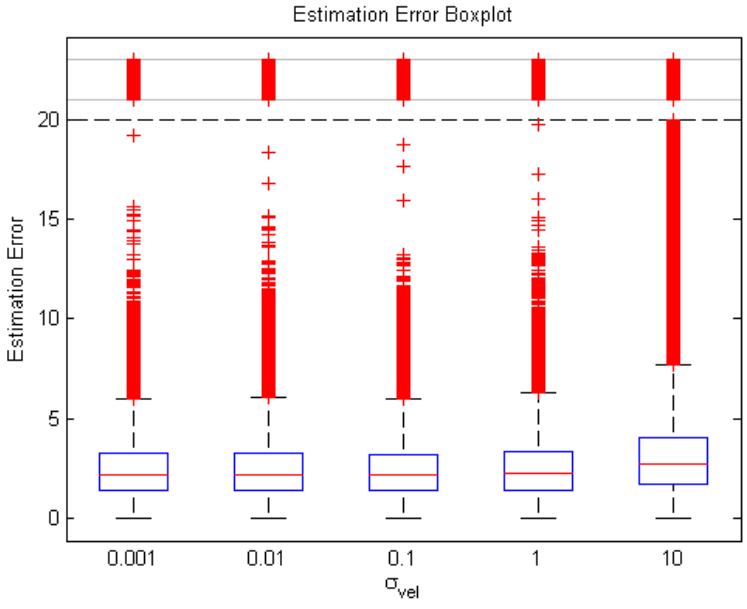


Fig. VI.2. Estimation error vs. σ_{vel}

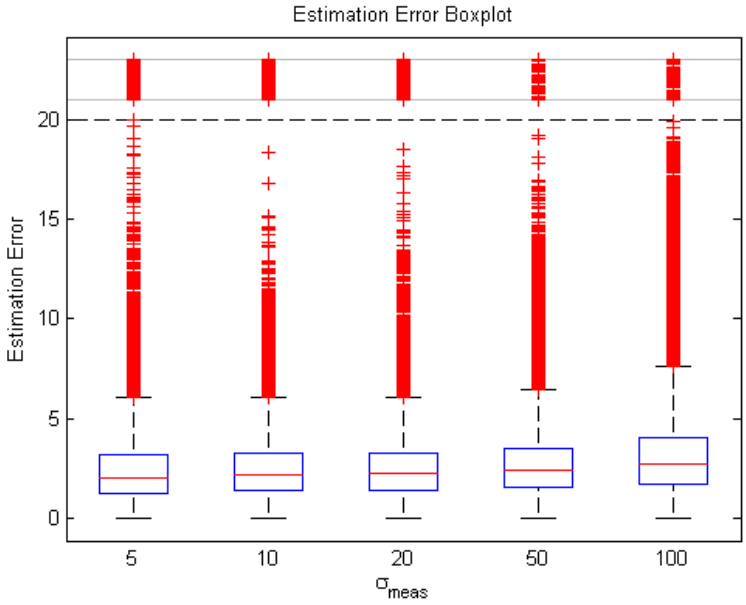


Fig. VI.3. Estimation error vs. σ_{meas}

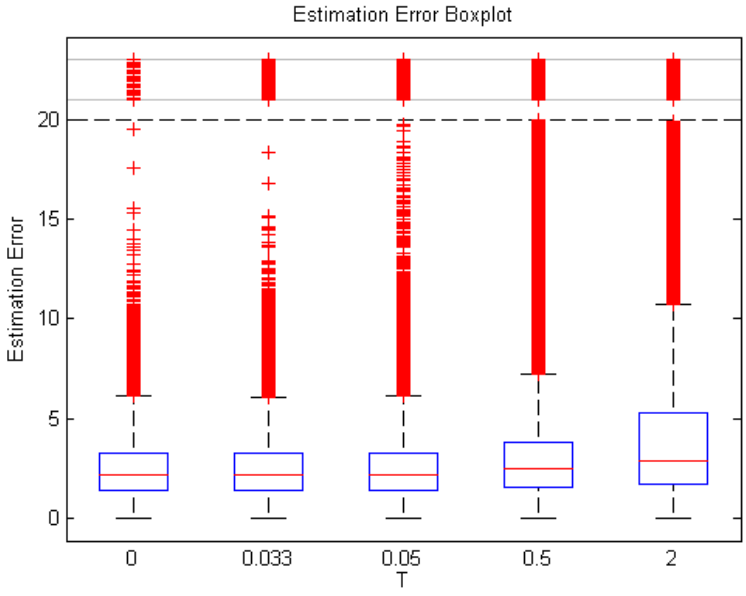


Fig. VI.4. Estimation error vs. sampling time (T)

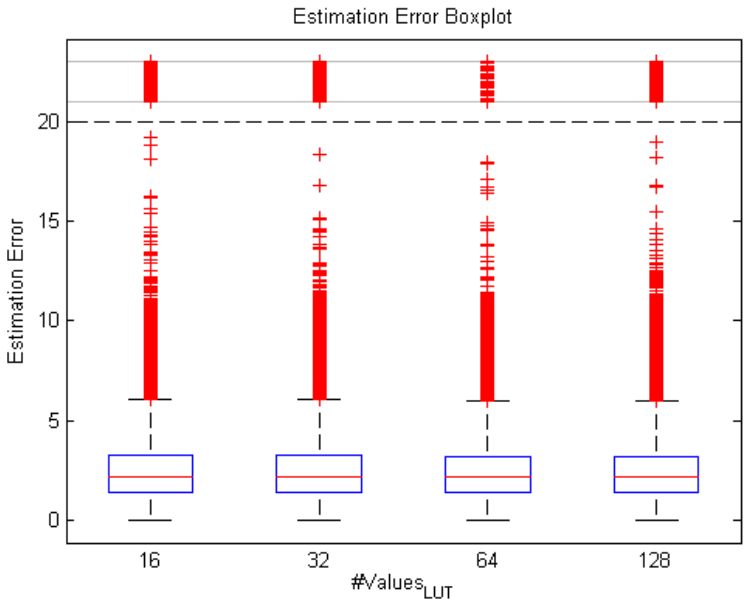


Fig. VI.5. Estimation error vs. number of LUT values

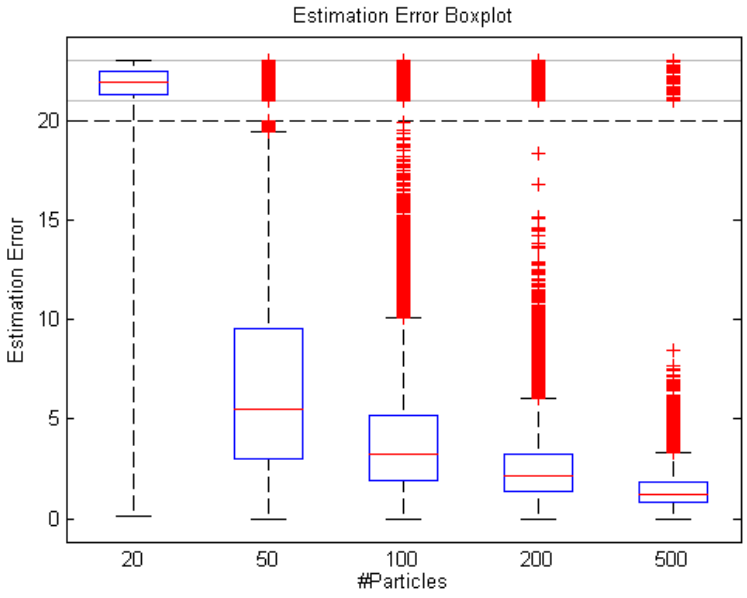


Fig. VI.6. Estimation error vs. number of particles (population size)

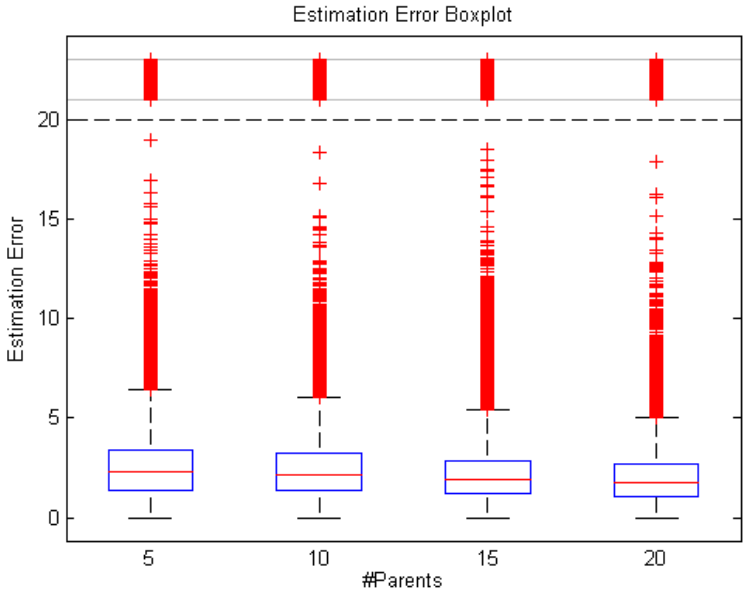


Fig. VI.7. Estimation error vs. number of parents

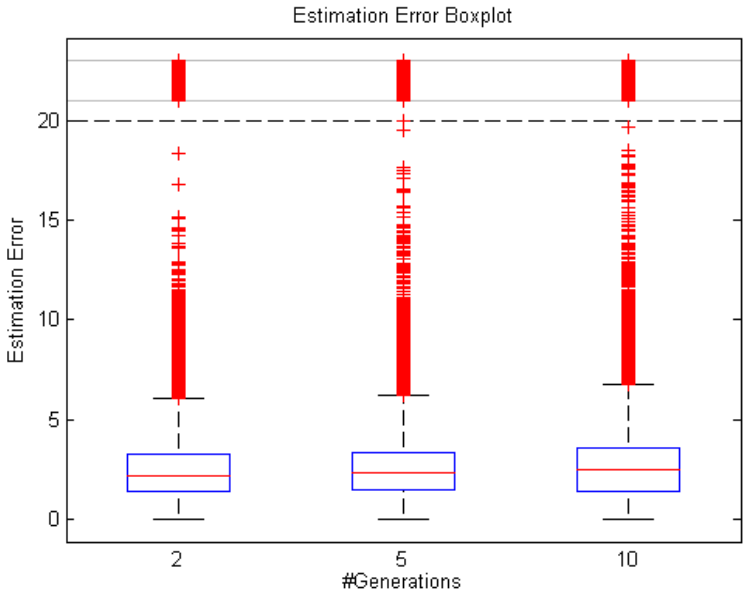


Fig. VI.8. Estimation error vs. number of generations

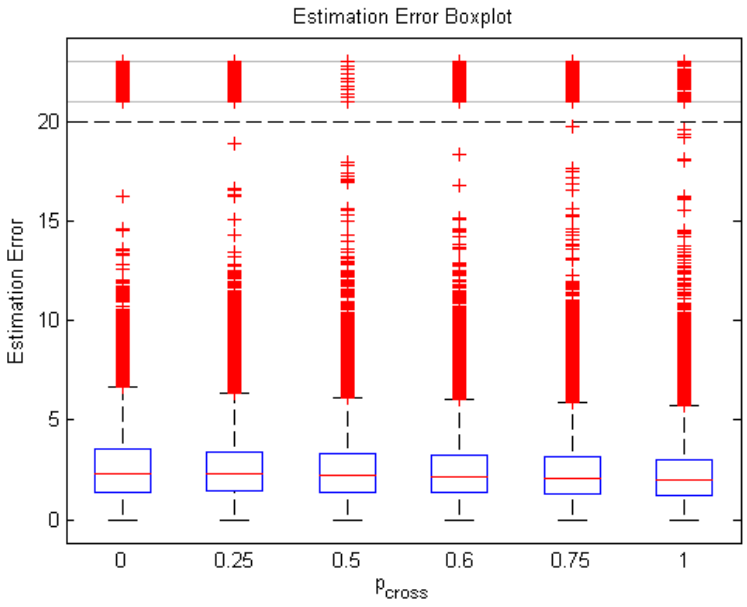


Fig. VI.9. Estimation error vs. crossover probability

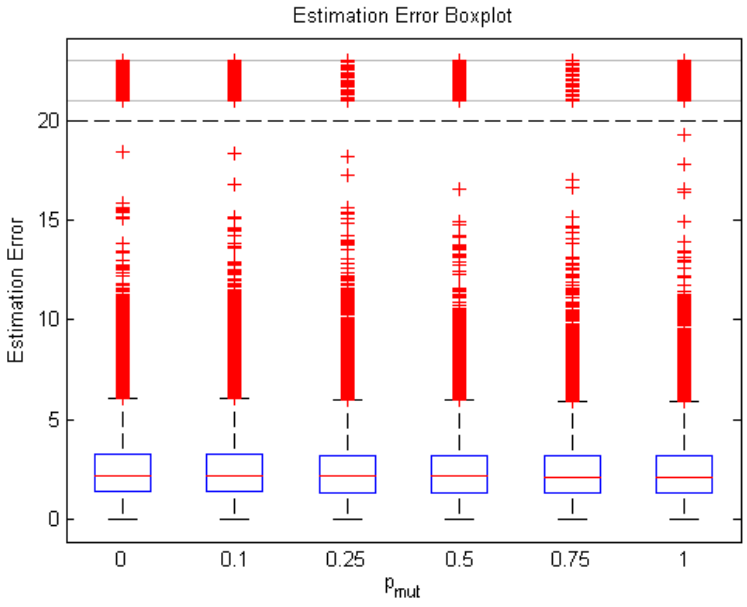


Fig. VI.10. Estimation error vs. mutation probability

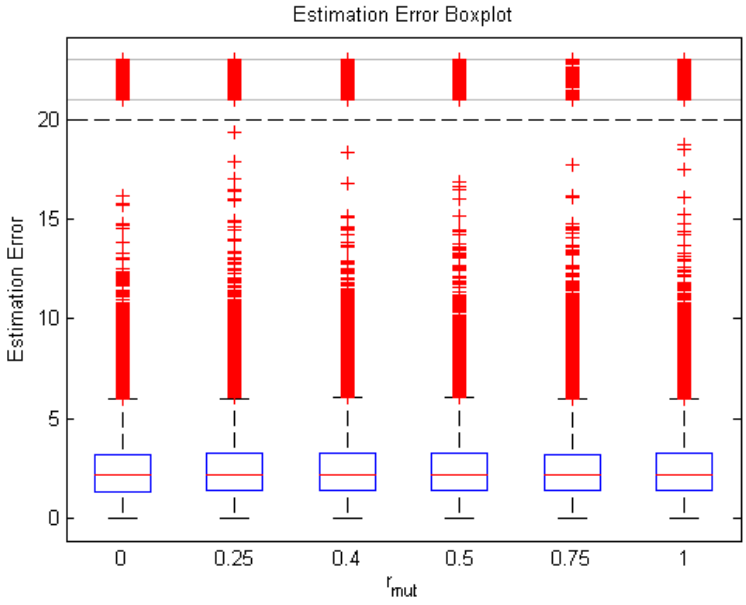


Fig. VI.11. Estimation error vs. mutation ratio

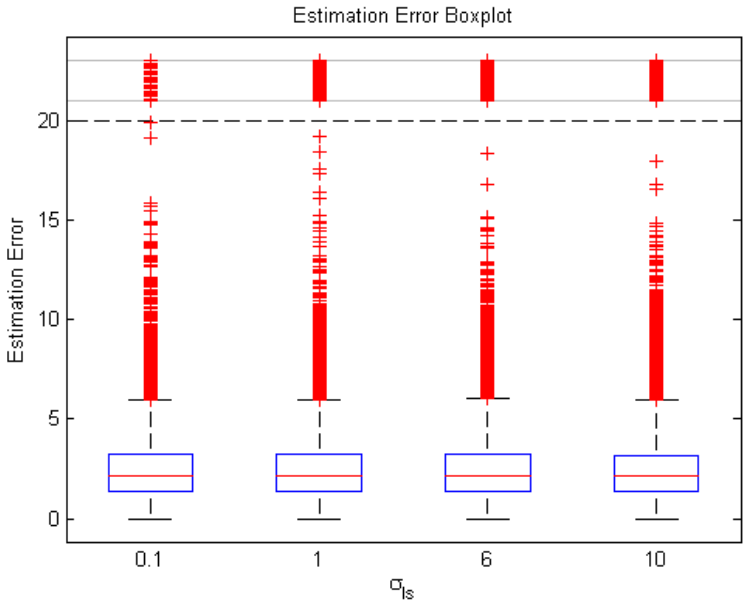


Fig. VI.12. Estimation error vs. σ_{mut}

The reference values that have been used for all the system parameters throughout these tests can be found in Table VI.1.

Parameter	Reference value
σ_{pos}	32
σ_{vel}	0.01
σ_{meas}	10
T	0.033
#LUT values	32
#particles	200
#parents	10
#generations	2
p_{cross}	0.6
p_{mut}	0.1
r_{mut}	0.4
σ_{mut}	6.0

Table VI.1. Reference values for system parameters

Let us now analyze the results that have been obtained. As for the sensitivity analysis, they have been represented as boxplots, using test input patterns of 128k samples.

- σ_{pos} (Fig. VI.1): the standard deviation of the position noise, i.e. the process noise, has a huge impact on system performance. Estimations are more accurate when the position noise is similar to the movement uncertainty. If the position noise is either set to be a small number (e.g. 4) or a large number (e.g. 128), the results that are obtained are not good enough. Taking a closer look at the graph, it is possible to see that the minimum estimation error is located at $\sigma_{pos} = 16$. However, $\sigma_{pos} = 32$ still provides good results, which is the reason why this value has been selected as the reference for this parameter.
- σ_{vel} (Fig. VI.2): the velocity noise has almost no relevant impact on the estimation error. Good estimations are obtained as long as this parameter remains small ($\sigma_{vel} < 1$).
- σ_{meas} (Fig. VI.3): measurement noise has no significant impact on system performance either. Again, this parameter should not be increased over a certain amount.
- Sampling time T (Fig. VI.4): this parameter weights the velocity impact on the estimation. Therefore, the results that have been obtained are very similar to the ones of the sensitivity analysis of σ_{vel}. The velocity is a state variable whose weight should be less than the weight of the other state variables, i.e. the position ones.

- #LUT values (Fig. VI.5): in this particular set of tests, the number of values that are used to build the fitness look-up table seem to be irrelevant. However, it is important to keep in mind that this is mainly due to the fact that the measurement noise is set to be small. If this value is increased, the number of LUT values will have to be increased as well in order not to lose precision.
- #particles (Fig. VI.6): the number of particles turns out to be another important parameter in the Evolutionary Particle Filter. This was already known from the very beginning of the design process, since the more particles there are, the more accurate the posterior distribution is. The reference value has been set to 200, since larger values increase sharply resource consumption rates.
- #parents (Fig. VI.7): the results show that the more parents the evolutionary algorithm uses, the more accurate the results are. However, there is a similar problem as with the population size: the resource consumption is highly increased. Therefore, the selection of the number of parents requires a tradeoff decision between precision and area overhead.
- #generations (Fig. VI.8): the more generations the evolutionary algorithm has, the less accurate the system becomes. This is obvious since it has been stated that having a large number of generations would cause sample impoverishment phenomena, thus reducing system performance.
- p_{cross} (Fig. VI.9), p_{mut} (Fig. VI.10), r_{mut} (Fig. VI.11) and σ_{mut} (Fig. VI.12): these evolutionary algorithm parameters seem to have no impact at all on the system performance. There is no particular reason for this to be this way. Therefore, further analysis have to be carried out, showing the estimation error variation when more than one parameter is changed at a time.

A second set of tests was developed in order to cope with the lack of valid results when modifying only one genetic operation (see last paragraph). The architecture was slightly improved by adding three extra control registers in which the microprocessor writes the crossover probability, the mutation probability, and the mutation ratio. Once these changes had been made, and using the same 128K samples, the system evaluated the estimation performance for all possible combinations of the aforementioned parameters. The results can be seen in Fig. VI.1, Fig. VI.2, Fig. VI.3, Fig. VI.4 and Fig. VI.5. Notice that estimation errors decrease when the probabilities are large. However, crossover seems to have larger impact on the results. Also, notice that mutations increase the estimation error if $r_{mut} > 0.5$. Therefore, random placement mutation, although necessary (see Fig. VI.1, where the error does not follow any pattern), has to be kept at a minimum value, since local search mutation provides better results. The mutation ratio has been set as 0.4 as its reference value in order to comply with this requirement.

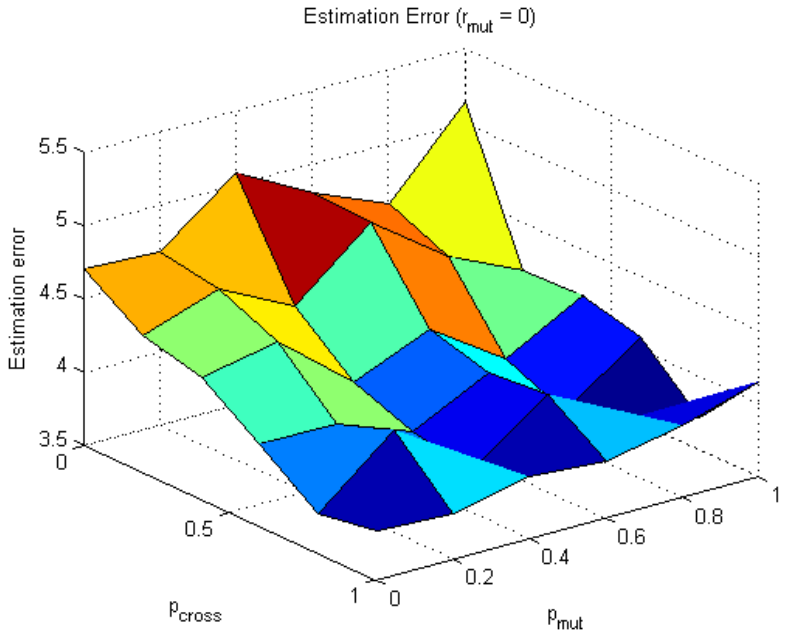


Fig. VI.1. Estimation error with 100 particles and $r_{mut} = 0$

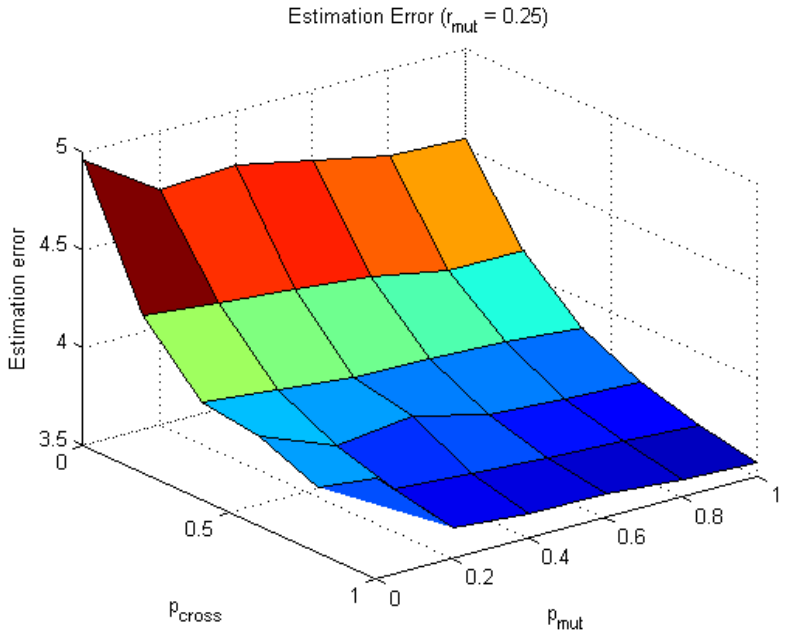


Fig. VI.2. Estimation error with 100 particles and $r_{mut} = 0.25$

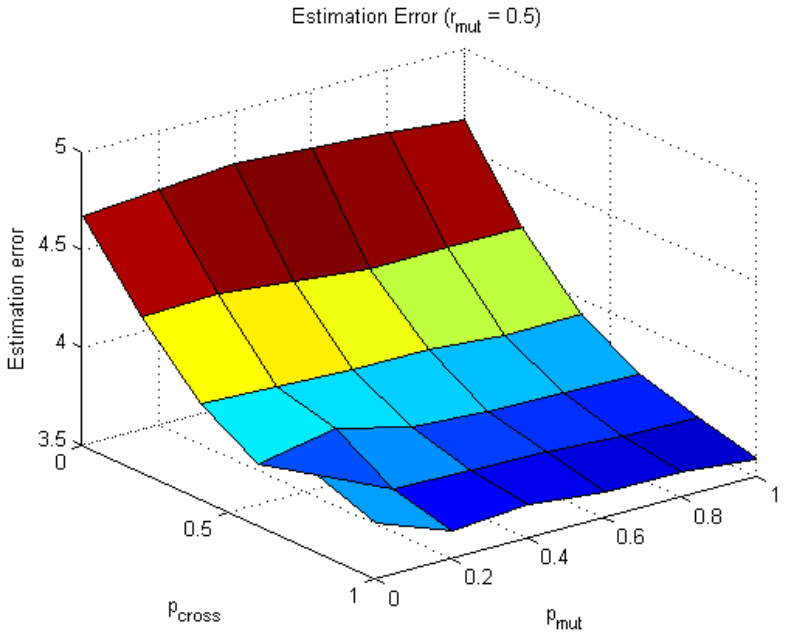


Fig. VI.3. Estimation error with 100 particles and $r_{mut} = 0.5$

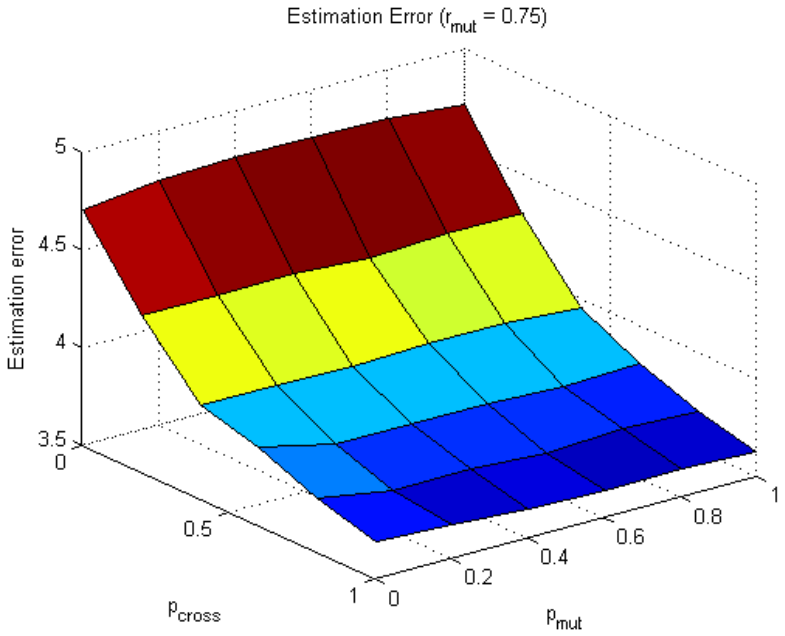


Fig. VI.4. Estimation error with 100 particles and $r_{mut} = 0.75$

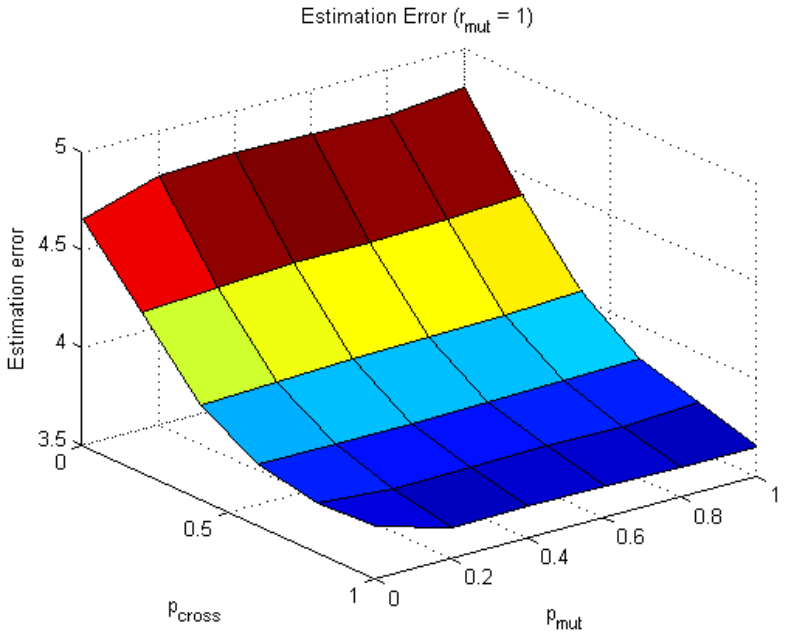


Fig. VI.5. Estimation error with 100 particles and $r_{mut} = 1$

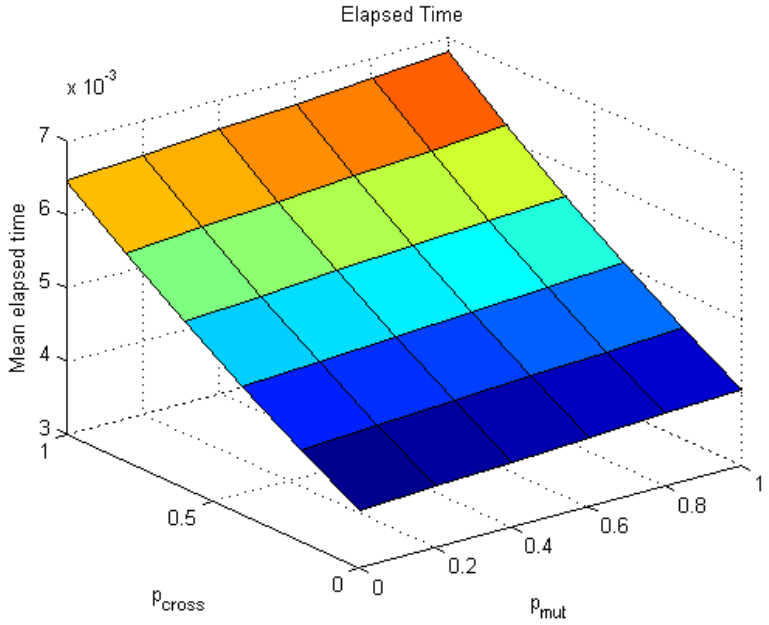


Fig. VI.6. Estimation times: crossover and mutation probability variations

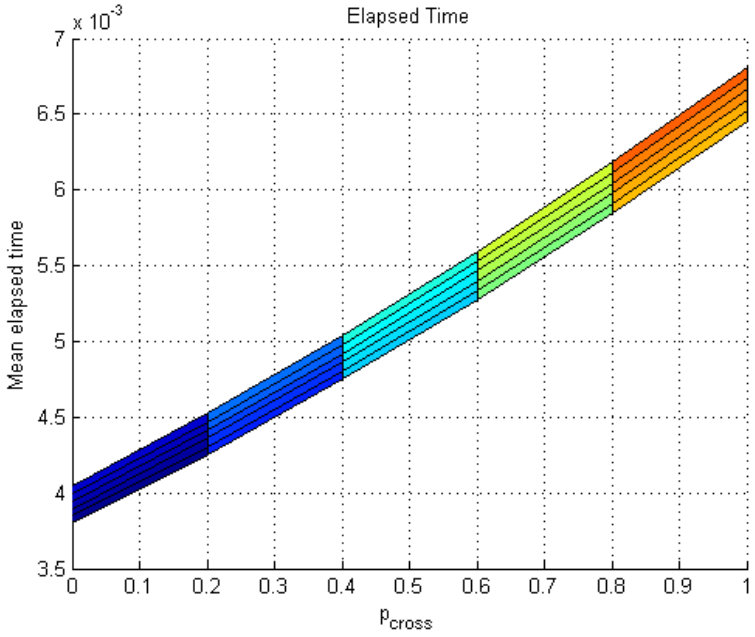


Fig. VI.7. Estimation times: crossover probability variations

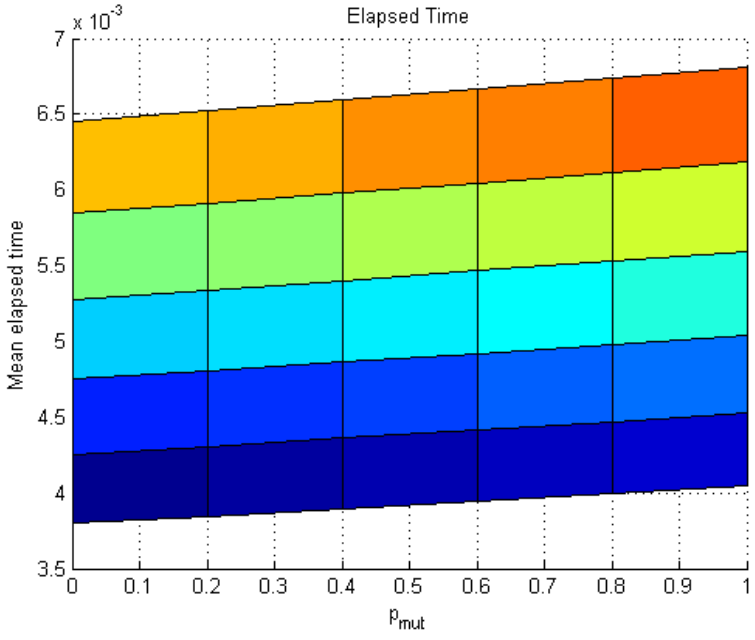


Fig. VI.8. Estimation times: mutation probability variations

Estimation times were also analyzed, as it can be seen in Fig. VI.6. Notice that changes in crossover probabilities (Fig. VI.7) have larger impact on estimation time than changes in mutation probabilities (Fig. VI.8), since crossover operations generate two children, whereas mutation only creates one. In these last three examples, the population size was set to be 200 particles and 10 parents were used to generate the offspring.

All things considered, the Evolutionary Particle Filter can be thought of as a robust system in which parameter variations seem not to affect estimation overall accuracy.

VII. Hardware vs. Software: Comparison

In order to assess the reduction in terms of time that the hardware implementation of the Evolutionary Particle Filter provides, three tests have been carried out:

- HW-EPF: the hardware architecture proposed in this thesis performs the estimation process.
- SW-MB (fixed point): the Evolutionary Particle Filter runs as a software application in a MicroBlaze. All the computations are performed using fixed-point precision.
- SW-MB (floating point): the Evolutionary Particle Filter runs as a software application in a MicroBlaze. All the computations are performed using floating-point variables.

Since this estimation algorithm is intended for embedded systems, only these three alternatives were considered. The results for different population sizes are shown in Table VII.1. Note that the HW-EPF takes less time than the other two approaches.

Implementation	Population size	Estimation time (ms)
HW-EPF	50 particles	0.291 – 1.509
	100 particles	1.01 – 2.829
	200 particles	3.797 – 6.81
SW-MB (fixed point)	50 particles	26.967 – 127.669
	100 particles	70.154 – 210.371
	200 particles	205.77 – 374.475
SW-MB (floating point)	50 particles	37.342 – 189.224
	100 particles	84.266 – 288.387
	200 particles	207.342 – 389.501

Table VII.1. Elapsed time comparison

A graph showing the different estimation times for the HW-EPF can be seen in both Fig. VII.1 and Fig. VII.2.

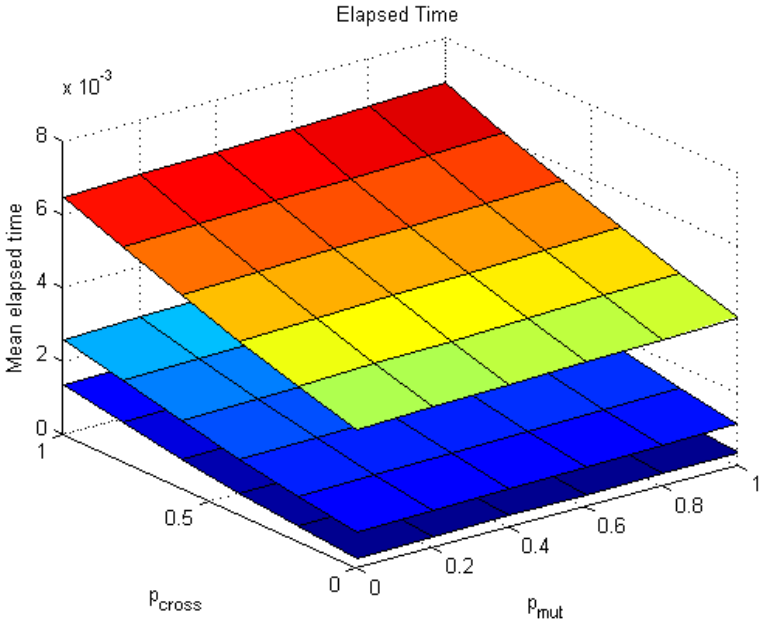


Fig. VII.1. HW-EPF estimation times

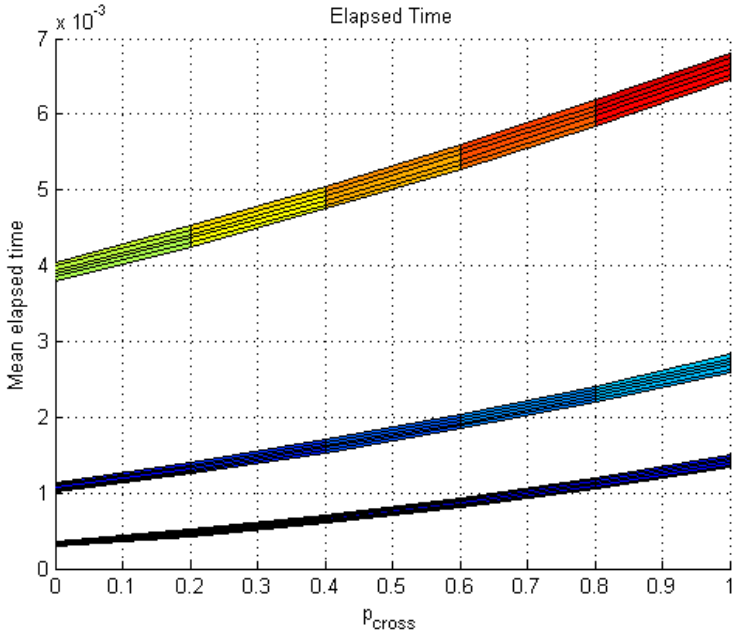


Fig. VII.2. HW-EPF estimation times: comparison

These results have been verified through simulation. In Fig. VII.1, the estimation time when there are no genetic operations (therefore, the elapsed time has its minimum value) is shown. Moreover, in Fig. VII.2, the estimation time when both crossover and mutation are always performed (therefore, the elapsed time has its maximum value) can be seen. Also, notice that the most time-consuming operation in the algorithm is the particle sorting stage. Therefore, the more particles and parents the system has, the more time is required to perform an estimation, as it has been shown in the previous section. For further increasing system performance, and thus reducing estimation times, new sorting algorithms can be implemented.

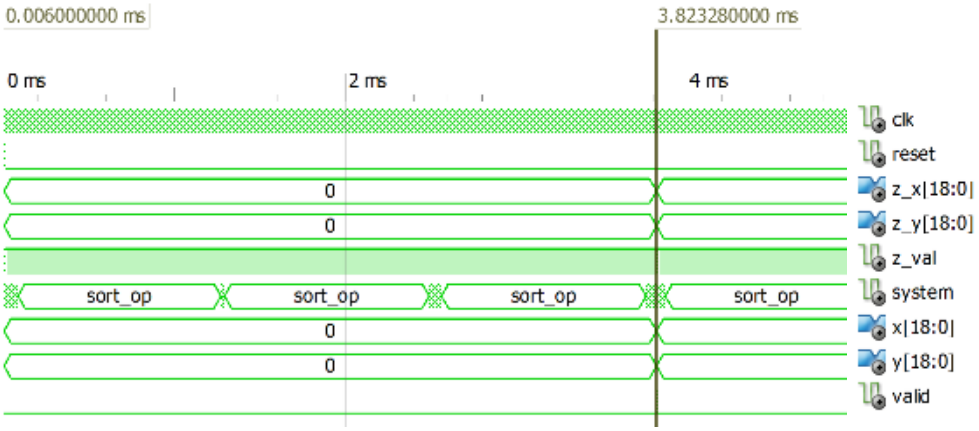


Fig. VII.1. Simulation results: estimation time with $p_{cross} = 0$ and $p_{mut} = 0$

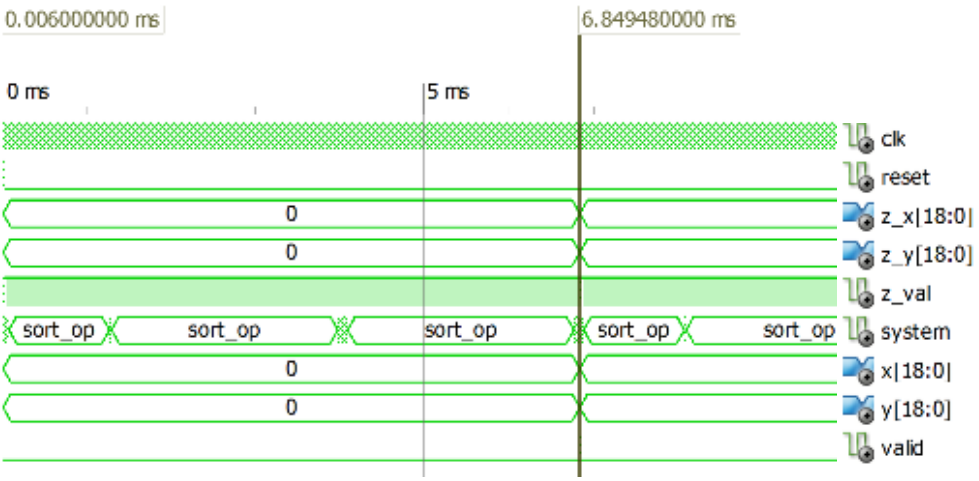


Fig. VII.2. Simulation results: estimation time with $p_{cross} = 1$ and $p_{mut} = 1$

VIII. Conclusions

A novel particle-filtering architecture, the HW-EPF, has been designed. Its performance analysis reveals that it not only provides accurate motion state estimations, but also outperforms other algorithms, e.g. the Bootstrap Filter. Particle-filtering common problems, i.e. particle degeneracy and sample impoverishment, are mitigated with the proposed algorithm, providing both accurate and realistic posterior distributions.

The HW-EPF modular architecture provides flexibility and reconfiguration capabilities to the embedded system in which is used. As a hardware accelerator, it speeds up estimation throughput. This acceleration has been verified when establishing a comparison between the same algorithm with different implementations (only hardware, software with fixed-point precision, and software with floating-point precision) over the same evaluation platform, and comes from the advantages that hardware processing have when dealing with repetitive operations.

The sensitivity analysis shows that those system parameters that increase particle number, e.g. population size, parent size (the more parents are selected, the more offspring is generated), have to remain under certain limits, in order to avoid excessively large implementations, i.e. with large area overhead. Moreover, mutation and crossover probability thresholds have to be selected taking into account the tradeoff between precision and execution time: higher values show, generally speaking, more accurate results but it takes longer to obtain the estimations. In addition, the maximum number of generations has to be small, in order to mitigate sample impoverishment phenomena and reduce estimation times.

All things considered, the HW-EPF has proved to be an outstanding filter, as well as a robust and powerful estimation tool. A proof-of-concept implementation using HIL co-simulation has been made in order to validate system functionality.

IX. Future Work

There are four main future research lines:

- Finish the global embedded system, including a reconfigurable preprocessing stage. With this enhancement, the Evolutionary Particle Filter would be able to track and estimate the position of different objects, as long as their coordinates can be expressed as 2D vectors.
- Fully exploit FPGA parallel-processing capabilities with concurrent execution of the designed hardware modules, which has not been fulfilled within this Master thesis.

- Increase maximum operating frequency introducing minor changes in the architecture, such as pipelined modules, or a new particle sorting algorithm, since the bubble-sorting algorithm is a bottleneck.
- Implement a reconfigurable distributed particle filtering strategy. If the tracking is lost, an additional particle filter is dynamically reconfigured in the FPGA fabric, evolving in parallel with the main population. Migration operations are expected to improve the quality of the estimations. Once a good result is obtained, the additional particle filter is removed from the FPGA.

Bibliography

1. Doucet, Arnaud; Freitas, Nando de; Gordon, Neil (Eds.). 2001. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag
2. Cappé, Olivier; Moulines, Eric; Ryden, Tobias. 2005. *Inference in Hidden Markov Models.* Springer-Verlag
3. Liu, Jun S.. 2001. *Monte Carlo Strategies in Scientific Computing.* Springer-Verlag
4. Eiben, Agoston E.; Smith, J.E.. 2003. *Introduction to Evolutionary Computing*. Springer-Verlag
5. Miller, Julian F. (Ed.). 2011. *Cartesian Genetic Programming.* Springer-Verlag