Particle Filter Implementation for Target Tracking in a Network of Sensors

Imtiaz Nizami
nizami@asu.edu

Project Supervisor:

Darryl Morrell
Arizona State University, GWC 416
Department of Electrical Engineering

December 15, 2003

Contents

List of Figures

List of Tables

1 Introduction

Object detection and tracking is of importance in many diverse fields. A few of the numerous applications that require target detection and tracking are mentioned below:

- Visual surveillance system
- Intelligent transport system
- Object pose estimation
- Automotive applications
- Missile guidance
- Etc

This project deals with a subclass of the general tracking problem, namely when the object of interest is present in a sensor field with known sensor locations. By tracking we mean that the target gets identified by some signature signal and then the target's path is estimated over k time steps. Specific applications might be robot navigation and tracking of a military vehicle. Using such network of sensors have the following advantages:

- Network of sensors has the capability of being extended without much alteration to the original system.
- Individual sensor failure is less likely to produce large estimation error.
- Inexpensive sensors can be deployed over the area of attention quickly and without high costs.
- Can be used in remote or inaccessible areas.

In many tracking problems some kind of uncertainty is involved. These uncertainties may arise from sensor errors, measurement errors, channel noise etc. In order to have a good system, these uncertainties have to be modeled using random variables and stochastic processes. Thus the use of stochastic methods becomes inevitable to handle random phenomenon in such dynamic processes as target tracking.

Several methods have been devised to solve the problem of target tracking. In this project, a particle filter approach is implemented. The following data is to be estimated:

- x: position of the x-coordinate
- y: position of the y-coordinate

- $\dot{x}$: velocity in the direction of the x-coordinate

- $\dot{y}$: velocity in the direction of the y-coordinate

The goal of this project is to investigate the performance of sensor networks in estimating the target's position and velocity. Each sensor in the network is assumed to be at a known location. The target position and velocity is estimated first using a particle filter and then a Rao-Blackwellized version of the particle filter. Performance of two kind of sensor networks is analyzed; namely configured sensor network and a non-configured sensor network. In the non-configured sensor network all the sensors are assumed to be active and contribute to the observation vector $\mathbf{z}_k$. In contrast, the in a configured sensor network only a subset of sensors are active and contribute to the observation vector $\mathbf{z}_k$. This may be desireable if one want to reduce the power consumption or cumputational complexity etc. The strategy used to reduce the active number of sensors is documented in section 4 of [1], and is also explained in section 6 of this paper. Performance of the sensor networks is analyzed when the sensors are placed in an even fashion and when placed randomly.

Each sensor decides wherther the target is present or absent. This decision is made using M samples of the recieved signal. This recieved signal reaches the sensor in a noisy channel. Hence, the decision made is not necessarily accurate. After making the decision, the sensor transmits a one or a zero if the target is present or absent respectively. This transmission takes place every Δt seconds, where Δt is the time interval between any two consecutive transmissions. Based on this possibly inaccurate observation, the particle filter estimates the position and velocity of the target.

The layout of this paper is as follows. In section 2 the target model is explained. Section 3 deals with the sensor model, whereas section 4 details how the observation is obtained from the sensor network. Section 5 briefly mentions the stochastic filters used, giving references to the detailed documentation of the same. Sensor configuration for reducing the number of active sensors is provided in section 6. Simulation resutls are provided in section 7, complemented with comments on the performance results. Lastly, conclusions are laid in section 8.

2 Target Model

The target model is a simple discrete-time, linear, constant-velocity model. The noise attributed with the system is WGN with constant covariance matrix Q. The target motion is restricted to a two dimensional plane.

Mathematically the model can be represented as:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{w}_k \tag{1}$$

where:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the state vector is:

$$\mathbf{x} = \begin{bmatrix} x(t_k) & y(t_k) & \dot{x}(t_k) & \dot{y}(t_k) \end{bmatrix}^T$$

We make assumption that the observations are made at evenly spaced time intervals Δt.

3 Sensor Model

The sensor network is described in detail in [2]. It is essentially a system of N small devices deployed over an area in an attempt to sense a signal transmitted by the target. Sensors can either be placed in a regular pattern or deployed in an ad-hoc manner. The binary decision is made at each time step k on the the basis of M samples of the received signal. It is assumed that M samples are gathered in a short time which is negligible compared to the time interval Δt between successive time steps k and $k+1$. At a particular time step, k, each sensor can either be active or inactive. Each active sensor makes a binary decision about whether the target is present or absent. This decision is based on the energy from the recieved signal. If the energy is greater than a defined threshold (N-P criteria, see [3]), the target is assumed to be present.

The target signal is modeled as white Gaussian process with a given mean and covariance. The noise attributed to observation from each sensor is assumed be WGN with same covariance for every sensor. Target signal and noise at each sensor is independent of each other. Notation used from this point onwards is as follows:

σ_N^2: noise variance at each sensor

$\sigma_{T_0}^2$: energy per sample of the target signal at a distance of 1 unit

d_i: distance between target and the i^{th} sensor

μ: Threshold to decide if the signal is present

M: Samples of the recieved signal at each time step

N: Number of particles used

$P_D(d_i)$: Probability of detection attributed with the i^{th} sensor

P_{FA}: Probability of false alarm

β: is defined as $\frac{\mu}{\sigma_N^2}$

H_0: Hypothesis that the target is not present

H_1: Hypothesis that the target is present

Energy per sample of the target signal at the i^{th} sensor is

$$\sigma_T^2(d_i) = \frac{\sigma_{T_0}^2}{d_i^2}$$

The detection process at each sensor is explained in this paragraph. The detection is based on theory provided in [3, pages 142-144]. The received signal at each sensor is an M-element vector denoted ρ. Each sensor performs a hypothesis test between H_0 (target absent) and H_1 (target present). Hypothesis H_0 dictates that the target is far away from the sensor and the energy recieved from the target is negligible. Thus under H_0, ρ is a Gaussian vector whose elements are independent with variance σ_N^2. Similarly, hypothesis H_1 indicates that the target is close to the sensor and energy recieved from the target is substantial. Thus, under H_1, ρ is a Gaussian vector whose elements are independent with variance $\sigma_N^2 + \sigma_T^2(d_i)$. The N-P criteria suggests the following test

$$H_0 : \sigma_N^2$$

$$H_1 : \sigma_N^2 + \sigma_T^2(d_i)$$

$$p(x; H_0) = \frac{1}{\sqrt{2\pi\sigma_N^2}} \; e^{-\frac{x}{2\sigma_N^2}}$$

$$p(x; H_1) = \frac{1}{\sqrt{2\pi[\sigma_N^2 + \sigma_T^2(d_i)]}} \; e^{-\frac{x}{2[\sigma_N^2 + \sigma_T^2(d_i)]}}$$

$$\mathcal{L}(x) = \frac{p(x; H_1)}{p(x; H_0)}$$

Carrying out the above N-P test, we can come up with the following optimal threshold test:

$$\rho^T \, \rho \;\; \lessgtr \;\; \mu$$

where the threshold μ is selected based on a desired P_{FA}. The P_D and P_{FA} can be computed using equations 2 and 3:

$$P_{FA} = \frac{\Gamma(\frac{M}{2}, \frac{\beta}{2})}{\Gamma(\frac{M}{2})} \tag{2}$$

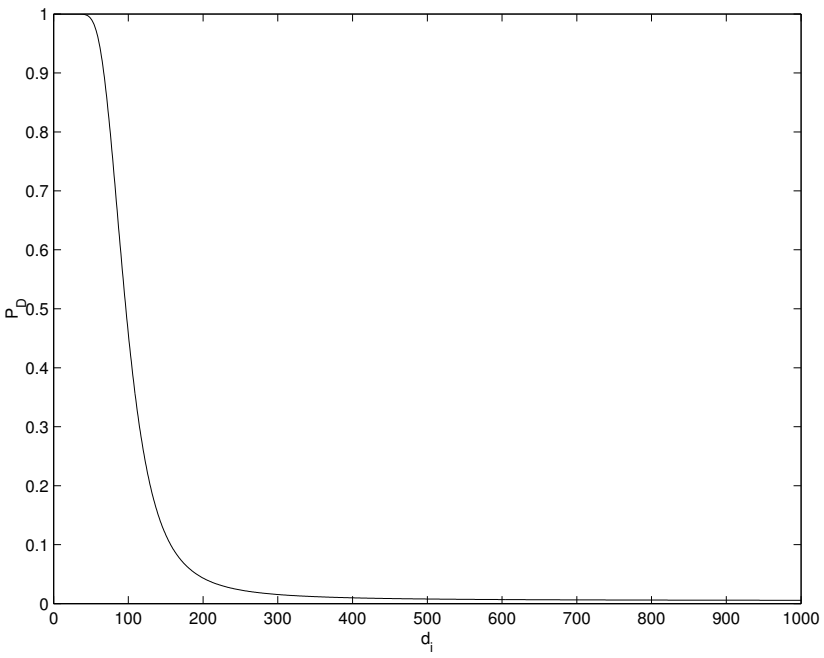

Figure 1: Sensor detection probability as a function of target-sensor distance

$$P_D(d_i) = \frac{\Gamma\left(\frac{M}{2}, \frac{\beta}{2\left(1+\frac{SNR_0}{d_i^2}\right)}\right)}{\Gamma(\frac{M}{2})} \tag{3}$$

In equations 2 and 3 we have defined $\beta \triangleq \frac{\mu}{\sigma_N^2}$ and $SNR_0 = \frac{\sigma_{T_0}^2}{\sigma_N^2}$ is the signal-to-noise ratio at a distance of one unit from the target. The symbol $\Gamma(.,.)$ corresponds to the incomplete Gamma funciton (a different implementation of which is provided in matlab as gammainc). As can be seen from equation 3, the $P_D(i)$ depends upon the distance between the i^{th} sensor and the target. A typical plot of P_D with respect to distance is shown in Figure 1. P_{FA} is computed from the cumulative distribution function of a chi-squared random variable. A table of parameter values provided in [1] is also included in Table 1.

4 Obtaining the Observation

For the purposes of this project, two observation models were studied. These are given in [1] and [2] respectively. The model maintained in [2] is based on the assumption that at a particular time only one sensor can detect the target. On the other hand, the model in [1] does not pose such a restriction. This project make use of the model given in [1].

M	Desired P_{FA}	β	μ
10	0.005	25.2	$25.2\sigma_N^2$
10	0.01	23.2	$23.2\sigma_N^2$
10	0.1	16.0	$16.0\sigma_N^2$
20	0.005	25.2	$25.2\sigma_N^2$
20	0.01	23.2	$16.2\sigma_N^2$
20	0.1	16.0	$25.0\sigma_N^2$

Table 1: Thresholds to achieve desired values of P_{FA}

The size of the observation vector is equal to the number of active sensors. At a given time k, the vector $\mathbf{z}_k$ contain binary observations from each active sensor. If a sensor detects the target, corresponding element of $\mathbf{z}_k$ is one; otherwise the corresponding element of $\mathbf{z}_k$ is zero. Thus, the probability distribution $p(z_k(i)|x_k(i))$ attributed with a single sensor is given as:

$$p(z_k(i)|\mathbf{x}_k) = [P_D(d_i)]^{z_k(i)}[1 - P_D(d_i)]^{1-z_k(i)}$$

We can observe the strong dependence of P_D on d_i. The probability distribution of the vector $\mathbf{z}_k$ is thus

$$p(\mathbf{z_k}|\mathbf{x_k}) = \prod_{i=1}^{n}[P_D(d_i)]^{z_k(i)}[1 - P_D(d_i)]^{1-z_k(i)} \tag{4}$$

5 Particle Filter Implementation

A simple particle filter and a Rao-Blackwellized version of the particle filter are used to determine the performance of the sensor networks. The details of the particle filter can be studied from [4]. The poposal distribution used is the conditional state density $p(x_k|x_{k-1})$ obtained from the dynamics model presented in equation 1. As our observations only depend on the target position, it seems a good idea to implement a Rao-Blackwellized version of the particle filter. This is done using the equations derived in [5].

The basic idea behind particle filtering is to represent the location density funtion by a set of random points (or particles) which are updated based on the observation (sensor readings) and the target model (provided in equation 1). Estimation of the true location is then performed using these particles. Rao-Blackwellization comes into play if $p(x_k|z_{1:k})$ can be factored into two terms, one of which can be computed analytically. This may lead

to a reduced number of particles with good performance. It is not always clear if the Rao-Blackwellization is a good idea for the problem at hand. A collection of particles $x_k^{(j)}$ and associated weights $w_k^{(j)}$ are used to estimate the state and its associated covariance matrix.

$$\hat{\mathbf{x}} = \sum w_k^{(j)} \mathbf{x_k^{(j)}} \tag{5}$$

$$P_{k|k} = \sum w_k^{(j)} (\mathbf{x_k^{(j)}} - \hat{\mathbf{x}})(\mathbf{x_k^{(j)}} - \hat{\mathbf{x}})^T \tag{6}$$

A low error variance scheme is used for resampling. The scheme progress as follows:

$r \sim U[0, \frac{1}{N_s}] \longrightarrow$ to get first sample

$r + \frac{1}{N_s} \longrightarrow$ to get second sample

$r + \frac{2}{N_s} \longrightarrow$ to get third sample

$\downarrow$

Resampling is done at each iteration and hence the weight update equation for each particle is simplified to

$$w_k = p(z_k | \mathbf{x_k})$$

6 Sensor Configuration

In an attempt to minimize the sensor usage, the performance evaluation is done for a reduced set of sensor network. This reduced set of sensor network dynamically determines if a particular sensor should be active or inactive at each successive time step. This decision is based on the previous state estimate and the squared position error on the state estimate. This minimization is done such that the accuracy of the results remain moderately close to the case where all the sensors are active. The following two parameters are to be defined before elaborating on the strategy:

$$e_k = P_{k|k}^{1,1} + P_{k|k}^{2,2} \tag{7}$$

and

$$r_k = C\ e_k \tag{8}$$

This paragraph describes the heuristic sensor configuration algorithm used in [1]. In equation 7, $P_{k|k}^{1,1}$ and $P_{k|k}^{2,2}$ denote the first and second diagonal elements of the covariance matrix $P_{k|k}$. C denote a constant term for tweaking the radius r_k. It is helpful to consider Figure 2, where $\hat{x}_{k|k}$ denotes the estimate at time step $k-1$, and r_k is the same as given in equation 8. The rectangle in Figure 2 represents the area in which the target can move. At

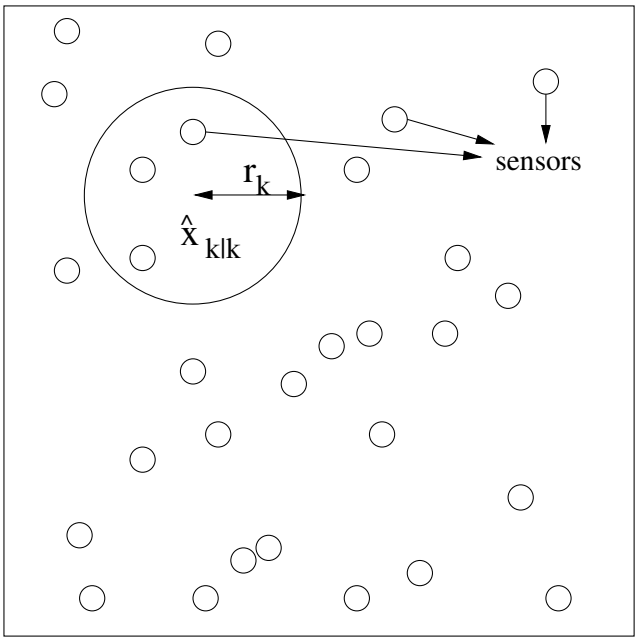

Figure 2: Activation of sensors within a distance r_k of the target position estimate $\hat{x}_{k|k}$

each time step k, only sensors in the circle with radius r_k are active. The strategy makes a lot of intuitive sense as the radius r_k is infact depending on the squared error of the position estimate. When the squared error from time step $k-1$ is low, radius r_k will be small, hence decreasing the number of active sensors. Similarly, when the squared error from previous time step $k-1$ is large, r_k will be large thus increasing the number of active sensors.

The choice of parameter C depend on simulation results. In [1] C was chosen to be $1/36$. According to simulations results from this project $C = 1$ gave the best results, thus using $r_k = e_k$. The values of $\hat{x}_{k|k}$ and $P_{k|k}$ are computed by the particle filter. Only the sensors that remain in the circle of radius r_k are used in the following time step to update the state estimate.

7 Simulation Results

The performance criteria of choice is the average squared error. All the sumulations are performed for the case where the initial state is known. In other words, true initial state of the target is provided to the filter. One other factor that should be kept in mind is that in the case where sensors are randomly distributed, the random distribution is chosen once. All the Monte Carlo iterations are based on this particular distribution of sensors. This oc-

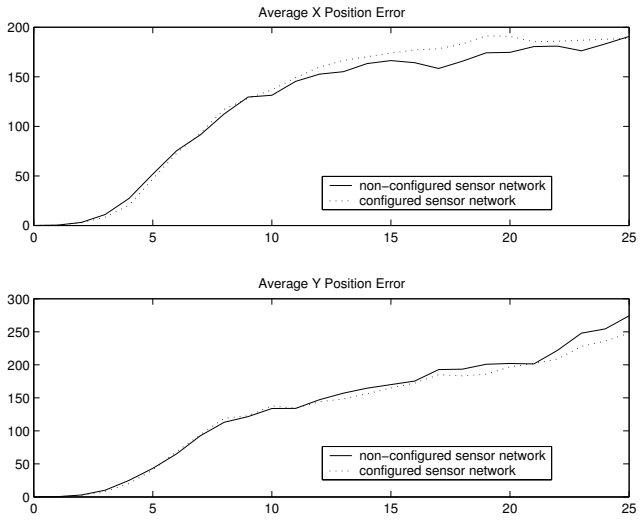

Figure 3: Particle filter using evenly spaced sensors and 100 particles

cured without intention and only was noticed once all the simulations were performed. The reason for this mistake is that while thinking over the layout of the system, the mind-set was that once sensors are deployed over an area, their locations cannot be changed.

The radius r_k is always taken to be $r_k = e_k$. Unless stated otherwise, the following assumptions are made while simulating the filter. The target was tracked using 100 sensors distributed in a 1 km square area. The default values of important simulation parameters were:

- $N = 500$
- $M = 20$
- $\beta = 40$
- $SNR_0 = 10^4$

7.1 Particle filter using evenly spaced sensors and 100 particles

From Figure 3, it is clear that the performance of the configured and non-configured cases have a noticeable difference. The advantage of the configured network was that it used only about 19% of the total number of sensors. This is a significant decrease in the number of sensors used, while the average squared error has not changed significantly. It is interesting to note that in the case of y-position error, the configured network performs better than that of the non-configured case. This might be due to the reason that there were more false

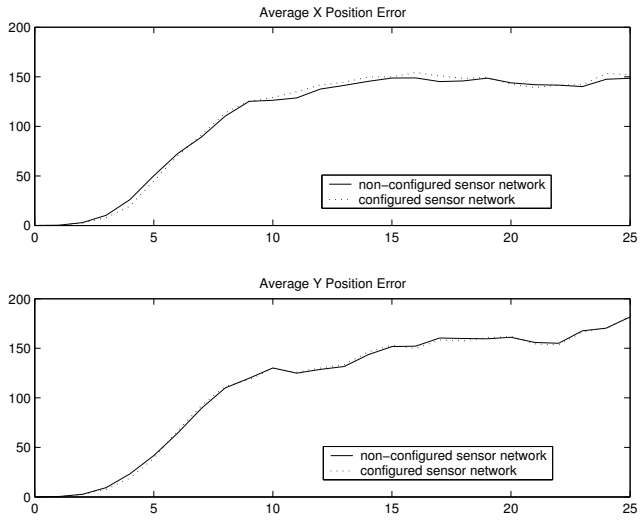

Figure 4: Particle filter using evenly spaced sensors and 1000 particles

alarms in the case of the non-configured network. Also, reducing the number of particles should decrease the performance of the configured network relative to non-configured case.

7.2 Particle filter using evenly spaced sensors and 1000 particles

In the case when 1000 particles were used, the performance of the configured and non-configured cases were almost identical. From Figure 4, it is seen that performance plots for both x-position and y-position are overlapping for the two sensor configurations. The average number of sensors used for the configured case turns out to be 21% of the total number of sensors placed in the field. This is again a significant decrease in the number of sensors used, while it is difficult to see the difference in the plots of average squared error.

7.3 Particle filter using randomly distributed sensors and 100 particles

From Figure 5, we can see that there is a slight difference in the error performance of the configured and non-configured networks. This difference is not too much even when we are dealing with 100 particles. Again, this might be due to the fact that random locations for the sensors were chosen only once, and remained constant over all Monte Carlo iterations (this was mentioned at the begining of this section). It can also be the case that the total number of sensors being used is so large that no performance degradation is noticeable. Infact, simulations for a smaller number of particles showed noticeable differences in the performance of the two sensor configurations. Simulation results show that on average, 18% of the total sensors were used in the configred network case. This is a significant decrease in the number of sensors used, while the average squared error has not changed significantly.

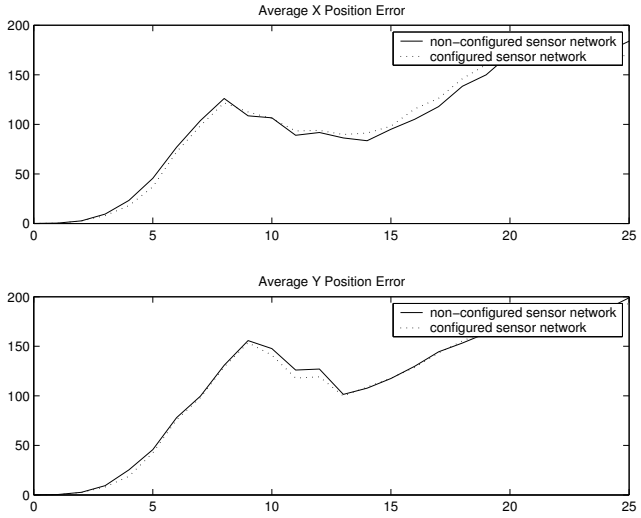

Figure 5: Particle filter using randomly distributed sensors and 100 particles

7.4 Particle filter using randomly distributed sensors and 1000 particles

In the case where 1000 particles were used, the performance of the configured and non-configured cases were almost identical. From Figure 6, it can be seen that performance plots for both x-position and y-position are overlapping. The average number of sensors used for the configured case turns out to be 20% of the total number of sensors placed in the field. This is again a significant decrease in the number of sensors used, while it is difficult to see the degradation in the plots of average squared error for the two cases.

7.5 Rao-Blackwellized filter using evenly spaced sensors and 100 particles

From Figure 7, the difference between the performance of the configured and non-configured sensor network is obvious. This performance degradation is achieved by reducing the number of active sensors by 74%. This can be very desireable when power saving is much required, e.g., tracking a robot on Mars. As true initial state is provided to the filer, both configured and non-configured case start out with almost the same squared errors. Over time a clear indication of performance loss in the case of configured sensor network is observed. This is what we expect, as we are reducing the number of sensors and thus reducing the total probability of detection. Another notification is that the performance of the Rao-Blackwellized version of particle filter is inferior compared to a simple particle filter. Also the average error difference between the configured and non-configured case for a Rao-Blackwellized particle filter is worse when compared with a similar case of simple particle filter, as can be seen in Figure 3.

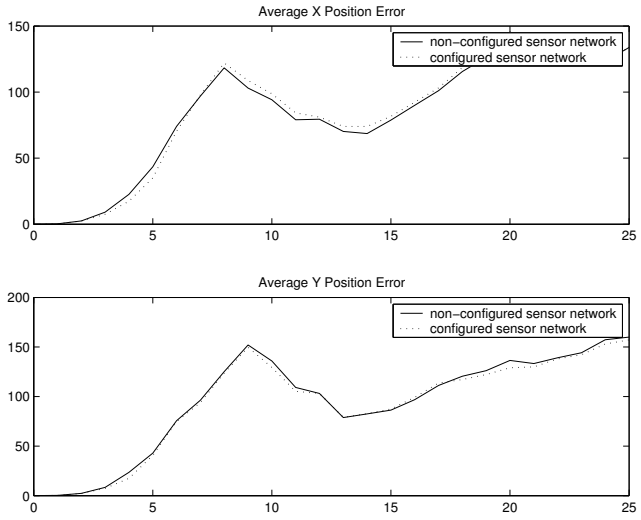

Figure 6: Particle filter using randomly distributed sensors and 1000 particles

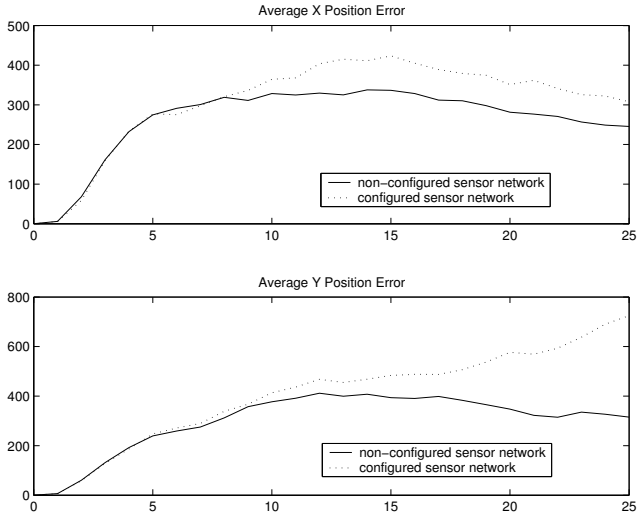

Figure 7: Rao-Blackwellized filter using evenly spaced sensors and 100 particles

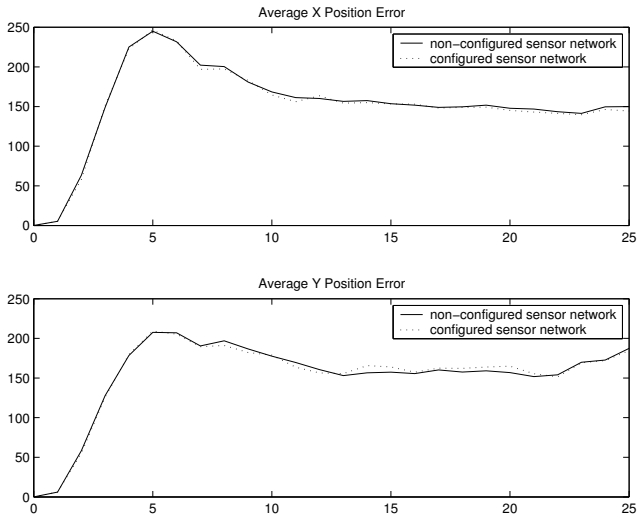

Figure 8: Rao-Blackwellized filter using evenly spaced sensors and 1000 particles

7.6 Rao-Blackwellized filter using evenly spaced sensors and 1000 particles

In the case of Rao-Blackwellized filter with 1000 particles in use, the performance of the configured and non-configured networks were almost identical. From Figure 8, it is seen that performance plots for both x-position and y-position are overlapping. The average number of sensors used for the configured case turns out to be 31% of the total number of sensors placed in the field. This is again a significant decrease in the number of sensors used, while it is difficult to see the difference in the plots of average squared error.

7.7 Rao-Blackwellized filter using randomly distributed sensors and 100 particles

From Figure 9, the difference between the performance of the configured and non-configured sensor network is obvious. This performance degradation is achieved by reducing the number of active sensors by 72%. Again it can be noted that the performance for the case of evenly distributed sensors was better. This is also clear by comparing Figure 9 with Figure 3. This difference between the performace of evenly placed sensor network and randomly distributed sensor network was much more evident when total number of sensors was small.

7.8 Rao-Blackwellized filter using randomly distributed sensors and 1000 particles

In the case of Raso-Blackwellized filter when 1000 particles are in use, the performance of the configured and non-configured cases are almost identical. From Figure 10, it is seen that

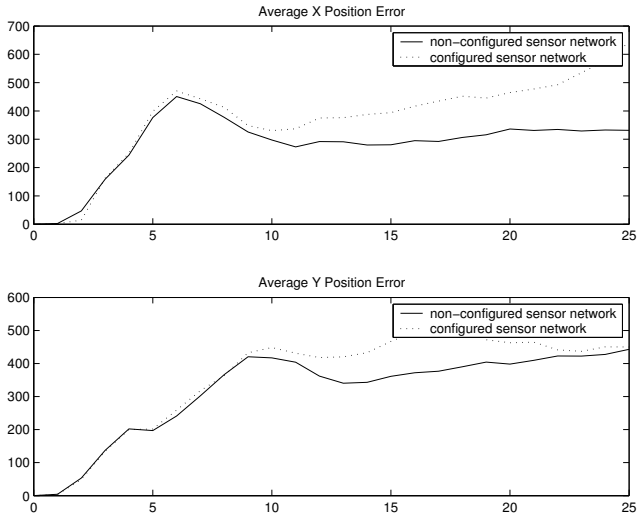

Figure 9: Rao-Blackwellized filter using randomly distributed sensors and 100 particles

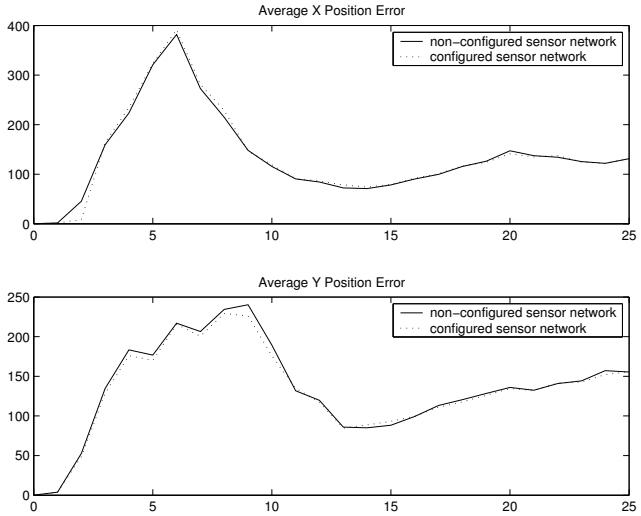

Figure 10: Rao-Blackwellized filter using randomly distributed sensors and 1000 particles

performance plots for both x-position and y-position are overlapping. The average number of sensors used for the configured case turns out to be 34% of the total number of sensors placed in the field. This is again a significant decrease in the number of sensors used, while it is difficult to see the difference in the plots of average squared error.

8 Conclusion and Future Work

8.1 Conclusion

The conclusions are divided into the following sub-sections for easier performance evaluation.

- Particle filter Vs. Rao-Blackwellized filter
- 100 particles Vs. 1000 particles
- Evenly placed sensor network Vs. randomly distributed sensor network
- Other Remarks

8.1.1 Particle filter Vs. Rao-Blackwellized filter

Particle filter provided visibily better results than the corresponding Rao-Blackwellized filter. This was the case even when same number of particles were used by both the filters. Thus, effort put into the Rao-Blackwellization of the paritcle filter was not worth the trouble. Another notification was that the error difference between the configured and the non-configured networks was more in the case of Rao-Blackwellized filter when compared with particle filter. Also, configured case of Rao-Blackwellized filter used more sensors on average than that of the configured case of particle filter. One explanation to this is the fact that in a sensor network problem the accuracy of position estimate is not good, which results in poor perfomance by the Rao-Blackwellized filter.

8.1.2 100 particles Vs. 1000 particles

Using 1000 particles showed better performance when compared with the case of 100 particles. Although there was a performance difference, it was not as evident as expected before the simulations were carried out. This turned out the case as the total number of sensors used were large. When using a smaller number of sensors, performance difference increase substantially. Also, it seems that for a sensor network type of problem, the performance increase with increasing number of particles do not have a constant slope. After reaching a certian number of particles, the increase in performance come with a much larger increase in the number of particles. This is as expected, because in estiamtion problems we usually have a bound on the performace, e.g. Cramer-Rao bound. The filter cannot exceed the performance once the bound is reached.

8.1.3 Evenly placed sensor network Vs. randomly distributed sensor network

Although, there was a performance degradation when a randomly distributed sensor network was used, the comparative degradation was smaller than expected. There can be two possible explanations of this behaviour. One is that the total number of sensors placed in the filed is large. Using less sensors will give results that can elaborate on the performance difference between the two sensor placement strategies. Another reason, as was mentioned earlier, might be the random placement of sensors. This random placement was done once, and all the Monte Carlo iterations depended on this sensor placement. This was done as an oversight. The thought process while designing simulation code was that once the sensors are deployed, they cannot change locations. It was noticed only after all the simulations were completed that for the purpose of performance evaluation the sensor placement should have changed at each Monte Carlo iteration.

8.1.4 Other Remarks

From Figure 1, P_D is monotonically decreasing as a function of distance. This tells us that the strategy explained in section 6 should actually perform very well depending on the chosen threshold. It was also noticed during the course of the project that the Rao-Blackwellization of the filter does not depend on the observation model. An important factor in getting good estimates was the choice of inital state and covariance matrix. Our model allowed the target to move within 1000 square units. It was noticed that if the initial state provided to the particle filter has an error that is greater than 150 units, the filter performance was very bad. The choice of initial covariance matrix had a huge impact on the performance Rao-Blackwellized filter. A bad initial covariance matrix resulted in entirely incorrect estimates. The covariance had little effect on the performance of the particle filter.

Our results were based on the assumption that the total number of sensors is hundred and the particle filter knows the true initial state of the target. Figure 11 show a sample rum of the Rao-Blackwellized filter with fifty sensors and true initial state provided to the filter. As can be seen, the estimated values are far from the true trajectory. Even the initial values are not matched well. This effect can be due to the initial choice of covariance matrix. Figure 12 shows a similar sample run with fifty esnsors and when the true state is not given to the filter. As expected, the error in the configured case is greater than that of the non-configured case.

8.2 Future Work

The results in this project are based on the assumption that the initial state of the target is known. Although simulations were performed for the case of poor initial states, they were not extensive and therefore are not documented. Therefore, the first step towards a better understanding of the network might be to perform the simulations with varied initial states. In addition, all the results are based on the assumption that a hundred sensors are in

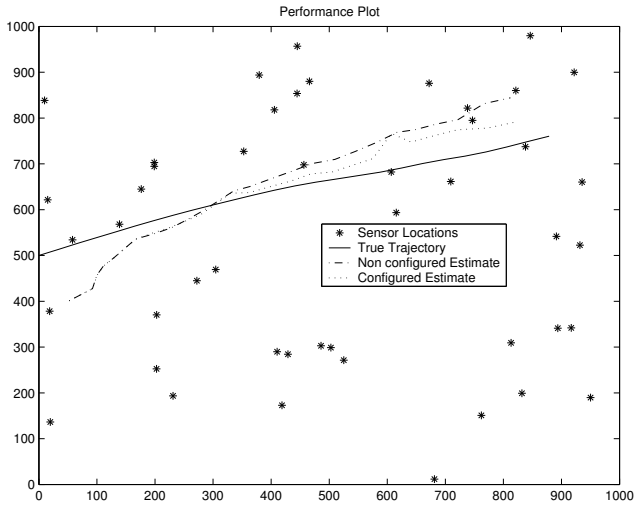

Figure 11: Sensor laydown whith true track and estimates for the case of configured and non-configured networks with true initial state

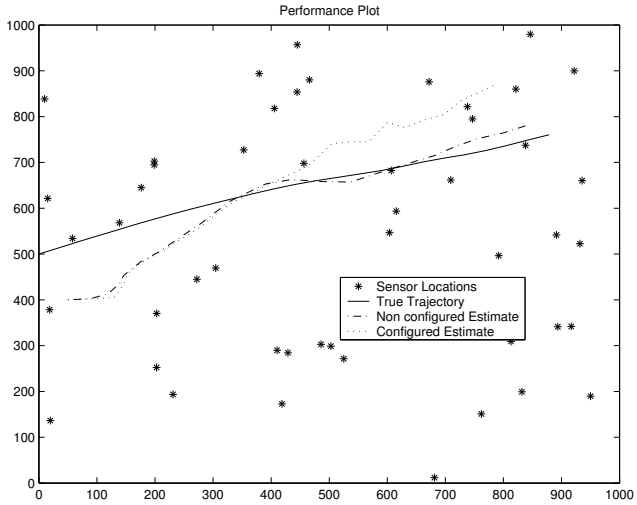

Figure 12: Sensor laydown whith true track and estimates for the case of configured and non-configured networks with false initial state

use. Some experimentation with fifty sensors showed visibly greater errors. Thus, the code provided in the index can be used to do a detailed study of the case with varying number of sensors. The theory and code can also be extended to the case of a three dimensional system.

In addition, the case of configured sensors can be optimized. For example, a strategy can be used to make sure that the radius r_k of the configured sensor network is such that it has at least one sensor actually detecting the target. Another possibility is to generalize the target model so as to incorporate non-linearities.

References

[1] T. Stevens and D. Morrell, "Minimization of sensor usage for target tracking in a network of irregularly spaced sensors," *IEEE Workshop on Statistical Signal Processing*, Sep. 2003.

[2] C. Champlin and D. Morrell, "Target tracking using irregularly spaced detectors and a continuous-state viterbi algorithm," *Technical Report, Arizona State University*, Nov. 2002.

[3] S. M. Kay, *Fundamentals of Statistical Signal Processing : Detection Theory*. Prentice-Hall Inc., 1998.

[4] N. G. M. Arulampalam, S. Maskell and T. Clapp, "A tutorial on particle fillters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, pp. 174–188, Feb. 2002.

[5] P. Nordlund and F. Gustafsson, "Sequential monte carlo filtering techniques applied to integrated navigation systems," *Proceedings of the American Control Conference*, pp. 4375–4380, June 2001.

A Matlab function for calculating probability of detection P_D

```
%=====================================================================
%
%   Author: Imtiaz Nizami
%   Course: Filtering of Stochastic Processes (EEE 581)
%
%   Purpose: To return the probability of detection
%            when the target is located at a particular
%            distance.
%
%   Arguments
%     Input(s):
%       d - scalar or vector of distances from target to sensor(s)
%       M - number of samples of recieved signal
%       beta - threshold normalized over noise variance
%       snr - signal to noise ratio when target and sensor are
%             at a distance of 1 unit
%     Output(s):
%       result - Probability of detection for each element of the
%                input distance vector 'd'
%
%   Assumptions:
%
%   Function Declaration:
%     function result = P_d (d, M, beta, snr)
%=====================================================================

function result = P_d (d, M, beta, snr)

arg1 = 1 + snr ./ d.^2;
arg1 = beta ./ ( 2 * arg1 );
arg2 = M / 2;
result = 1 - gammainc (arg1,arg2);

%plot(d,result,'b*')

% ============================================================
% Unused code
% ============================================================
%
```

```
% one line implementation of Pd
%  Pd(i) = 1 - gammainc( beta/( 2 * (1 + snr/d(i)^2) ), M/2 );k
% ============================================================
%
% Implementation of Pd given in paper.
% Do not work as the gamma and incomplete gamma function given
% in paper and matlab are not compatible.
%  Pd_den = gamma( M/2 );
%  if Pd_den == 0
%    error('Denomenator is zero')
%  end
%  arg1 = M/2;
%  arg2 = 1 + snr/d(i)^2;
%  arg2 = beta/2 * 1/arg2;
%  Pd_num = gammainc( arg1,arg2 );
%  Pd(i) = Pd_num / Pd_den;
% ============================================================
%
% Non-matrix impelentation of the code (code with loops instead
% of matrix algebra.
%  d_len = length(d);
%  for i = 1:d_len
%    arg1 = 1 + snr / d(i)^2;
%    arg1 = beta / ( 2 * arg1 );
%    arg2 = M / 2;
%    result(i) = 1 - gammainc (arg1,arg2);
%  end
```

B Matlab function that returns the obsrvation vector Z_k

```
%=====================================================================
%
%   Author: Imtiaz Nizami
%   Course: Filtering of Stochastic Processes (EEE 581)
%
%   Purpose: To generate the sequence of observations Z
%
%   Arguments
%     Input(s):
%       M - number of samples of recieved signal
%       d - distance between target and each sensor
%       mu - threshold to determine hypothesis, i.e.,
%            target present or absent
%     Input(s):
%       result - vector of observations where each element
%                of the vector is 1 (if energy > mu) or 0
%
%   Assumptions: vector 'd', which is an input vector should
%                have no value less than 0
%
%   Function declaration:
%     function result = Z_k(M,d,mu)
%
%=====================================================================

function result = Z_k(M,d,mu)

len_d = length(d);
noise_mean = zeros(1,M);
signal_mean = zeros(1,M);
mean_H1 = noise_mean + signal_mean;
noise_var = 10^(-4);

for i = 1:len_d;

  if d(i) > 0
    signal_var = 1/d(i)^2;
    var_H1 = noise_var + signal_var;
    rho = mean_H1 + randn(1,M) * sqrt(var_H1);
    energy = abs(rho * rho');
```

```
    if energy > mu
      result(i) = 1;
    else
      result(i) = 0;
    end

  elseif d(i) == 0
    result(i) = 1;
  else
    error('Distance from sensor to target is less than zero!!!');
  end

end
```

C Matlab function to return the distance vector between each sensor and the target

```
%=====================================================================
%
%   Author: Imtiaz Nizami
%   Course: Filtering of Stochastic Processes (EEE 581)
%
%   Purpose: To calculate the distance from sensors to target
%
%   Arguments
%     Input(s):
%       target_x - horizontal location of target
%       target_y - vertical location of target
%       sensor_x - vector of horizontal locations of all sensors
%       sensor_y - vector of vertical locations of all sensors
%     Output(s):
%       result - vector of distances from target to each sensor
%
%   Assumptions:
%     target_x - is a scalar
%     target_y - is a scalar
%     sensor_x - is a scalar or a vector with same length as sensor_y
%     sensor_y - is a scalar or a vector with same length as sensor_x
%
%   Function Declaration:
%     function result = distance(target_x,target_y,sensor_x,sensor_y)
%=====================================================================

function result = distance(target_x,target_y,sensor_x,sensor_y)

no_of_sensors = length(sensor_x);

if no_of_sensors ~= length(sensor_y)
  error('sensor_x and sensor_y should be same length vectors');
end

target_loc = target_x + i * target_y;
sensor_loc = sensor_x + i * sensor_y;

result = abs(sensor_loc - target_loc);
```

D Matlab code to generate state and observation sequences

```
%=========================================================================
%
%   Author: Imtiaz Nizami
%   Course: Filtering of Stochastic Processes (EEE 581)
%
%   Purpose: To generate state and observation sequence.
%            In addition, deal with sensor placement
%
%=========================================================================

clear;
tic

no_of_sensors = 49; % choose a number whose square root is an integer

% Particle filter parameters
N = 100; % number of particles
M = 20; % samples of signal recieved
beta = 40;
snr = 10^4;
mu = beta / snr; % threshold
%mu = 40.0 * 10^-4; % threshold
endK = 25; % final time index
niter = 200; % number of Monte Carlo iterations

% Location of the sensors
xy_max = 1000;
%sensors_each_row = floor(sqrt(no_of_sensors));
%sensors_avg_dist = xy_max / sensors_each_row;
%x_loc(1) = 0;
%for (kk = 2:sensors_each_row)
%  x_loc(kk) = x_loc(kk-1) + sensors_avg_dist;
%end
%x_loc = repmat(x_loc,sensors_each_row,1);
%y_loc = x_loc';
%x_loc = reshape(x_loc,1,sensors_each_row^2);
%y_loc = reshape(y_loc,1,sensors_each_row^2);

x_loc = xy_max * rand(1,no_of_sensors);
y_loc = xy_max * rand(1,no_of_sensors);
```

```
%figure(1);
%clf;
%plot(x_loc,y_loc,'b*');
%title('sensor locations');

dt = 2;
F = [1 0 dt 0; 0 1 0 dt; 0 0 1 0; 0 0 0 1];
qtrue = .1;
Q = qtrue*[dt^3/3 0 dt^2/2 0; 0 dt^3/3 0 dt^2/2; dt^2/2 0 dt 0; 0 dt^2/2 0 dt];
%Q = qtrue*[dt^3/3 0 0 0; 0 dt^3/3 0 0; 0 0 dt 0; 0 0 0 dt];
sqrtQ = sqrtm(Q);

iter = 0;
while (iter < niter)
  iter = iter+1;
  flag = 0;
  % Generate state and observation sequences
  XTrue(:,1) = [0; 500; 17; 7]; % Initial state
%  XTrue(:,1) = [0; 50; 4; 3]; % Initial state

  fprintf('\nIteration no. ')
  fprintf( int2str(iter) )
  fprintf('. #')

  for k=1:endK
    XTrue(:,k+1) = F*XTrue(:,k) + sqrtQ*randn(4,1);
    fprintf('#')
    if ( abs( XTrue(1,k+1) ) > xy_max | abs( XTrue(2,k+1) ) > xy_max )
      flag = 1;
      iter = iter-1;
      fprintf('\nThis iteration is void. Regenerating data...')
      break
    end
    distance_vector(:,k) = distance( XTrue(1,k+1),XTrue(2,k+1),x_loc,y_loc)';
    Z(:,k) = Z_k(M,distance_vector(:,k),mu)';
%    plot(distance_vector(:,k),Z(:,k),'b*')
%    pause
  end

  if (flag == 0)
    fprintf('. Estimated time left: ')
```

```
    est_time = ( toc * (niter - iter) / iter ) / 60;
    fprintf( num2str(est_time,2) )
    fprintf(' minutes(s)\n')

    saved_Z(iter,:,:) = Z;
    saved_XTrue(iter,:,:) = XTrue;
  end
%figure(2);
%clf;
%plot(x_loc,y_loc,'b*',XTrue(1,:),XTrue(2,:),'r-');
%%plot(squeeze(saved_XTrue(1,1,:)),squeeze(saved_XTrue(1,2,:)),'r-');
%title('True trajectory');
%  pause
end

%figure(2);
%clf;
%plot(x_loc,y_loc,'b*');
%  plot(x_loc,y_loc,'b*',XTrue(1,:),XTrue(2,:),'r-');
%plot(x_loc,y_loc,'b*',squeeze(saved_XTrue(1,1,:)),squeeze(saved_XTrue(1,2,:)),'r-');
%title('True trajectory');
%  pause

save saved_parameters
```

E Matlab code to implement particle filter

```
%=========================================================================
%
%   Author: Imtiaz Nizami
%   Course: Filtering of Stochastic Processes (EEE 581)
%
%   Purpose: To implement particle filter for the estimation
%            of a target moving in a sensor network. Includes
%            estimate based on a all available sensors and
%            reduced set of sensors.
%            prefix 'red' denotes reduced sensor parameter
%            in the variables declared below.
%
%=========================================================================
clear;

% Load a data file which is generated by running the script
% named "generate_state_and_observation.m"
load saved_parameters

tic % Start timer

N=100;      % Total number of particles (temporary)
niter = 200;  % Total number of Monte Carlo iterations (temporary)

ErrSum = zeros(4,endK+1);
SqErrSum = zeros(4,endK+1);

red_ErrSum = zeros(4,endK+1);
red_SqErrSum = zeros(4,endK+1);

% Parameters for keeping track of reduced set of sensors
C_r_k = 1;
r_k = 500;

avg_sensors = zeros(1,niter);

for iter = 1:niter
  XTrue = squeeze( saved_XTrue(iter,:,:) ); % Retreiving values generated
  Z = squeeze( saved_Z(iter,:,:) );         % in data generation script
```

```
  % Particle filter implementation
  % Generate the initial particles

%  init_vector = [200;300;15;8];
  init_vector = XTrue(:,1);
  X = repmat(init_vector,1,N) + ...
        sqrtm([1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]) * randn(4,N);
%  X = repmat(init_vector,1,N) + ...
%        sqrtm([XTrue(2,1) 0 0 0; 0 XTrue(2,1) 0 0; 0 0 XTrue(3,1) 0; 0 0 0 XTrue(3,1)]) * r

  red_X = X;
  w = ones(1,N)/N;
  red_w = w;
  xhat(:,1) = X*w';
  red_xhat(:,1) = xhat(:,1);

  % Plot the initial particles
%  figure(21);
%  clf;
%  stem(X(2,:),w(1,:));
%  title('Initial Particles - x position');
%  pause

  for k=1:endK
    rand_matrix = randn(4,N);
    xExt = F*X + sqrtQ*rand_matrix;
    red_xExt = F*red_X + sqrtQ*rand_matrix;

    [red_no_of_sensors,red_Z,red_x_loc,red_y_loc] = reduced_sensor_position( red_xhat(1,k),r
%    [zero_one,red_Z,red_x_loc,red_y_loc] = one_sensor_position( red_xhat(1,k),red_xhat(2,k

    avg_sensors(1,iter) = avg_sensors(1,iter) + red_no_of_sensors;

    for n = 1:N
      dist = distance( xExt(1,n),xExt(2,n),x_loc,y_loc)';
      P_det = P_d(dist,M,beta,snr);
      exponent = Z(1,k);
      P_Zk_Xk = P_det(1)^exponent * ( 1 - P_det(1) )^(1-exponent);

      if red_no_of_sensors ~= 0
        red_dist = distance( red_xExt(1,n),red_xExt(2,n),red_x_loc,red_y_loc)';
        red_P_det = P_d(red_dist,M,beta,snr);
```

```
    red_exponent = red_Z(1);
    red_P_Zk_Xk = red_P_det(1)^red_exponent * ( 1 - red_P_det(1) )^(1-red_exponent);
  end

  for l=2:no_of_sensors
    exponent = Z(l,k);
    P_Zk_Xk = P_Zk_Xk * ( P_det(l)^exponent * ( 1 - P_det(l) )^(1-exponent) );
    if (l <= red_no_of_sensors)
      red_exponent = red_Z(l);
      red_P_Zk_Xk = red_P_Zk_Xk * ( red_P_det(l)^red_exponent * ( 1 - red_P_det(l) )^(1
    end
  end
  w(n) = P_Zk_Xk;

  if red_no_of_sensors ~= 0
    red_w(n) = red_P_Zk_Xk;
  else
    red_w(n) = 1/N;
  end

end

if sum(w) > 0
  w = w/sum(w);
else
  w = ones(1,N)/N;
end

if sum(red_w) > 0
  red_w = red_w/sum(red_w);
else
  red_w = ones(1,N)/N;
end

% Compute the estimate of the state
xhat(:,k+1) = xExt * w';
red_xhat(:,k+1) = red_xExt * red_w';

% Finding the covariance matrix
P_k_k = red_xExt - repmat( red_xhat(:,k+1),1,N );
P_k_k = ( repmat(red_w,4,1) .* P_k_k ) * P_k_k';
e_k = P_k_k(1,1) + P_k_k(2,2);
```

```
    r_k = C_r_k * e_k;
%    fprintf('\nr_k: ')
%    fprintf(num2str(r_k))
%    fprintf('\te_k: ')
%    fprintf(num2str(e_k))
%    fprintf('\n')


    % Plot the reweighted particles
%    figure(25);
%    clf;
%    stem(xExt(1,:),w(1,:));
%    title('Updated Particles - x position');
%    pause

    % Simulation
%    figure(71)
%     clf
%    plot(xExt(1,:),xExt(2,:),'g.',XTrue(1,k),XTrue(2,k),'r*')
%    axis([0,1000,0,1000])
%    pause(.1)


    % Resample the particles
    len = length(w);

    % Cumulative Distributive Function
    cumpr = cumsum(w(1,1:len))';
    cumpr = cumpr/max(cumpr);
    red_cumpr = cumsum(red_w(1,1:len))';
    red_cumpr = red_cumpr/max(red_cumpr);

    rand_no = rand(1,1);
    u(1,1) = (1/N)*rand_no;
    a=1;
    for b = 1:N
      u(b,1)= u(1,1) + (1/N)*(b-1);
      while (u(b,1) > cumpr(a,1))
        a = a+1;
        if a > N
          break
        end
```

```
      end
      if a <= N
        x_update(:,b) = xExt(:,a);
      end
    end

    X = x_update;
    w = 1/N*ones(1,N);

    red_u(1,1) = (1/N)*rand_no;
    red_a=1;
    for red_b = 1:N
      red_u(red_b,1)= red_u(1,1) + (1/N)*(red_b-1);
      while (red_u(red_b,1) > red_cumpr(red_a,1))
        red_a = red_a+1;
        if red_a > N
          break
        end
      end
      if red_a <= N
        red_x_update(:,red_b) = red_xExt(:,red_a);
      end
    end

    red_X = red_x_update;
    red_w = w;
  end

  avg_sensors(1,iter) = avg_sensors(1,iter) / endK;

  ErrSum = ErrSum + (XTrue-xhat);
  SqErrSum = SqErrSum + (XTrue-xhat).^2;

  red_ErrSum = red_ErrSum + (XTrue-red_xhat);
  red_SqErrSum = red_SqErrSum + (XTrue-red_xhat).^2;

  fprintf(' Estimated time left: ')
  est_time = ( toc * (niter - iter) / iter ) / 60;
  fprintf( num2str(est_time,4) )
  fprintf(' minutes(s)\n')
end
p = SqErrSum/niter-ErrSum.^2/niter^2;
```

```
red_p = red_SqErrSum/niter-red_ErrSum.^2/niter^2;


figure(1);
clf;
subplot(2,1,1);
plot(0:endK,p(1,:),'b--');
title('Average X Position Error');
subplot(2,1,2);
plot(0:endK,p(2,:),'b--');
title('Average Y Position Error');

figure(2);
clf;
subplot(2,1,1);
plot(0:endK,red_p(1,:),'b--');
title('Average X Position Error - reduced sensors');
subplot(2,1,2);
plot(0:endK,red_p(2,:),'b--');
title('Average Y Position Error - reduced sensors');


figure(3);
clf;
subplot(2,1,1);
plot(0:endK,p(3,:),'b--');
title('Average X Velocity Error');
subplot(2,1,2);
plot(0:endK,p(4,:),'b--');
title('Average Y Velocity Error');


figure(4);
clf;
subplot(2,1,1);
plot(0:endK,red_p(3,:),'b--');
title('Average X Velocity Error - reduced sensors');
subplot(2,1,2);
plot(0:endK,red_p(4,:),'b--');
title('Average Y Velocity Error - reduced sensors');
```

```
figure(5);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(1,:),'b--',0:endK,xhat(1,:),'r-');
title('Estimated x position sequence');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(2,:),'b--',0:endK,xhat(2,:),'r-');
title('Estimated y position sequence');
legend('True','Estimated');


figure(6);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(1,:),'b--',0:endK,red_xhat(1,:),'r-');
title('Estimated x position sequence - reduced sensors');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(2,:),'b--',0:endK,red_xhat(2,:),'r-');
title('Estimated y position sequence - reduced sensors');
legend('True','Estimated');


figure(7);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(3,:),'b--',0:endK,xhat(3,:),'r-');
title('Estimated x velocity sequence');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(4,:),'b--',0:endK,xhat(4,:),'r-');
title('Estimated y velocity sequence');
legend('True','Estimated');

figure(8);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(3,:),'b--',0:endK,red_xhat(3,:),'r-');
```

```
title('Estimated x velocity sequence - reduced sensors');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(4,:),'b--',0:endK,red_xhat(4,:),'r-');
title('Estimated y velocity sequenc - reduced sensorse');
legend('True','Estimated');


figure(9);
clf;
plot(x_loc,y_loc,'r*',XTrue(1,:),XTrue(2,:),'b--',xhat(1,:),xhat(2,:),'g-.',red_xhat(1,:),r
title('Performance Plot');
legend('Sensor Locations','True Trajectory','Non configured Estimate','Configured Estimate'

fprintf('\nAverage number of sensors - reduced sensor case: ')
fprintf( num2str(sum(avg_sensors)/(niter)) )


total_time = toc/60;
fprintf('\nTotal simulation time: ')
fprintf( num2str(total_time) )
fprintf(' minute(s)\n')
```

F Matlab code to implement Rao-Blackwellized filter

```
%=========================================================================
%
%   Author: Imtiaz Nizami
%   Course: Filtering of Stochastic Processes (EEE 581)
%
%   Purpose: To implement Rao Blackwellized filter for the estimation
%            of a target moving in a sensor network.
%            This is done with a fixed number of sensors and a
%            technique to update number of sensors to minimize
%            power.
%
%=========================================================================
clear;

% Load a data file which is generated by running the script
% named "generate_state_and_observation.m"
load saved_parameters

tic % Start timer

N=300;      % Total number of particles (temporary)
niter = 2;  % Total number of Monte Carlo iterations (temporary)

ErrSum = zeros(4,endK+1);
SqErrSum = zeros(4,endK+1);

red_ErrSum = zeros(4,endK+1);
red_SqErrSum = zeros(4,endK+1);

% Parameters for keeping track of reduced set of sensors
C_r_k = 1;
r_k = 500;

avg_sensors = zeros(1,niter);

% For the RB filter, we break F into several submatrices
Fpf = F(1:2,1:2); % Upper left corner
Fpf_a = F(1:2,3:4); % Upper right corner
Fa = F(3:4,3:4); % Lower right corner
```

```
% For the RB filter, we break Q into components
Qpf = Q(1:2,1:2);
U = Q(1:2,3:4);
Qa = Q(3:4,3:4);

% Initial covariance of velocity estimate
Pk0 = [XTrue(3,1) 0; 0 XTrue(3,1)];

for iter = 1:niter
  XTrue = squeeze( saved_XTrue(iter,:,:) );
  Z = squeeze( saved_Z(iter,:,:) );

  % Filter implementation
  init_vector = [XTrue(1,1);XTrue(2,1)];
%  X = repmat(init_vector,1,N) + sqrtm([XTrue(2,1) 0; 0 XTrue(2,1)]) * randn(2,N);
  X = repmat(init_vector,1,N) + sqrtm([1 0; 0 1]) * randn(2,N);
%  xi = repmat([1;0],1,N);
  xi = repmat([XTrue(3,1);0],1,N);
  C = Pk0;

  w = ones(1,N)/N;
  xhat(:,1) = [X;xi]*w';

  red_X = X;
  red_xi = xi;
  red_w = w;
  red_xhat(:,1) = xhat(:,1);
  red_C = C;

  % Plot the initial particles
%  figure(21);
%  clf;
%  stem(X(2,:),w(1,:));
%  title('Initial Particles - x position');
%  pause

  for k=1:endK
    % Sample using the velocity estimate for each particle
    S = Fpf_a*C*Fpf_a' + Qpf;
    sqrtS = sqrtm(S);

    red_S = Fpf_a*red_C*Fpf_a' + Qpf;
```

```
red_sqrtS = sqrtS;

rand_matrix = randn(2,N);
xExt = X + Fpf_a*xi + sqrtS*rand_matrix;
red_xExt = red_X + Fpf_a*red_xi + red_sqrtS*rand_matrix;

[red_no_of_sensors,red_Z,red_x_loc,red_y_loc] = reduced_sensor_position( red_xhat(1,k),

avg_sensors(1,iter) = avg_sensors(1,iter) + red_no_of_sensors;

for n = 1:N
  dist = distance( xExt(1,n),xExt(2,n),x_loc,y_loc)';
  P_det = P_d(dist,M,beta,snr);
  exponent = Z(1,k);
  P_Zk_Xk = P_det(1)^exponent * ( 1 - P_det(1) )^(1-exponent);

  if red_no_of_sensors ~= 0
    red_dist = distance( red_xExt(1,n),red_xExt(2,n),red_x_loc,red_y_loc)';
    red_P_det = P_d(red_dist,M,beta,snr);
    red_exponent = red_Z(1);
    red_P_Zk_Xk = red_P_det(1)^red_exponent * ( 1 - red_P_det(1) )^(1-red_exponent);
  end

  for l=2:no_of_sensors
    exponent = Z(l,k);
    P_Zk_Xk = P_Zk_Xk * ( P_det(l)^exponent * ( 1 - P_det(l) )^(1-exponent) );
    if (l <= red_no_of_sensors)
      red_exponent = red_Z(l);
      red_P_Zk_Xk = red_P_Zk_Xk * ( red_P_det(l)^red_exponent * ( 1 - red_P_det(l) )^(1
    end
  end
  w(n) = P_Zk_Xk;

  if red_no_of_sensors ~= 0
    red_w(n) = red_P_Zk_Xk;
  else
    red_w(n) = 1/N;
  end

end

if sum(w) > 0
```

```
      w = w/sum(w);
    else
      w = ones(1,N)/N;
    end

    if sum(red_w) > 0
      red_w = red_w/sum(red_w);
    else
      red_w = ones(1,N)/N;
    end

    % Compute the estimate of the state
    xhat(:,k+1) = [xExt;xi] * w';
    red_xhat(:,k+1) = [red_xExt;red_xi] * red_w';

    % Finding the covariance matrix
    P_k_k = [red_xExt;red_xi] - repmat( red_xhat(:,k+1),1,N );
    P_k_k = ( repmat(red_w,4,1) .* P_k_k ) * P_k_k';
    e_k = P_k_k(1,1) + P_k_k(2,2);
    r_k = C_r_k * e_k;
%    fprintf('\nr_k: ')
%    fprintf(num2str(r_k))
%    fprintf('\te_k: ')
%    fprintf(num2str(e_k))
%    fprintf('\n')

    % Filter step for the velocities
    Zi = xExt - Fpf*X;
    KGain = C*Fpf_a'*inv(S);
    D = C - KGain*Fpf_a*C;
    rho = xi + KGain*(Zi - Fpf_a*xi);

    red_Zi = red_xExt - Fpf*red_X;
    red_KGain = red_C*Fpf_a'*inv(red_S);
    red_D = red_C - red_KGain*Fpf_a*red_C;
    red_rho = red_xi + red_KGain*(red_Zi - Fpf_a*red_xi);

    % Predictor step for the velocities
    PredGain = Fa - U'*inv(Qpf)*Fpf_a;
    C = PredGain*D*PredGain' + Qa-U'*inv(Qpf)*U;
    xi = PredGain*rho + U'*inv(Qpf)*Zi;
```

```
    red_PredGain = Fa - U'*inv(Qpf)*Fpf_a;
    red_C = red_PredGain*red_D*red_PredGain' + Qa-U'*inv(Qpf)*U;
    red_xi = red_PredGain*red_rho + U'*inv(Qpf)*red_Zi;

    % Plot the reweighted particles
%    figure(25);
%    clf;
%    stem(xExt(1,:),w(1,:));
%    title('Updated Particles - x position');
%    pause

    % Simulation
%    figure(71)
%    clf
%
%    plot(xExt(1,:),xExt(2,:),'g.',XTrue(1,k),XTrue(2,k),'r*')
%    axis([0,1000,0,1000])
%    pause(.1)

    % Resample the particles
    len = length(w);

    % Cumulative Distributive Function
    cumpr = cumsum(w(1,1:len))';
    cumpr = cumpr/max(cumpr);
    red_cumpr = cumsum(red_w(1,1:len))';
    red_cumpr = red_cumpr/max(red_cumpr);

    rand_no = rand(1,1);
    u(1,1) = (1/N)*rand_no;
    a=1;
    for b = 1:N
      u(b,1)= u(1,1) + (1/N)*(b-1);
      while (u(b,1) > cumpr(a,1))
        a = a+1;
        if a > N
          break
        end
      end
      if a <= N
        x_update(:,b) = xExt(:,a);
        xi_update(:,b) = xi(:,a);
```

```
      end
    end

    X = x_update;
    xi = xi_update;
    w = 1/N*ones(1,N);

    red_u(1,1) = (1/N)*rand_no;
    red_a=1;
    for red_b = 1:N
      red_u(red_b,1)= red_u(1,1) + (1/N)*(red_b-1);
      while (red_u(red_b,1) > red_cumpr(red_a,1))
        red_a = red_a+1;
        if red_a > N
          break
        end
      end
      if red_a <= N
        red_x_update(:,red_b) = red_xExt(:,red_a);
        red_xi_update(:,red_b) = red_xi(:,red_a);
      end
    end

    red_X = red_x_update;
    red_xi = red_xi_update;
    red_w = w;
  end

  avg_sensors(1,iter) = avg_sensors(1,iter) / endK;

  ErrSum = ErrSum + (XTrue-xhat);
  SqErrSum = SqErrSum + (XTrue-xhat).^2;

  red_ErrSum = red_ErrSum + (XTrue-red_xhat);
  red_SqErrSum = red_SqErrSum + (XTrue-red_xhat).^2;

  fprintf(' Estimated time left: ')
  est_time = ( toc * (niter - iter) / iter ) / 60;
  fprintf( num2str(est_time,4) )
  fprintf(' minutes(s)\n')
end
p = SqErrSum/niter-ErrSum.^2/niter^2;
```

```
red_p = red_SqErrSum/niter-red_ErrSum.^2/niter^2;


figure(1);
clf;
subplot(2,1,1);
plot(0:endK,p(1,:),'b--');
title('Average X Position Error');
subplot(2,1,2);
plot(0:endK,p(2,:),'b--');
title('Average Y Position Error');

figure(2);
clf;
subplot(2,1,1);
plot(0:endK,red_p(1,:),'b--');
title('Average X Position Error - reduced sensors');
subplot(2,1,2);
plot(0:endK,red_p(2,:),'b--');
title('Average Y Position Error - reduced sensors');


figure(3);
clf;
subplot(2,1,1);
plot(0:endK,p(3,:),'b--');
title('Average X Velocity Error');
subplot(2,1,2);
plot(0:endK,p(4,:),'b--');
title('Average Y Velocity Error');


figure(4);
clf;
subplot(2,1,1);
plot(0:endK,red_p(3,:),'b--');
title('Average X Velocity Error - reduced sensors');
subplot(2,1,2);
plot(0:endK,red_p(4,:),'b--');
title('Average Y Velocity Error - reduced sensors');
```

```
figure(5);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(1,:),'b--',0:endK,xhat(1,:),'r-');
title('Estimated x position sequence');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(2,:),'b--',0:endK,xhat(2,:),'r-');
title('Estimated y position sequence');
legend('True','Estimated');


figure(6);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(1,:),'b--',0:endK,red_xhat(1,:),'r-');
title('Estimated x position sequence - reduced sensors');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(2,:),'b--',0:endK,red_xhat(2,:),'r-');
title('Estimated y position sequence - reduced sensors');
legend('True','Estimated');


figure(7);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(3,:),'b--',0:endK,xhat(3,:),'r-');
title('Estimated x velocity sequence');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(4,:),'b--',0:endK,xhat(4,:),'r-');
title('Estimated y velocity sequence');
legend('True','Estimated');

figure(8);
clf;
subplot(2,1,1);
plot(0:endK,XTrue(3,:),'b--',0:endK,red_xhat(3,:),'r-');
```

```
title('Estimated x velocity sequence - reduced sensors');
legend('True','Estimated');

subplot(2,1,2);
plot(0:endK,XTrue(4,:),'b--',0:endK,red_xhat(4,:),'r-');
title('Estimated y velocity sequenc - reduced sensorse');
legend('True','Estimated');


figure(9);
clf;
plot(x_loc,y_loc,'r*',XTrue(1,:),XTrue(2,:),'b--',xhat(1,:),xhat(2,:),'g-.',red_xhat(1,:),r
title('Performance Plot');
legend('Sensor Locations','True Trajectory','Non configured Estimate','Configured Estimate'

fprintf('\nAverage number of sensors - reduced sensor case: ')
fprintf( num2str(sum(avg_sensors)/(niter)) )


total_time = toc/60;
fprintf('\nTotal simulation time: ')
fprintf( num2str(total_time) )
fprintf(' minute(s)\n')
```