Comparing Some Bayesian Filtering Techniques

Abheek Ghosh
Department of Computer Science
The University of Texas at Austin
2317 Speedway, Stop D9500
Austin, TX 78712

Abstract

We investigate and compare Bayesian filtering algorithms, methods like particle filter. The filtering algorithms we implement are fixed size grid (histogram) filter, tree-based dynamic grid (histogram) filter, particle filter, filter based on a polynomial of fixed degree, and a filter based on feed-forward deep neural networks. We develop a modular testing and simulation framework where these techniques and possibly other filtering techniques can be compared. Assuming that the estimation of a very very fine grid filter is the best that can be generated using the past history of samples, we compare the algorithms on the basis of: the error in the estimated mean state of the system, state with maximum estimated probability, and the L_2-error in the cumulative distribution function of the estimated belief.

We observe that for the localization of Nao robot, particle filter works better than other techniques given same computation resources. We anticipate that this is due to a combination of factors: the high error in the vision samples, simple belief distribution, non-requirement of high accuracy in state estimation. When we need very high accuracy and deterministic behavior, techniques like dynamic grid filter and neural network perform better.

Introduction

Uncertainty is ubiquitous in robotics, which has made probabilistic algorithms an essential part of it. One of the most important uses of the probabilistic algorithms is for the estimation of the state of a robot. The general algorithm used for this is the Bayesian filtering algorithm, pseudocode given below:

Algorithm 1 Bayes-Filter

Require: $bel(x_{t-1})$, u_t, z_t
1: for all x_t do
2: $\overline{bel(x_t)}$ = motion_update($bel(x_{t-1})$, u_t)
3: $bel(x_t)$ = sensor_update($\overline{bel(x_t)}$, x_t)
4: end for
5: **return** $bel(x_t)$

In the above algorithm, the time is discrete and denoted by t. x_t is the random variable representing the state at time t. Similarly, u_t is the motion measurement and z_t is the sensor measurement at time t. $bel(x_t)$ denotes the belief distribution. The motion_update updates the belief using the previous belief and current action measurement, and the sensor_update, similarly, updates the belief using current sensor measurement.

To practically implement a fast and reasonably accurate Bayesian filtering algorithm we need to decide upon these three main points:

1. Efficiently store/approximate the belief distribution $bel(x_t)$. The belief is essentially a probability distribution function. Some of the common methods used for function approximation are:
   - Storing the value of the function at a finite number of points in the domain.
   - Using a finite degree polynomial that fits the function with a least squared error.
   - Recently very popular and successful in several other areas, using a deep neural network.
   - As the belief is a PDF with its special properties, it is non-negative and sums (integrates) up to 1, we can store it using a fixed number of samples from the distribution.
2. Efficiently does the motion update. If the motion measurement has, or can be easily transformed into one that has, the same dimension as the state, and each dimension of the motion measurement has a one-to-one correspondence to that of the current state, then usually the update in the belief looks similar to the translation of the function.
3. Efficiently does the sensor update. This mainly involves finding the product of two PDFs.

The ideas mentioned in the (1) above motivate the techniques we study. But, as we will see later, (2) and (3) will have a significant impact on the performance of the algorithms.

For our project, we assume that the state, the motion measurement, and the sensor measurement have the same number of dimensions with one-to-one correspondence among the dimensions. This helps simplify the algorithms and our testing and simulation framework. We also assume that motion and sensor measurements can be queried by the algorithm in constant time.

Related Work

For linear systems with white noise, Kalman filter (KF) (Kalman 1960; Thrun, Burgard, and Fox 2005) is the optimal Bayesian filter algorithm and is used popularly. But KF doesn't work well for non-linear systems. For non-linear systems, the extended Kalman filter (EKF) (Thrun, Burgard, and Fox 2005) is one of the most popular filtering algorithms. EKF is usually implemented using a first order Taylor expansion, but higher order EKFs have also been used. Although the EKF (in its many forms) is a widely used filtering strategy, over several decades of experience with it has led to a general consensus within the tracking and control community that it is difficult to implement, difficult to tune, and only reliable for systems which are almost linear on the time scale of the update intervals (Julier and Uhlmann 1997). We implemented KF and EKF for assignment 4 ball tracking.

Unscented Kalman filter (UKF) is an extension of KF and EKF. It was introduced by (Julier and Uhlmann 1997; Julier, Uhlmann, and Durrant-Whyte 2000) with the accuracy of EKF with second-order Taylor approximation, and improved by (Wan and Van Der Merwe 2000) to make its accuracy comparable to third order. In UKF, a set of appropriately chosen weighted points are used to parameterize the means and covariances of the probability distributions. UKF has the same time complexity as first-order EKF while mitigating the sub-optimal performance and possible divergence.

Another aimilar algorithm is the Gaussian sum filter introduced in (Kotecha and Djuric 2003). In this method, there is a weighted sum of multiple non-correlated Gaussian particles. As a sum of Gaussian distributions is a Gaussian distribution, the estimated distribution is also Gaussian. In the paper, they show that under the Gaussianity assumption, the Gaussian particle filter is asymptotically optimal in the number of particles and, hence, has much-improved performance and versatility over other Gaussian filters, especially when nontrivial nonlinearities are present. On the other hand, it has lower complexity than particle filters.

The simplest non-parametric technique is the grid (histogram) algorithm. An extension to grid algorithm is the dynamic grid algorithm (Bucy 1975; Thrun, Burgard, and Fox 2005). In this algorithm, the grid is dynamically resized if the probability mass in the grid goes above a certain threshold. We implement these algorithms along with particle filter, they will be described in detail in the next section.

A different approach for filtering that we looked into is where the belief distribution is stored using a neural network. (Bobrowski et al. 2008) mention that organisms acting in uncertain dynamical environments often employ exact or approximate Bayesian statistical calculations in order to continuously estimate the environmental state, integrate information from multiple sensory modalities, form predictions and choose actions. They study how these computations could have been implemented in the neural network, and introduce the spike neural networks. Other relevant works used a recurrent neural network (RNN) to learn the internal behavior of the dynamic system, and a feedforward neural network attached to the RNN that outputs the estimated state (Haarnoja et al. 2016; Yadaiah and Sowmya 2006; Talebi et al. 2010). Another paper (Kanekar and Feliachi 1990) used a multilayer network that approximates the posterior, and gives output the state estimate when subjected to unknown noises.

Several analytical techniques are also used for filtering (Sorenson and Stubbeud 1968; Kushner 1967; Daum 1994). We mentioned only a few papers, but there is an enormous literature related to filtering, estimation theory, and their use in probabilistic robotics.

Algorithms and Implementation

In this section, we describe the filtering algorithms we used: fixed size grid filter, tree-based dynamic grid filter, particle filter, filter based on a polynomial of fixed degree, and a filter based on feed-forward deep neural networks. We implemented the algorithms in C++, the simulation data was generated using Octave (Matlab). The source code is available at *https://github.com/AbheekG/filter*. We used the Eigen library and some code from *https://github.com/yixuan/MiniDNN/* to implement the feed-forward neural network. For running in Nao, the LARG codebase given to the class was used.

The *Filter* C++ class contains the parameters and functions common to all filtering algorithms, like sensor and motion measurement functions, timing functions, I/O, etc. Each individual algorithm has its own derived class.

Particle Filter

It is similar to what we did in assignment 5 (localization), with slight modifications to fit our testing framework. The class *ParticleFilter* implements this algorithm. The number of particles is given as the hyper-parameter.

Fixed Grid Filter

In this method, the entire state space is divided into a grid of a fixed number of points. For each grid point, the value of the belief distribution is stored. The grid structure is pre-decided and doesn't change with the time or the belief distribution.

The algorithm works for any arbitrary dimension and grid size. Each grid is a hyperrectangle, e.g., a rectangle for 2-D. It stores the function values in a linear vector with appropriate indexing for multi-dimension. The class *GridFilter* implements this algorithm. The grid size is given as the hyper-parameter.

Dynamic Grid Filter

Similar to the fixed grid filter, in this method also the entire domain of the belief distribution is divided into sub-domains (grid block) and the value of the belief distribution is approximated for each grid block.

The entire grid is stored in the form of a tree. Each node corresponds to a grid block. A node of the tree is a leaf node if the total probability (weight) for the grid block (hyperrectangle) corresponding to the node is less than a given probability threshold. If the weight of a node is more than the probability threshold then its grid block is partitioned into its children. For example, the root of the tree has a grid block with the size of the entire domain for the robot state

Figure 1: Fixed grid filter. (Figure taken from (Thrun, Burgard, and Fox 2005))

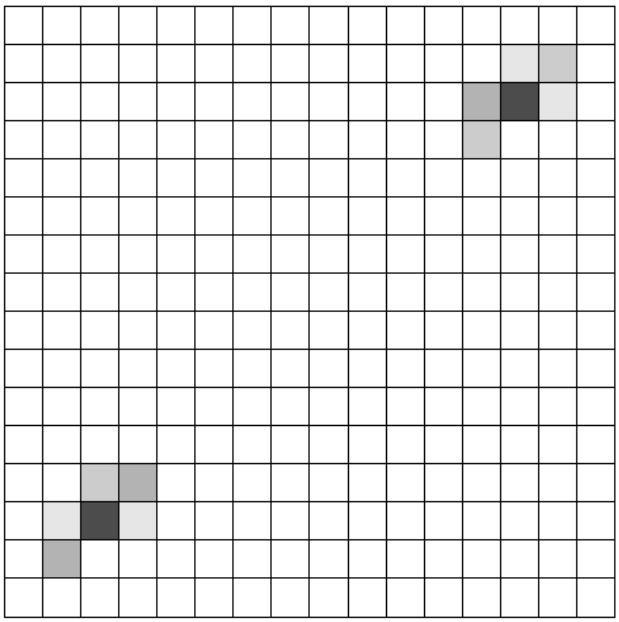
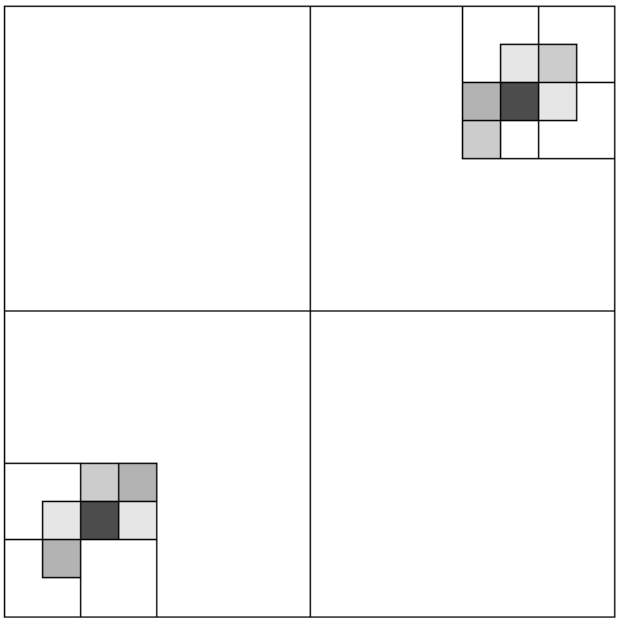


Figure 2: Dynamic grid filter. (Figure taken from (Thrun, Burgard, and Fox 2005))



and probability 1. If the probability threshold is less than 1 then the root will have children.

The nodes are branched and collapsed dynamically as the belief distribution changes. A node whose weight goes above the threshold is branched, while if it goes below the threshold it is collapsed and its children vanish. We also have a partial implementation for using different lower and upper probability thresholds. The class *DynamicGridFilter* implements this filtering method. The probability thresholds are given as the hyper-parameters.

Polynomial Filter

In this method, we approximate the belief using a polynomial of fixed degree. The original state is converted to a new modified state with higher order terms, and then the least square fit is done to find the coefficients of the polynomial. The class *PolynomialFilter* implements this algorithm. The degree is given as a hyperparameter. Due to lack of time, we could only have implementation up to degree 2.

Neural Network Filter

In this method, we use a feed-forward deep neural network (NN). The input to the neural network is a state and output is the value of the belief distribution corresponding to the state. Essentially, the function represented by the NN is the approximate belief distribution. The neural network is partially trained in each time step (sensor and motion update). For each time step, the input data (set of states), X, is generated depending on the previous belief (or NN), more samples from areas with higher weight and less from the lower weight. The output data, y, is generated as a product of current sensor measurement and previous belief. Currently, the motion is updated by doing a shift in the X, assuming that motion updates don't have an error.

The class *NeuralNetwork* implements this algorithm. All the standard hyper-parameters for a neural network, like the number of layers, size of each layer, learning rate, batch size, etc. are applicable here also.

Evaluation

We evaluated the algorithms on multiple datasets (included with the code, in $data$ folder). We will present statistical data for two of them. We also ported the fixed grid filter and dynamic grid filter algorithms to Nao robot and did the practical demonstration.

In the first test ($test1$ in $data$ folder), we have one-dimensional state space. Initially, the sensor measurements have 3 peaks. With time only one of the peaks survive. All the algorithms have around 100 variables. We keep this fixed across algorithms to measure their relative performance. Using any other number of variables also gives similar results. (Except polynomial filter, due to lack of time, we could implement only up to degree 2, so 3 parameters only.)

- Particle filter (PF): 100 particles.
- Fixed grid filter (FGF): 100 grid points.
- Dynamic grid filter (DGF): 50 to 100 grid points. (dynamic.)
- Neural network (NN): Two layers of size 10. $10 * 10 = 100$. (approx.)

In the second test ($test3$ in $data$ folder), we have multi-dimensional state space. Here also, all the algorithms have around 100 parameters.

Table 1: Error in the state with maximum probability.

Step	PF	FGF	DGF	PolyF	NN
1	0.7640	0.0640	7.8570	2.0640	4.5320
2	0.0510	0.0620	0.6720	7.9380	2.7760
3	0.0970	0.0600	2.3150	2.9910	2.9410
4	0.4800	5.6930	0.0020	1.8340	0.9330
5	0.5600	0.0390	1.8930	2.1310	1.6100
6	0.1020	0.0270	1.1850	1.0340	1.0370
7	0.2680	0.0330	0.2470	1.0580	0.8970
8	0.5250	0.0160	0.1320	0.8490	0.0010
9	0.2520	0.0230	0.1910	0.9160	0.3730
10	0.0820	0.0290	0.1310	0.7960	0.4750

Table 2: Error in the mean of estimated state.

Step	PF	FGF	DGF	PolyF	NN
1	0.5007	0.0010	0.1915	0.0224	0.0259
2	0.6320	0.0015	0.4271	0.2390	0.2663
3	0.3487	0.0018	0.7181	0.3715	0.4382
4	0.1031	0.0020	1.0043	0.5999	0.4570
5	0.3854	0.0021	1.2934	0.5197	0.2080
6	0.4882	0.0022	1.4082	0.0116	0.0971
7	0.4264	0.0023	1.3687	0.8871	1.0026
8	0.1852	0.0023	1.2192	1.8244	1.5367
9	0.0173	0.0023	1.0895	2.7725	2.1200
10	0.2027	0.0023	0.9733	3.5915	2.3507

Table 3: L2-error in computed CDF.

Step	PF	FGF	DGF	PolyF	NN
1	5.5895	0.3385	5.6089	4.7394	5.5770
2	3.1485	0.3799	4.7029	7.9123	8.0532
3	5.1527	0.4117	5.0591	8.9555	8.0353
4	4.5226	0.4256	5.4113	9.6027	7.7276
5	9.2158	0.4360	6.1948	10.0842	8.2515
6	7.7962	0.4820	5.6106	13.5393	9.0468
7	7.1645	0.5667	4.6130	18.8865	14.8883
8	7.4783	0.6547	4.0991	21.6299	12.3882
9	5.9185	0.7172	3.4860	22.7425	12.8611
10	5.7165	0.7603	3.0654	22.6650	10.1677

Figure 3: Test 1. Cumulative Distribution Functions after 3 time steps.

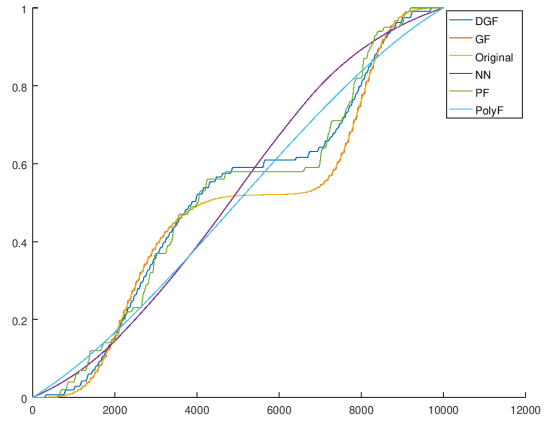


Figure 4: Test 1. Cumulative Distribution Functions after 6 time steps.

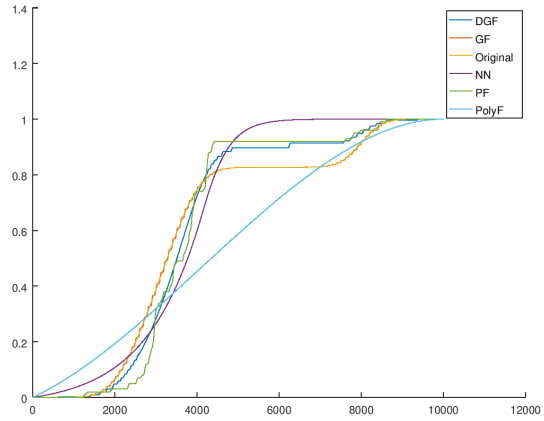


Figure 5: Test 1. Cumulative Distribution Functions after 10 time steps.

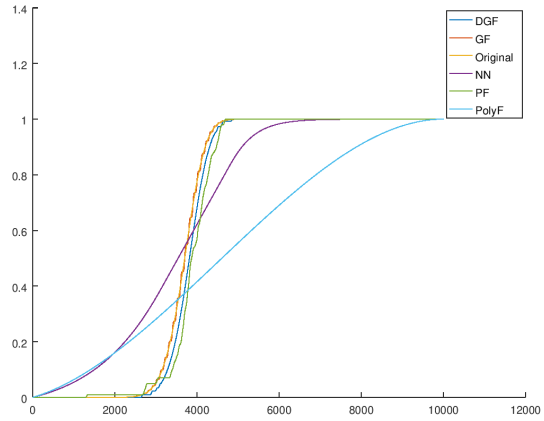


Table 4: Error in the state with maximum probability.

Step	PF	FGF	DGF	PolyF	NN
1	73.5391	85.7030	87.8465	74.4648	
2	72.0139	4.0000	72.0069	84.4808	80.2808
3	81.0247	35.8469	21.4709	19.0263	85.0706
4	51.4782	1.0000	64.4127	111.3598	
5	76.0592	50.2494	2.0000	48.7647	109.6586
6	1.0000	40.3113	53.8516	109.6586	
7	74.6860	28.1603	54.7449	47.6760	110.4943
8	75.1532	19.4165	57.8705	48.0416	110.0727
9	42.0595	26.9258	4.4721	6.3246	56.8595
10	20.5913	32.9848	1.4142	7.0711	58.5235

Table 5: Error in the mean of estimated state.

Step	PF	FGF	DGF	PolyF	NN
1	20.3808	15.3506	8.1420	13.7211	63.1493
2	36.8619	36.3728	17.3435	30.9885	128.6551
3	43.8748	40.6532	18.4665	38.5199	195.4936
4	20.8605	70.5275	16.4776	43.2564	267.7114
5	22.3279	108.4227	24.0023	62.1079	344.1171
6	62.2529	138.1839	50.4984	95.0064	422.2877
7	112.1986	149.7854	82.5249	126.4093	496.9704
8	165.0571	149.9109	115.0814	156.5516	569.8333
9	197.9973	125.5558	115.3565	151.7752	611.9340
10	205.5056	93.5752	113.8170	149.8863	654.5354

- Particle filter (PF): 100 particles.
- Fixed grid filter (FGF): 100 grid points.
- Dynamic grid filter (DGF): 50 to 100 grid points. (dynamic.)
- Neural network (NN): Two layers of size 10. $10 * 10 = 100$. (approx.)

In these experiments, we observed that NN and PolyF severely underperformed. Also, using appropriate hyperparameters for the NNs can improve the performance significantly.

In the first test, the FGF performed better than the PF. The PF and the DGF performed comparably. In the second test, the FGF was worse than PF and DGF. In all these tests, although the DGF and PF have the same number of parameters, DGF takes much more processing time than PF. This makes using DGF in Nao much slower.

Table 6: L2-error in computed CDF.

Step	PF	FGF	DGF	PolyF	NN
1	7.6426	10.2127	9.9363	11.2119	32.0021
2	10.5246	14.1116	11.6408	15.4432	51.6483
3	5.9152	23.5160	1.9137	10.3824	63.1219
4	12.5345	15.8151	2.9442	11.9974	56.4408
5	25.6487	7.7291	5.7329	16.4528	46.7826
6	24.7311	6.4264	5.8499	17.4547	46.0956
7	32.6005	5.0391	5.9658	14.7962	50.0060
8	37.1039	7.5916	10.7382	13.6126	47.1922
9	47.8120	19.3311	9.7274	19.8916	26.0166
10	13.7839	11.4408	4.3487	20.5015	25.8191

Conclusion

In this project, we used some standard filtering techniques, like the grid filter (fixed and dynamic) and the particle filter, and some slightly unconventional ones, like the polynomial filter and the neural network. We made the algorithms work on both simulation and Nao, albeit they didn't perform very well. The algorithms were far more complex to implement than the particle filter but performed worse than the particle filter. Some possible reasons that the algorithms didn't work well:

- The sub-algorithms used for incorporating sensor and motion updates as part of the individual filtering algorithms may not be the best way to do it. For example, in the dynamic grid filtering algorithm, for sensor update of each grid block, we have to reach the full depth of the tree multiple times. For some motion updates, the entire tree has to be reconstructed.
- The implementation of the algorithms is not efficient.
- In areas where the polynomial methods and neural networks are used and the neural networks have given tremendous results, the estimated function doesn't vary over time. The NNs have to learn a complex fixed function using a lot of samples. In our case, the belief distribution varies over time. In some experiments, we found that once the neural network is already trained it gets stuck there and cannot approximate an entirely new function.
- The belief distributions are not very complex, at least for the examples we constructed and for Nao's localization.
- The error in sensor and motion measurements is very high. The high accuracy provided by deep NNs becomes irrelevant here.

Acknowledgements

We thank Prof. Peter Stone and TA Josiah Hannah for their guidance, regular valuable comments, and providing us with the Nao robot and appropriate development environment.

References

Bobrowski, O.; Meir, R.; Shoham, S.; and Eldar, Y. 2008. A neural network implementing optimal state estimation based on dynamic spike train decoding. In *Advances in Neural Information Processing Systems*, 145–152.

Bucy, R. S. 1975. Nonlinear filtering. Technical report, University of Southern California, Los Angeles, Dept. of Aerospace Engineering and Mathematics.

Daum, F. E. 1994. New exact nonlinear filters: Theory and applications. In *Signal and Data Processing of Small Targets 1994*, volume 2235, 636–650. International Society for Optics and Photonics.

Haarnoja, T.; Ajay, A.; Levine, S.; and Abbeel, P. 2016. Backprop kf: Learning discriminative deterministic state estimators. In *Advances in Neural Information Processing Systems*, 4376–4384.

Julier, S. J., and Uhlmann, J. K. 1997. New extension of the kalman filter to nonlinear systems. In *Signal processing, sensor fusion, and target recognition VI*, volume 3068, 182–194. International Society for Optics and Photonics.

Julier, S.; Uhlmann, J.; and Durrant-Whyte, H. F. 2000. A new method for the nonlinear transformation of means and covariances in filters and estimators. *IEEE Transactions on automatic control* 45(3):477–482.

Kalman, R. E. 1960. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering* 82(Series D):35–45.

Kanekar, A. J., and Feliachi, A. 1990. State estimation using artificial neural networks. In *System Theory, 1990., Twenty-Second Southeastern Symposium on*, 552–556. IEEE.

Kotecha, J. H., and Djuric, P. M. 2003. Gaussian sum particle filtering. *IEEE Transactions on signal processing* 51(10):2602–2612.

Kushner, H. 1967. Approximations to optimal nonlinear filters. *IEEE Transactions on Automatic Control* 12(5):546–556.

Sorenson, H. W., and Stubbeud, A. R. 1968. Non-linear filtering by approximation of the a posteriori density. *International Journal of Control* 8(1):33–51.

Talebi, H. A.; Abdollahi, F.; Patel, R. V.; and Khorasani, K. 2010. Neural network-based state estimation schemes. In *Neural Network-Based State Estimation of Nonlinear Systems*. Springer. 15–35.

Thrun, S.; Burgard, W.; and Fox, D. 2005. *Probabilistic robotics*. MIT press.

Wan, E. A., and Van Der Merwe, R. 2000. The unscented kalman filter for nonlinear estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, 153–158. Ieee.

Yadaiah, N., and Sowmya, G. 2006. Neural network based state estimation of dynamical systems. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, 1042–1049. IEEE.