A Fully Parallel Particle Filter Architecture for FPGAs

Fynn Schwiegelshohn(✉), Eugen Ossovski, and Michael Hübner

Embedded Systems for Information Technology, Ruhr-Universität-Bochum, Bochum, Nord-Rhein-Westfalen, Germany
{fynn.schwiegelshohn,eugen.ossovski,michael.huebner}@rub.de

Abstract. The particle filter is a nonparametric filter which approximates the posterior system state through a finite number of state samples i.e. particles drawn from a probability distribution. It consists of three steps which are motion update, sensor update and resampling. The first two steps are easily parallelized since the calculations do not depend on other particles. The resampling step however requires all particles to determine the particle set for the next iteration of the particle filter. In this paper, we introduce a novel FPGA optimized resampling (FO-resampling) approach to solve the parallelization problem of the resampling step by introducing virtual particles. Compared to multinomial resampling, FO-resampling achieves similar results with the added benefit of being able to completely parallelize all the steps of the particle filter. Additional to evaluating our approach with simulations, we implement a particle filter with FO-resampling on an FPGA.

Keywords: Particle filter · FPGA · Resampling · Parallelization · Robotics · Localization · Sensor update · Motion update · FPGA optimized resampling

1 Introduction

The particle filter is an alternative to the common and well known Kalman filters which use Gaussian techniques for state estimation. They have been successfully employed in a wide variety of robotic application scenarios [9]. Since they do not make strong parametric assumptions on the posterior density, they are able to represent complex multi-modal beliefs. Due to this fact, particle filters are often used in robotics when a robot might face data association problems which result into several different system hypothesis. The accuracy of the state estimation depends on the amount of samples that are drawn from the probability distribution by the particle filter. An infinite amount of samples lets the particle filter converge to the correct posterior state. Therefore, the particle filter usually operates with a large number of particles. In [8], the authors state that autonomous robots can benefit from FPGAs as processing platforms due to their ability to execute tasks in parallel and to fully utilize elastic algorithms. In order

K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 91–102, 2015.
DOI: 10.1007/978-3-319-16214-0_8

to achieve the best performance from a given processing platform, the algorithm has to be adapted accordingly. If the robot is able to execute the particle filter algorithm in parallel, the localization accuracy would only influence the particle filters processing time slightly compared to a sequential implementation.

A lot of research has been conducted on the parallelization of particle filters [3,4,6,7]. Gong et al. introduce a parallel resampling method for shared memory architectures based on systematic resampling [4]. With the shared memory approach, they are able to eliminate the dependency between the left and right boundary when choosing a particle from the particle set and thus are able to parallelize the resampling step. Through the use of shared memory, this solution is not easily implemented on FPGAs but is useful for graphical processing units (GPU). Choppala et al. propose to use a random network as fixed resampling unit for the particle filter [3]. Here, each particle will interact with a fixed set of other particles provided by the network. The resampler then samples one particle form the respective set either deterministically or stochastically. Miao et al. developed a parallel particle filter architecture for FPGAs which uses independent Metropolis-Hastings sampling to increase performance on the root mean square error value [6]. They successfully achieved high speed and accurate performance with their implementation. However, only several processing elements responsible for 250 particles each are used thus no full parallelization is implemented. Mountney et al. present another parallel particle filter architecture for neural signal processing [7]. Here, the time needed to execute the particle filter is independent of the number of particles. This is not the case in the other particle filter architectures since they use processing elements to calculate several particles. We aim to let the processing time be independent from the number of particles.

Therefore, this paper proposes a novel, fully parallel particle filter architecture. It's main contribution is a new resampling scheme to fully parallelize each step of the particle filter for each particle. This new resampling scheme is based on the idea of Gibbs sampling and does not require the complete particle set to execute the resampling step. In order to adapt each state hypothesis to the current sensor data, virtual particles are generated that randomly move through the state space. If one virtual particle achieves a higher importance factor than the actual particle, the real particle assumes the state hypothesis of the virtual particle. The performance of this new resampling approach has been evaluated against the multinomial resampling scheme through MATLAB simulations. The implementation of the particle filter is done on a Zynq7000 System on Chip where further performance measurements and evaluations are conducted. There, we analyze the frequency at which the particle filter can be driven, the amount of real particles that can be utilized when using the complete chip area, and the performance of the particle filter on hardware compared to the MATLAB implementation.

In the following section, we will introduce each component and in our architecture before explaining our implementation in Section 3. Section 4 presents our results with several test cases for the particle filter with FO-resampling. Finally, we will conclude our work in Section 5.

2 Architecture Design

In order to be able to parallelize the particle filter, it is necessary to examine the algorithm and detect the parallelizable parts. Code 1 shows the basic particle filter algorithm and its functionality.

Code 1. Algorithm of the Particle Filter

```
1:  particle_filter(χ_{t-1}, u_t, z_t)
2:  {
3:        χ̄_t = χ_t = ∅
4:        for m = 1 to M do
5:              sample x_t^[m] ~ p(x_t | u_t, x_{t-1}^[m])
6:              w_t^[m] = p(z_t | x_t^[m])
7:              χ̄_t = χ̄_t + ⟨x_t^[m], w_t^[m]⟩
8:       endfor
9:       for i = 1 to M do
10:          draw i with probability
11:          add x_t^[i] to χ_t
12:       endfor
13:       return χ_t
14:  }
```

χ_{t-1} resembles the set of particles from the particle filter iteration, u_t describes the most recent control input, and z_t stands for the current sensor measurement. In Line 5, the algorithm generates a first state hypothesis for the current particle based on the particle $x_{t-1}^{[m]}$ and the control input u_t. In robotic localization, this step is also known as the motion update step. In Line 6, a weight is then calculated for the sampled particle from Line 5 with the help of the sensor measurement z_t. The function with which the weight is calculated can be an environment model or a simple Gaussian distribution function. Calculation of the importance factor is generally called sensor update step in robotics. Line 8 simply saves all sampled particles with their respective weights in a temporary set $\bar{\chi}_t$. The last step, which involves the Lines 9 to 12 of Code 1, is called the resampling step and is the most important one of the particle filter. Here, the algorithm draws M particles from the temporary set $\bar{\chi}_t$. The probability for each particle to be selected is defined by the respective weight w_t^m which has been calculated in Line 6. Therefore, each particle can be selected multiple times, thus changing the composition of the final set χ_t compared to the temporary set $\bar{\chi}_t$. After the resampling step, all particles have uniform weights until the sensor update step is executed again.

2.1 FPGA Optimized Resampling

Instead of replacing particles with a small importance factor through particles with a high one, FPGA optimized resampling (FO-resampling) requires the

particles to increase their weight by themselves. This approach eliminates the possible interrelationship between each particle from the former particle set χ_{t-1} to the current particle set χ_t. Code 2 shows the FO-resampling algorithm in pseudo code with N being the number of particles.

Code 2. Algorithm of FPGA optimized resampling

```
1: foreach particle i ∈ {1,...,N}
2:   fo_resampling(x_i, w_i)
3:   {
4:       for n = 1 to B do
5:           sample r ~ U(−1, 1)
6:           x̂_{i,n} = x_i + σ_{x_i} · r
7:           ŵ_{i,n} = p(z_t | x̂_{i,n})
8:           if ŵ_{i,n} > w_i then
9:               x_i = x̂_{i,n}
10:              w_i = ŵ_{i,n}
11:          endif
12:      endfor
13: return x_i
14: }
```

In FO-resampling, every particle has a constant number of B opportunities to increase its weight w_i, see Line 4. In every opportunity, a virtual particle $\hat{x}_{i,n}$ will be randomly generated around the actual particle x_i. This is shown in Line 5 and 6. First we draw a random number r from a uniform distribution with the boundaries $[-1, 1]$. This random number is multiplied with the standard deviation σ_{x_i} and then added to the current particle state x_i. The standard deviation σ_{x_i} is used to define the spread of $\hat{x}_{i,n}$ around x_i. If the initial importance factor is very low, a higher standard deviation can be chosen in order to potentially reach regions where higher weights can be achieved. If the initial weight is already very high, a smaller value for σ_{x_i} should be chosen since the position with the highest probability is close by and only minor corrections are required in order to reach the highest possible weight. Consequently, if the importance factor $\hat{w}_{i,n}$ of the virtual particle achieves a higher value than the corresponding factor w_i of the real particle, x_i will be replaced with $\hat{x}_{i,n}$. If $\hat{w}_{i,n} < w_i$ is true, the real particle is not replaced and the virtual particle from the next iteration will be compared to the real particle again. In Code 2, Line 8 to 11 show the replacement of the real with the virtual particle parameters. After B iterations, all particles x_i are incorporated into the posterior estimation.

In this resampling method, no operations are used which require a relation between all particles x_i. This enables the design of a complete parallel particle filter architecture and is thus optimized for parallel architectures such as FPGAs. One of the main benefits from this architecture is that each component is reused for creating and evaluating the virtual particles for each particle. Generally, the motion and sensor models of particle filters are com-

putationally the most complex components. This leads to a high processing time when these components are reused on a general purpose processor. When implemented on an FPGA, these components consume a lot of chip area when implemented as hardwired logic circuits but do not have a high processing time. Therefore, we do not add additional components to the particle filter for resampling but use the components which are already implemented. This leads to an efficient chip area usage on FPGAs and is the reason why we call this resampling technique FO-resampling.

An argument against our novel resampling technique is that FO-resampling is biased since it does not eliminate particles with low weights through replacement and the iteration value B is finite. We will empirically show in Section 4 that an estimation of the iteration value B can be made in order to achieve satisfactory weight quality for each particle.

3 Implementation

As mentioned in the previous section, the particle filter consists of the three components motion model, sensor model, and FO-resampler. We will discuss the implementation of each component in this section. The particle filter receives as input control data u_t, sensor data z_t, and random numbers from a random number generator (RNG). The particle's state hypothesis is represented by the parameter triple $x_i = (x, y, \theta)$, with x and y being the coordinates in a 2D space and θ being the orientation of the object which is to be localized, in this use case the robot.

All particle filter components have to be connected to each other in order to create the parallel particle filter architecture. Figure 1 shows the particle filter's architecture. Here, the parallel calculation of each particle can be clearly seen as well as the reuse of the motion and sensor model components. The RNG, which is required by the motion model, is implemented as a linear feedback shift register. The RNG generates normally distributed random numbers. Since several random number have to be available at once, we decided to let the RNG run continuously and fill a shift register with it's random numbers. When the motion model components require the random numbers, three of them for each system state parameter will be sent to each component. This approach enables us to use only one RNG for the complete particle filter and not one RNG for each particle. After the resampling step is completed, the particle set has to be interpreted in order to gain a position estimation. This is done through calculation of the mean for each parameter of all particles. This approach is only feasible when no multi-modality is present in the system but is easily adaptable to support multi-modality when required. After calculating the mean, the position estimation is ready to be sent to an output for further processing.

The control logic, for enabling each component is not present in Figure 1. It's state machine is designed as a Moore machine and is presented in Figure 2. Here, the value "calc_en" stands for the enable signal for the motion and sensor model and the value "res_en" stands for the enable signal for the resampling component. The state machine is in the state "undefined" directly after initialization,

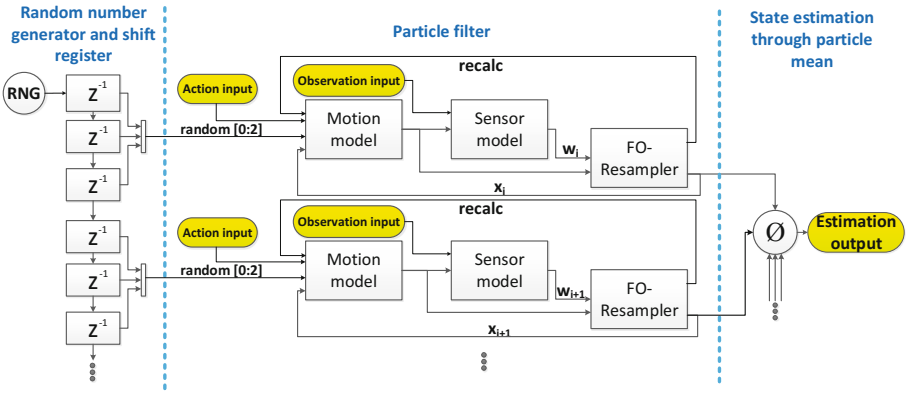

Fig. 1. Complete particle filter architecture

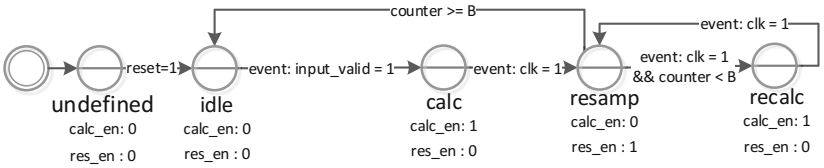

Fig. 2. The control logic to enable the respective component for each particle filter state

as undefined signals may be present on the control logic inputs. The particle filter is brought to a defined state through the "reset" signal. This signal also lets the state machine transition from the "undefined" state to the "idle" state. In this state, all components of the particle filter are locked and wait for valid input signals. The "input_valid" signal indicates that valid inputs are available for the particle filter and that the motion and sensor model can start processing the inputs. This is shown in the state machine by the transition from the "idle" state to the "calc" state through the "input_valid" signal. The results of both components are available after one clock cycle and the state machine can transition at the clocks rising edge "clk=1" to the state "resamp", where evaluation of the particle state hypothesis through FO-resampling takes place. The state machine will alternate between the "recalc" and "resamp" state for the creation and evaluation of the B virtual particles. After B iterations of both states, the final particle state hypothesis has been calculated and will be sent to the output of the FO-resampling component. The state machine will then transition from the "resamp" state into the "idle" state, where it will wait until the next valid input is available for processing.

4 Evaluation and Results

Since FO-resampling is a novel resampling approach, it's accuracy performance should be compared to other established resampling methods. Traditional resampling methods only have the number of particles to increase accuracy. In FO-resampling however, both the number of real and virtual particles can be adjusted in order to reach the desired accuracy. For this purpose, we investigate the accuracy performance of FO-resampling at first in simulation, see Section 4.1, before evaluating the implementation, see Section 4.2.

4.1 Simulation of FO-resampling

We employ a well established model for evaluating resampling methods [1,2,6] with the following functions

$$x_{k+1} = \frac{x_k}{2} + \frac{25 \times x_k}{1 + x_k^2} + 8cos(1.2k) + v_k \tag{1}$$

$$z_k = \frac{x_k^2}{20} + n_k. \tag{2}$$

x_k describes the current actual state the device under test (i.e the robot) is in. z_k are the sensor readings which the particle filter receives. v_k and n_k are zero mean Gaussian random variables with the variances $\sigma_{v_k}^2 = 10$ and $\sigma_{n_k}^2 = 1$. This system model's equation (1) has a high nonlinearity which makes localization very challenging. We compare FO-resampling with multinomial resampling on this system model. The number of particles in both particle filters is $N = 100$ and additionally, the particle filter with FO-resampling is executed with $B = 10$ virtual particle iterations. The variance of the motion update is $\sigma_{mot}^2 = 1$ and of the sensor model $\sigma_z^2 = 0.05$ The results are depicted in Figure 3.

When comparing both estimations with the true system states, it is apparent that both resampling method perform satisfactory in accuracy and robustness. Even when estimations prove to be very wrong, both resampling methods manage to recover and regain a correct estimation for the system state. Since the model for the sensor data induces a dual-modality to the system through x_k^2, the performance of both particle filters can be increased when a more sophisticated clustering scheme is used to determine the posterior system state. In order to quantify the performance of the particle filter, we use the Root Mean Square Error (RMSE) value $RMSE = \sqrt{\frac{1}{k}\sum_{t=0}^{k}(x_t - \hat{x}_t)^2}$. The RMSE is traditional measure of performance for Bayes filters. The RMSE value was determined with different numbers of particles N for multinomial resampling and with different numbers of particles N and iterations B for FO-resampling. The system model described in the equations (1) and (2) was used for each simulation. The parameters σ_{mot}^2, σ_z^2, $\sigma_{v_k}^2$, and $\sigma_{n_k}^2$ have not been changed. Table 1 shows the RMSE results for 1 to 10000 particles.

A poor performance with a small number of particles is expected. Even with a small number of 10 particles, a significant improvement in the RMSE value can

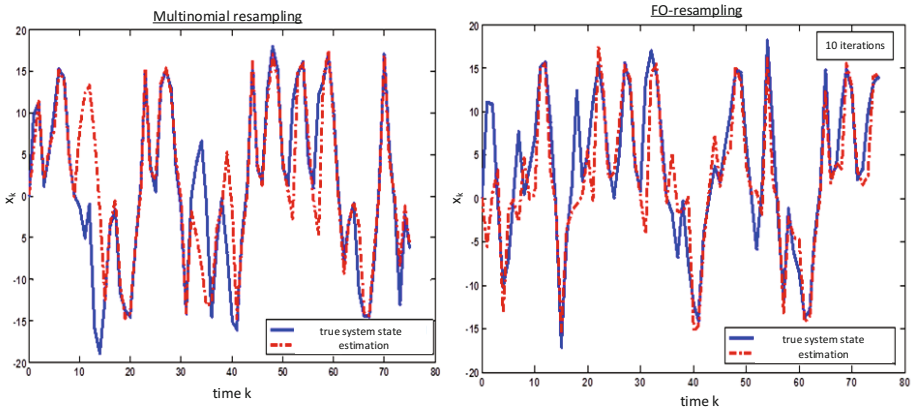

Fig. 3. Comparison between multinomial resampling and FO-resampling

Table 1. RMSE values for multinomial resampling dependent on the number of particles

Number of particles N	1	5	10	50	100	1000	10000
RMSE	0.220	0.159	0.062	0.065	**0.044**	**0.049**	**0.045**

be seen. With a particle number above 100, we see the RMSE value converging towards σ_z without any indication for further improvement. Table 2 shows the RMSE values of FO-resampling dependent on the particle number N and the number of virtual particles B.

Table 2. RMSE values for FO-resampling dependent on the number of particles and the number of iterations

RMSE		**Number of particles** N					
	/	1	5	10	50	100	1000
Number of iterations B	1	0.381	0.216	0.206	0.292	0.244	0.267
	5	0.069	0.148	0.140	0.163	0.151	0.152
	10	0.097	0.077	0.071	0.114	0.104	0.092
	50	**0.031**	**0.037**	**0.043**	**0.046**	**0.036**	**0.039**
	100	0.042	0.033	0.041	0.044	0.048	0.030

If FO-resampling is compared to multinomial resampling solely based on the particle number N (B=1), multinomial resampling is far superior. This is due to the fact that multinomial builds the final particle set out of the particles with the highest weights and therefore is able to converge faster to the true system state. However, as the number of virtual particles increases, we can see that even with a small number of particles a similar RMSE performance is achieved. With

$B = 50$ and $N = 10$, the RMSE value is in the same region as with multinomial resampling. Furthermore, we can see that at certain values for N and B no significant improvement regarding the RMSE value is made. This shows that FO-resampling converges towards a final estimation even with small values for N and B. This is especially attractive for parallel hardware implementations, as the N defines the chip area usage of the particle filter.

4.2 Performance of FO-resampling on Real Hardware

We implemented the particle filter with FO-resampling on the programmable logic of a Zynq7020 [5]. In order to send data to the particle filter, the ARM Cortex-A9 dual core processor has to communicate with the programmable logic. This is done with an AXI-Lite bus. The ARM processor executes a program which sends simulated sensor and motion data to the particle filter. When the position estimation is complete, the particle filter will send its result to the processor. In order to determine how many particles we are able to use for our test case, we synthesize our implementation with different number of particles. Table 3 shows the resource requirements for $N = 1$, $N = 10$, and $N = 14$.

Table 3. Resource usage of the particle filter with $N = 1$, $N = 10$, and $N = 14$ particles

	$N = 1$ **particle**		$N = 10$ **particles**		$N = 14$ **particles**		
Resources	Used	in %	Used	in %	Used	in %	**Available**
Slice LUTs	4079	7.66	28860	54.24	52412	98.51	**53200**
Slice Registers	1549	1.45	4636	4.35	5934	5.57	**106400**
DSPs	22	10	220	100	214	97.27	**220**

It can be seen that a maximum of $N = 14$ particles is possible on a FPGA from the Artix-7 family. With 14 particles, the slice lookup tables (LUT) have almost been fully utilized with 98.51%. The particle filter does not require that much slice registers, but fully utilizes the available DSPs. The DSPs are used for calculating the estimations in the motion and sensor model as well as in the RNG. What is interesting to point out is, that all of the DSPs are already used at $N = 10$, but at $N = 14$ only 97.27% are utilized. This is due to the synthesizing process, which remaps some components to LUTs when the preferred component is not available.

Another important attribute of the hardware implementation of the particle filter, is the operating frequency that can be achieved. We therefore simulated the complete particle filter after synthesis to analyze the number of clock cycles needed to perform one position estimation. Figure 4 shows a snippet of the cycle accurate simulation. Here, the signals "clk" for the clock, "action[0:2]" for the control data input, "observ[0:2]" for the measurement, "x_estimate[0:2]" for the particle state estimation, "random" for the random number generation, and

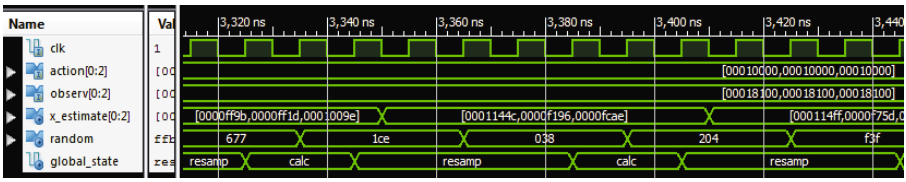

Fig. 4. Excerpt of the implemented particle filter simulation

"global_state" for the control logic state machine are depicted. The calculation of a new random variable takes three clock cycles. This random variable is then written to the shift register which updates all of the other shift register's contents. The process of calculating a new particle state estimation "x_estimate[0:2]" *calc* state requires both control logic states *resamp* and *calc*. *calc* can only be executed after the random variable shift register is updated, as it requires unbiased random variables. Therefore, one iteration of *resamp* and *calc* determines the time for calculating one iteration of "x_estimate[0:2]". As can be seen in Figure 4, this takes six clock cycles. We can determine the processing time with the equation$t_{pf} = 6 \times B \times \frac{1}{f_{max}}$, with B being the number of iterations for FO-resampling and f_{max} being the maximum operating frequency of the synthesized design on the Zynq7020 for $N = 10$ particles. Based on the information from the Synthesis tool, we are able to operate the particle filter at a clock frequency of $f_{max} = 150MHz$. With $B = 100$, this leads to a processing time of $t_{pf} = 6 \times 100 \times \frac{1}{150MHz} = 3.996\mu s$ for one final position estimation by the particle filter.

While the maximum number of particles seems rather small,we can still achieve a comparable performance to multinomial sampling with $N = 1000$ particles when we execute our particle filter with $N = 10$ particles and $B = 100$ virtual particles. These are our architecture parameters for the use case validation. First we evaluate our implementation with five very simple test cases. The initial position for all particles is $x_{init} = (0, 0, 0)$, with the parameters being the x-, the y-coordinate, and the rotation θ respectively. In every test case, the particle filter receives the motion control input $u = (1, 1, 1)$. The sensor measurement z is different with each test case in order to analyze the performance of the implemented particle filter, when the initial position estimation is increasingly inaccurate. Here, we assume that the measurement z does not suffer from severe interferences and closely resembles the actual system state. The results of these test cases are presented in Table 4.

The results from x_{est} shows that particle filter can reach an accurate state estimation even when the measurement z is far off compared to the control input u. In our final use case, we assume the particle filter is performing the localization step of a robot. The robot follows a trajectory which resembles a hexagon. After each time step Δt, the robot performs localization based on it's odometric model with the control data u_t for respective time step. The particle filter receives the actual system state overlayed with white Gaussian noise with

Table 4. Test cases to determine the performance of the particle filter with increasing inaccurate initial state estimations

Initial state	**Control input** u	**Measurement** z	**Estimated state** x_{est}
$(0, 0, 0)$	$(1, 1, 1)$	$(1, 1, 1)$	$(1.0190, 1.0281, 0.9896)$
$(0, 0, 0)$	$(1, 1, 1)$	$(1.35, 1.35, 1.35)$	$(1.3784, 1.3926, 1.3329)$
$(0, 0, 0)$	$(1, 1, 1)$	$(1.5, 1.5, 1.5)$	$(1.4502, 1.4829, 1.5159)$
$(0, 0, 0)$	$(1, 1, 1)$	$(1.75, 1.75, 1.75)$	$(1.7451, 1.7633, 1.7548)$
$(0, 0, 0)$	$(1, 1, 1)$	$(2, 2, 2)$	$(1.8904, 1.8987, 1.8320)$

a variance of $\sigma^2 = 1$. The following Figure 5 shows the localization results of the particle filter. The starting point of the robot is the position $(0, 0)$. It then follows

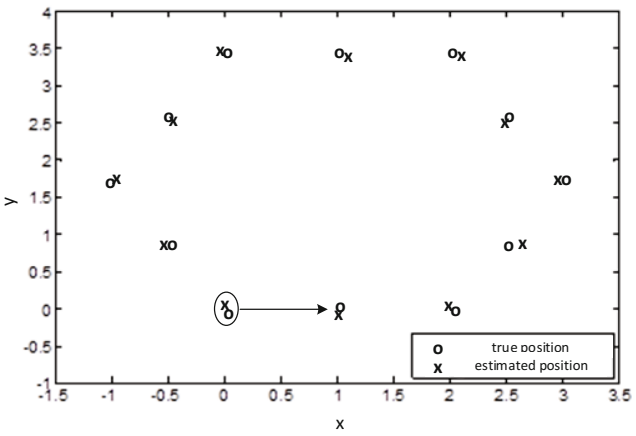

Fig. 5. State estimation in the robot use case

a hexagon form counterclockwise until it is at the starting point again. It can be clearly seen that the particle filter is able to follow the robot's trajectory very closely and that the estimated positions do not have a large deviation from the true position. This can also be seen by the RMSE value of 0.0469 of this use case. The particle filter with FO-resampling managed to generate accurate estimate in several different scenarios. Of course more tests have to be conducted with different parameters, but for an initial evaluation, the particle filter performed satisfactory.

5 Conclusion

In this paper, we introduced a novel resampling scheme, FO-resampling, for particle filters. Normally, the resampling step cannot be fully parallelized, since the next particle set is created out of the current particle set based on each particles

importance factor. FO-resampling eliminates this bottleneck by treating each particle individually. In order to gain a better importance factor without considering every other particle in the set, we FO-resampling introduced the concept of virtual particles which are spread around the initial particle estimation. The importance factor of these virtual particles is compared to the importance factor of the real particle and replaced if the virtual particle's weight is higher. FO-resampling shows promising results in terms of accuracy and stability for the test cases we conducted. However, more use cases have to be implemented and different motion and sensor models have to be implemented in order to fully evaluate this new resampling scheme. This will be done in our ongoing research. Furhtermore, we aim to implement this architecture on larger FPGAs such as Virtex-7 or a rapid prototyping machine like Chipit, in order to better evaluate the implemented particle filter's performance with a larger number of particles.

References

1. Arulampalam, M., Maskell, S., Gordon, N., Clapp, T.: A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. IEEE Transactions on Signal Processing **50**(2), 174–188 (2002)
2. Carlin, B.P., Polson, N.G., Stoffer, D.S.: A monte carlo approach to nonnormal and nonlinear state-space modeling. Journal of the American Statistical Association **87**(418), 493–500 (1992)
3. Choppala, P., Teal, P., Frean, M.: Particle filter parallelisation using random network based resampling. In: 2014 17th International Conference on Information Fusion (FUSION), pp. 1–8 (July 2014)
4. Gong, P., Basciftci, Y., Ozguner, F.: A parallel resampling algorithm for particle filtering on shared-memory architectures. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 1477–1483 (May 2012)
5. Inc., X.: Zynq 7000 all programmable soc. Tech. rep.. (http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html)
6. Miao, L., Zhang, J., Chakrabarti, C., Papandreou-Suppappola, A.: A new parallel implementation for particle filters and its application to adaptive waveform design. In: 2010 IEEE Workshop on Signal Processing Systems (SIPS), pp. 19–24 (October 2010)
7. Mountney, J., Silage, D., Obeid, I.: Parallel field programmable gate array particle filtering architecture for real-time neural signal processing. In: 2010 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), pp. 2674–2677 (August 2010)
8. Schwiegelshohn, F., Huebner, M.: An application scenario for dynamically reconfigurable fpgas. In: 2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), pp. 1–8 (May 2014)
9. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics, chap. Chapter 4.3 The Particle Filter, pp. 96–112. Intelligent Robotics and Autonomous Agents series, The MIT Press (2006)