EM ALGORITHMS FOR OPTICAL POSITION SENSING

Myles Black-Ingersoll, David Lavy and Joshua Rapp

May 2, 2016

Boston University

Department of Electrical and Computer Engineering

Technical Report No. ECE-YYYY-NN

BOSTON UNIVERSITY

EM ALGORITHMS FOR OPTICAL POSITION SENSING

Myles Black-Ingersoll, David Lavy and Joshua Rapp

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
`www.bu.edu/ece`

May 2, 2016

Technical Report No. ECE-YYYY-NN

Contents

List of Figures

1 Introduction

Optical position sensing is a common task in calibration and tracking procedures. In applications as diverse as star tracking [1], satellite navigation [2], and medical x-ray localization [3], the exact position of a source signal captured by a detector array is needed to properly align a target. For many signals, such as those from weak stars or rapidly moving satellites, the resulting accumulated photon flux is low. In the presence of significant ambient noise and detector dark counts, the low signal-to-noise ratio (SNR) makes position estimation a difficult problem [2].

In this report, we investigate the Expectation-Maximization algorithm as a tool for estimating signal positions on a two-dimensional detector when detection counts are low. We evaluate performance at different noise levels compared to other standard approaches and test the effects of removing certain classical assumptions from the solution. Finally, we expand from the case of estimating a beam position at a single static frame to the practical challenge of tracking multiple moving targets.

2 Literature Review

For the problem of identifying the arrival time of a one-dimensional waveform in additive white Gaussian noise, the optimal solution is given by a correlator or matched filter [4]. This filter has an impulse response equal to the time-reversed input waveform. The problem of finding the position of a signal on a two-dimensional detector is analogous to the 1D problem in time [5], so a matched filter would seem to be a simple solution to the optical position sensing problem. Unfortunately, both 1D and 2D formulations require knowledge of the pulse shape and input SNR [6], which are parameters that may not always be known. Furthermore, the additive white Gaussian model of noise is a poor fit for the low-flux photodetection case that we consider. Instead, we aim to use an adaptable solution that can be broadly applied to many contexts without too many *a priori* assumptions. The EM algorithm is one such solution and will be thoroughly explored in the following report.

3 Problem Statement

3.1 Photodetection Model

A laser beam orthogonal to the detector surface has an intensity that can be described by a circularly symmetric bivariate Gaussian distribution:

$$I(\mathbf{x};\boldsymbol{\mu}) = \frac{1}{2\pi\rho^2}\exp\left\{-\frac{(\mathbf{x}-\boldsymbol{\mu})^\top(\mathbf{x}-\boldsymbol{\mu})}{2\rho^2}\right\}, \tag{1}$$

where $\mathbf{x} = [x_1,\ x_2]^\top$ describes any point on the detector, $\boldsymbol{\mu} = [\mu_1,\ \mu_2]^\top$ denotes the beam's center, and ρ characterizes the beam's width [7].

Measurements at the photodetector are characterized by an inhomogeneous mixed Poisson process with intensity

$$\lambda(\mathbf{x}, t) = \lambda_s(\mathbf{x}, t) + \lambda_n(\mathbf{x}, t). \tag{2}$$

The first component is an inhomogeneous signal process due to the incident beam with intensity $\lambda_s(x, t) = \Lambda_s I(\mathbf{x}; \boldsymbol{\mu})$. The second component is a homogeneous noise process with intensity $\lambda_n = \frac{\Lambda_n}{\|A\|}$, where A is the detector area. This noise is a combination of dark counts—false registrations of detections due to non-ideal detector properties—and photons from ambient light, both of which are uniformly distributed over the detector's surface and can thus be described by a single term.

3.2 Position Estimation of a Single Static Beam

In the ideal case where $\lambda_n = 0$, all detections are due to the signal source, so the maximum likelihood position estimate $\hat{\boldsymbol{\mu}}_{ML}$ is simply the centroid of the data [7]. Unfortunately, in any practical setting, $\lambda_n > 0$, so there is no closed-form solution for $\hat{\boldsymbol{\mu}}_{ML}$. Instead, numerical methods are needed to form a beam position estimate.

Expectation-Maximization One common numerical method for parameter estimation when closed-form solutions are not available is the Expectation-Maximization (EM) algorithm introduced in [8]. The basic approach of EM is to view experimental observations X as being *incomplete data*. The missing components of the experiment are called *latent* or *hidden* variables Y, which are random and only indirectly observed through X. Together, X and Y combine to form the *complete data*. Since many possible latent variable formulations are possible for the same observed data, the challenge of EM is in properly describing the complete data so that maximum likelihood analysis is computationally tractable [7].

Given an incomplete data set X with unknown data Y and parameters θ, the EM algorithm follows these basic steps to estimate θ:

1. Determine the log-likelihood function $\log\left(p\left(x, y|\theta\right)\right)$ of the complete data set.

2. Take the conditional expectation of the log-likelihood given X and the current θ^t estimate:
$$E_{Y|X,\theta^t}[\log(p(x, y|\theta^{t+1}))|x, \theta^t]$$

3. Maximize the expectation in step 2 to determine the updated parameter estimate θ^{t+1}:
$$\underset{\theta^{t+1}}{\arg\max}\, E_{Y|X,\theta^t}[\log(p(x, y|\theta^{t+1}))|x, \theta^t]$$

4. Repeat steps 2 and 3 until the parameter estimate has converged.

The benefit of using this formulation is that the incomplete data log-likelihood is guaranteed to converge to a limiting value, although there is no general guarantee of that value being the global maximum [7] [8].

Applying the EM Algorithm to Optical Position Sensing In the case of optical position sensing, the parameter to be estimated is the center $\hat{\boldsymbol{\mu}}_s$ of an optical beam on a 2D grid. The incomplete data are the locations $\{\mathbf{x}_i\},\ i = 1 \ldots N$ of the N photon detections. The hidden data $m_i \in \{s,\ n\}$ are marks denoting the detection sources as either signal or noise. Combined, we have the complete data described as $\{(\mathbf{x}_i,\ m_i)\},\ i = 1 \ldots N$. The key intuition is that, given the marks for each detection, the optimal estimate would use only the detections due to signal. However, since we observe only the incomplete data, we must estimate the marks along with the beam position.

From [1], the incomplete data log-likelihood function is given as

$$\mathcal{L}(\boldsymbol{\mu}_s) = -\int_A \lambda_s(\mathbf{x}) d\mathbf{x} - \int_A \lambda_n(\mathbf{x}) d\mathbf{x} + \int_A \log[\lambda_s(\mathbf{x}) + \lambda_n(\mathbf{x})] N d\mathbf{x}, \tag{3}$$

which is intractable unless $\lambda_n = 0$. However, reframing parameter estimation for the complete data model, the expression simplifies as

$$\begin{aligned}\mathcal{L}_{cd}(\mathbf{x}) &= \mathcal{L}_s(\mathbf{x}) + \mathcal{L}_n(\mathbf{x}) \\ &= -\int_A \lambda_s(\mathbf{x}) d\mathbf{x} + \int_A \log(\lambda_s(\mathbf{x})) N d\mathbf{x}(m_s) \end{aligned} \tag{4}$$

since the noise component provides no information about the beam position, so detections due to noise can be ignored.

The *Expectation Step* of the EM algorithm yields weight terms for each point, representing the probability that a detection is due to signal:

$$w(\mathbf{x}, \hat{\boldsymbol{\mu}}_s^{(t)}) = \frac{\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t)})}{\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t)}) + \lambda_n} \tag{5}$$

$$E[\mathcal{L}_{cd}(\mathbf{x}) | \{m\}_{i=1}^N, \hat{\boldsymbol{\mu}}^{(t)}] = -\int_A \lambda_s(\mathbf{x}) d\mathbf{x} + \int_A w(\mathbf{x}, \hat{\boldsymbol{\mu}}_s^{(t)}) \log(\lambda_s(\mathbf{x})) N d\mathbf{x}.$$

The *Maximization Step* of the EM algorithm varies depending on the beam shape. In this instance, the beam is assumed to be circularly symmetric and Gaussian, so setting the gradient of (3.2) to zero yields

$$\int_A w(\mathbf{x}, \hat{\boldsymbol{\mu}}_s^{(t)}) \boldsymbol{\Sigma}_s^{-1} (\mathbf{x} - \boldsymbol{\mu}) N d\mathbf{x} = 0 \tag{6}$$

$$\hat{\boldsymbol{\mu}}^{(t+1)} = \frac{\sum_{i=1}^N w_i(\mathbf{x}_i, \hat{\boldsymbol{\mu}}^{(t)}) \mathbf{x}_i}{\sum_{i=1}^N w_i(\mathbf{x}_i, \hat{\boldsymbol{\mu}}^{(t)})}. \tag{7}$$

3.3 Position Tracking of a Single Moving Beam

Including a Prior for the Beam Position Once the position has been estimated for the static case using the EM Algorithm, we analyze a moving beam and focus on

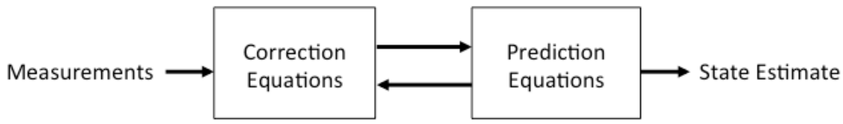


Figure 1: Kalman filter cycle

target tracking. This problem can be thought of as applying the EM algorithm to multiple frames sampled as the beam traverses some unknown path. We assume the path is continuous, not a series of random jumps. Therefore we can include a prior on the beam position for all but the estimate for the first frame. For a given frame, we assume that the prior is a circularly symmetric Gaussian centered at the final position estimate of the previous frame [1]. The standard deviation, σ of the Gaussian can be estimated by taking the average of the distances between the previous $\min\{M, N_f - 1\}$ frames' final position estimates (where M is the maximum number of frames to use and N_f is the current frame number) and multiplying that average by a regularization parameter.

Applying Kalman Filtering for beam tracking Since applying EM at each frame will cause jumps due to noise changing from one frame to the next, we apply a linear Kalman filter to smooth out beam tracking. The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean-squared error [9]. It is currently used for many different applications including filtering noisy signals, guidance, navigation and control of vehicles. The filter addresses the problem of estimating the state x_k of a discrete-time controlled process, which is governed by a dynamic model that relates the previous state at time $k-1$ with the current state, denoted as:

$$x_k = Ax_{k-1} + Bu_k + w_{k-1},$$

where A is the state transition model which is applied to the previous state x_{k-1} and B is the control-input model applied to the control vector u_k. A measurement model which relates the current state to the measurement z_k is denoted as

$$z_k = Hx_k + v_k,$$

where H is the observation model, which maps the true state space into the observation space. The variables w_k and v_k are the process and measurement noise respectively and they are assumed to be independent white Gaussian noise:

$$p(w) \sim N(0, Q), \ p(v) \sim N(0, R).$$

The two covariance matrices Q and R are assumed to be constant. Finally, the Kalman Filter cycle is divided in two steps: a prediction and a correction step. The first step predicts the current state based on the previous state and the commanded action. The second step observes the measurements, and judges whether they are reliable based on the overall state estimate. This filter works by predicting the current state using the prediction equations, followed by checking the quality of the prediction using the update equations. This process is repeated continuously to update the current state. Figure 1 shows a block diagram of the procedure.

The specific equations for the time and measurement updates are presented below:

Time update (Prediction)	**Measurement update (Correction)**
$x_k^- = Ax_{k-1} + Bu_k$ $P_k^- = AP_{k-1}A^T + Q$	$K_k = P_k^- H^T(HP_K^- H^T + R)^{-1}$ $x_k = x_k^- + K_k(z_k - Hx_k^-)$ $P_k = (I - K_k H)P_k^-$

The time update equations project the state and covariance estimates forward from time step $k-1$ to step k. The measurement update equations first compute the Kalman gain K_k. After this step we incorporate our measurement z_k and generate an *a posteriori* state estimate. The final step is to obtain an *a posteriori* error covariance estimate. After each time and measurement update pair, the process repeats and uses the previous *a posteriori* estimates and predicts new *a posteriori* estimates.

3.4 Position Estimation of Multiple Static Beams

The above EM algorithm was designed for a single position estimate, so identifying the location of multiple beams requires some pre-processing. Given k beams, where the number of beams is known, we want to partition the set of photoevents into clusters of signal photoevents corresponding to each beam. If the data is separated into different beam clusters, the EM algorithm can be applied to each cluster separately providing estimates of each position.

Ideal clusters only have signal photoevents from a single beam along with some noise photoevents. The worst case scenario is when clusters are entirely due to noise. The EM algorithm cannot provide an accurate beam position prediction if it is only provided with noise photoevents. For this reason we chose to use the k-medians algorithm as our multi-beam clustering algorithm. It is simple to implement, and unlike the k-means algorithm it does not include a squared term that heavily penalizes outlying noise photoevents.

3.5 Position Tracking of Multiple Moving Beams

Given the current estimated positions for the position of the beams, we now move onto tracking these beams. The main problem is that several beams are detected but they have no identifying characteristics, making them difficult to track. Using the Kalman filter as before for state estimation, we now add an assignment step to match the detected positions to associated beams by the Hungarian algorithm [10]. By solving the assignment problem, it is then possible to do the correction step, as every beam will have its correct prediction. We set a threshold on the maximum distance between the predicted and observed state, so in the case that the beam gets lost or incorrectly mismatched, the system will just predict the position of the beam up to a certain number of frames until it decides that the beam is no longer present.

4 Implementation

4.1 Data Generation

Unfortunately, no real photodetection data was available for testing, so the only available results are based on simulated data sets. One upside of this approach, however, is that performance can be evaluated quantitatively, since the true beam positions are known for simulated data. Data sets were generated based on the model introduced in [1]. Given the detector size (i.e., rows and columns of detector matrix, yielding area A), the beam size (given by the standard deviation ρ) and the signal and noise photoconversion rates (Λ_s, Λ_n), a matrix of detections was generated. First, a Poisson random number generator determined the number of noise detections N_n and signal detections N_s by

$$N_n \sim \text{Poisson}(\Lambda_n),\ \ N_s \sim \text{Poisson}(\Lambda_s).$$

The true signal position $\boldsymbol{\mu}$ was chosen uniformly at random on the detector, with some constraints to avoid a position too close to the detector edge so that detections all land on the detector surface. N_n noise detections were generated uniformly at random over the entire detector surface; N_s signal detections were generated from a circularly symmetric bivariate Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \rho^2 I_2)$ and were rounded to the nearest detector element. Slight modifications were made for the cases of a beam with an unknown shape or for multiple beams. For a moving beam, data was also simulated in this way, with $\boldsymbol{\mu}$ falling along a deterministic path.

4.2 Algorithm Implementation

The basic EM algorithm implementation is found in Algorithm 1. Modifications for adding a prior when tracking a moving beam, modifications for multiple beams, and integration with the Kalman filter and Hungarian algorithm are listed in Appendix A, with Matlab code included in Appendix B.

5 Experimental Results

5.1 Single Beam Estimation

Figure 2 shows the results of position estimation for a single static beam. Figure 2a shows a sample image of a detector, where a small number of photons have been detected. The EM algorithm clearly outperforms the centroid as a beam position

Algorithm 1 Expectation-Maximization

Inputs:
  $\mathbf{\Sigma}_s, \Lambda_s, \Lambda_n, A, \mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$

Initialize:
  $\hat{\boldsymbol{\mu}}_s^{(0)} = \hat{\boldsymbol{\mu}}_{COG} = \frac{1}{n}\sum_{j=1}^{n} \mathbf{x}_j$
  $\lambda_n = \frac{\Lambda_n}{\|A\|}$

for $t = 1, \ldots, t_{max}$ or $\hat{\boldsymbol{\mu}}_s^{(t)} = \hat{\boldsymbol{\mu}}_s^{(t-1)}$ **do**
  $\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t-1)}) = \frac{\Lambda_s}{2\pi\sqrt{|\mathbf{\Sigma}_s|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})\mathbf{\Sigma}_s^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$
  $w(\mathbf{x}, \hat{\boldsymbol{\mu}}_s^{(t-1)}) = \frac{\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t-1)})}{\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t-1)}) + \lambda_n}$
  $\hat{\boldsymbol{\mu}}_s^{(t)} = \frac{\sum_{j=1}^{n} w(\mathbf{x}_j, \hat{\boldsymbol{\mu}}_s^{(t-1)})\mathbf{x}_j}{\sum_{j=1}^{n} w(\mathbf{x}_j, \hat{\boldsymbol{\mu}}_s^{(t-1)})}$

end for

estimate. Figure 2b shows a different instance of detections for the same parameters, with data clustered via k-means for $k = 2$. We observe that the centroid of one of the clusters is very near the true position estimate, suggesting that hard clustering may be an alternative method of position estimation. The beam detections have smaller spatial variance than the uniformly distributed noise, so we could choose the cluster containing more points as a "signal cluster," since the signal detections contribute little to the within-cluster-sum-of-squares if the centroid is close to the true $\mathbf{x}$.

Figures 2c and 2d show the trends for these types of estimation for a larger range of parameters. For $\Lambda_s = 50$ and 500, we investigated the performance for 100 estimation trials in terms of the input SNR. The performance of EM vs the centroid closely mirrors that in [1], with EM producing significantly more accurate position estimates for all SNRs. The k-means method consistently performs worse than EM but better than the centroid in terms of output SNR, although for $k = 3$, the low-SNR performance approaches that of EM. This intuitively makes sense, since k-means is less tailored to the particular problem than EM, but it does reduce the noise contribution to the position estimate relative to the centroid method.

5.1.1 Position Sensing for Unknown Beam Covariance

Beam position estimation was also evaluated for a beam of an unknown size/shape as given by the covariance $\mathbf{\Sigma}_s$. Figure 3 shows results versus SNR for $\Lambda_s = 500$, which was similar across other signal detection rates. The basic version of EM as in Algorithm 1 was tested against a modified version where the beam covariance was also estimated at each iteration. The usual data generation function was modified to create signal detections of a correlated bivariate Gaussian with random marginal variances. It was assumed for this investigation that the mean marginal standard deviation was $\rho = 40$, which matched the parameter for our circular Gaussian beam previously. The key difficulty for the modified algorithm was finding a suitable covariance initialization. A first attempt using the global covariance proved no better than the centroid at position estimation. Eventually, cross-validation determined that the best initialization was with a circularly symmetric Gaussian with standard deviation slightly larger or smaller than $\rho = 40$. Figure 3 shows that choosing $\rho = 35$ was slightly better at high SNR and $\rho = 45$ was slightly better at low SNR, but in either case, there was little benefit in performance for position estimation over using the basic, non-adaptive EM. This result justifies our continued assumption of circular symmetry, even though in practice, the incident beams may not be perfectly orthogonal to the detector.

5.1.2 Beam Presence Detection

While our previous performance evaluations of EM position estimation had performed well, we had always assumed the presence of a signal in the data. In practical scenarios, however, it may not always be known whether a beam signal is indeed present at the detector. This is particularly important for a moving beam, since position estimates from only noise would lead to poor tracking over time. Approaching the problem of beam detection as a binary hypothesis test, the difficulty with deciding whether a beam is present is that the beam position itself is unknown. In order to make such a decision we followed a simple procedure: 1) make a position estimate $\hat{\boldsymbol{\mu}}_s$ with EM, regardless of whether signal is present; 2) for the given signal and noise rates, calculate the theoretical CDFs for only noise and for a mixture of signal and noise, assuming the signal is located at $\hat{\boldsymbol{\mu}}_s$; 3) compare the empirical CDF from the data to the theoretical CDFs; 4) decide whether signal is present depending on which theoretical CDF is closest to the data in terms of mean-squared error. In practice, calculating the empirical CDF for low-flux data is unreliable, since the detector matrix is extremely sparse. However, since both the signal and noise had independent coordinates, a simple workaround was to integrate detections over the rows and columns and compare the 1D results to the theoretical results separately. Figure 4 shows the results for a range of input SNRs. The classification of signal presence or absence performed perfectly for input SNR greater than zero, so only low-SNR performance is shown. It is clear from the plot that higher signal detection rates make detecting beam presence much easier, since the correct classification rate (CCR) is consistently higher.

5.2 Multi-Beam Tracking

Figure 5a shows the true and estimated paths for two incident beams. The Kalman filtered estimates follow the true path fairly smoothly even when the beams overlap. The largest error in either beam is only about 6% of the detector's diagonal size, as shown in Figure 5b. Figures 5c and 5d provide an additional visualization of the path tracking.

The quality of the beam tracking during the overlap is largely due to the inclusion of the prior based on previous beam positions. When the beams cross one another, the k-medians algorithm clusters the signal photoevents together, and creates a second cluster composed almost entirely of noise. Without a prior, the EM algorithm would produce one good beam estimate and one estimate in the center of the noise cluster. This would be acceptable if the beams passed one another quickly, as the Kalman filter can predict the path for a few frames at a time before losing track of it. In this case the beams stay close together for numerous frames, and the prior is required to force the EM algorithm to produce estimates based on the beams' speed of travel. However, if the beams remained close together for too long, the standard deviation of the prior would grow, and eventually the EM estimates would end up completely degraded by noise.

6 Conclusions

Through testing against other methods and successful integration within larger systems, it is clear that the EM algorithm is a high-performing tool for optical position sensing. Expectation Maximization proved to be best method for beam position estimation at an individual frame, with virtually no decline in performance even when the beam shape was unknown. Taking advantage of the accuracy of the estimation results using EM, the algorithm was further useful as a preprocessing tool for a number of systems. The position estimate produced by EM was sufficient to decide whether data included a beam signal with high accuracy, even at low SNR. Furthermore, in tracking the motion of single or multiple moving beams, EM provided reasonable position estimates that could be improved on with an applied Kalman filter and matching algorithms.

While this work covers a number of applications of EM for optical position sensing, there are countless variations that we could have implemented. A more general-purpose version of the algorithm would perform estimation with even fewer assumptions, such as unknown signal and noise rates Λ_s and Λ_n. A multiple-beam tracking application would also benefit from a nonparametric implementation, where the number of beams is not known in advance, so the position estimates could adapt to beams leaving the detector surface or new beams entering. While the combination of k-means and EM worked adequately for multiple beams when the centers were sufficiently separated, future adjustments could compare this method with the traditional Gaussian Mixture Model, which has its own EM implementation. Finally, instead of applying the Kalman filter after EM produces position estimates, a more advanced algorithm could use the Kalman prediction step to adjust the prior on the EM estimate.

7 Description of Individual Effort

Josh Implementation of data generation from model, implementation of EM algorithm with unknown covariance Σ_s, performance evaluations for known covariance (EM vs k-means and centroid) and unknown covariance, performance evaluation of beam presence detection

Myles Implementation of EM algorithm, EM algorithm with prior, prior parameter estimation, multi-beam data generation, and multi-beam static position estimation.

David Implementation of the Kalman Filter for single beam tracking using both static and dynamic EM provided by Myles, implementation of the Kalman Filter integrated with the Hungarian Algorithm for multibeam tracking, performance statistics of both trackings.

References

[1] B. J. Slocumb and D. L. Snyder, "Maximum-likelihood estimation applied to quantum-limited optical position-sensing," in *Acquisition, Tracking, and Pointing IV*, vol. 1304, pp. 165–176, 1990.

[2] A. Hero, "Theoretical limits for optical position estimation using imaging arrays," in *13 Colloque sur le traitement du signal et des images, FRA, 1991*, GRETSI, Groupe dEtudes du Traitement du Signal et des Images, 1991.

[3] S. L. Meeks, W. A. Tom, T. R. Willoughby, P. A. Kupelian, T. H. Wagner, J. M. Buatti, and F. J. Bova, "Optically guided patient positioning techniques," *Seminars in Radiation Oncology*, vol. 15, no. 3, pp. 192 – 201, 2005.

[4] L. L. Scharf, *Statistical signal processing*, vol. 98. Addison-Wesley Reading, MA, 1991.

[5] S. Kishner and T. Barnard, "Point-source position estimation in under-sampled optical systems," *JOSA*, vol. 63, no. 12, pp. 1578–1583, 1973.

[6] S. Haykin, *Communication systems.* John Wiley & Sons, 2008.

[7] D. Snyder and M. Miller, *Random Point Processes in Time and Space.* Springer Texts in Electrical Engineering, Springer New York, 1991.

[8] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society. Series B (methodological)*, pp. 1–38, 1977.

[9] P. Kalane, "Target tracking using kalman filter," *International Journal of Science & Technology*, vol. 2, 2012.

[10] J. F. Felix Lutteke, Xu Zhang, "Implementation of the hungarian method for object tracking on a camera monitored transportation system," *7th German Conference on Proceedings of ROBOTIK 2012*, pp. 1–6, 2012.

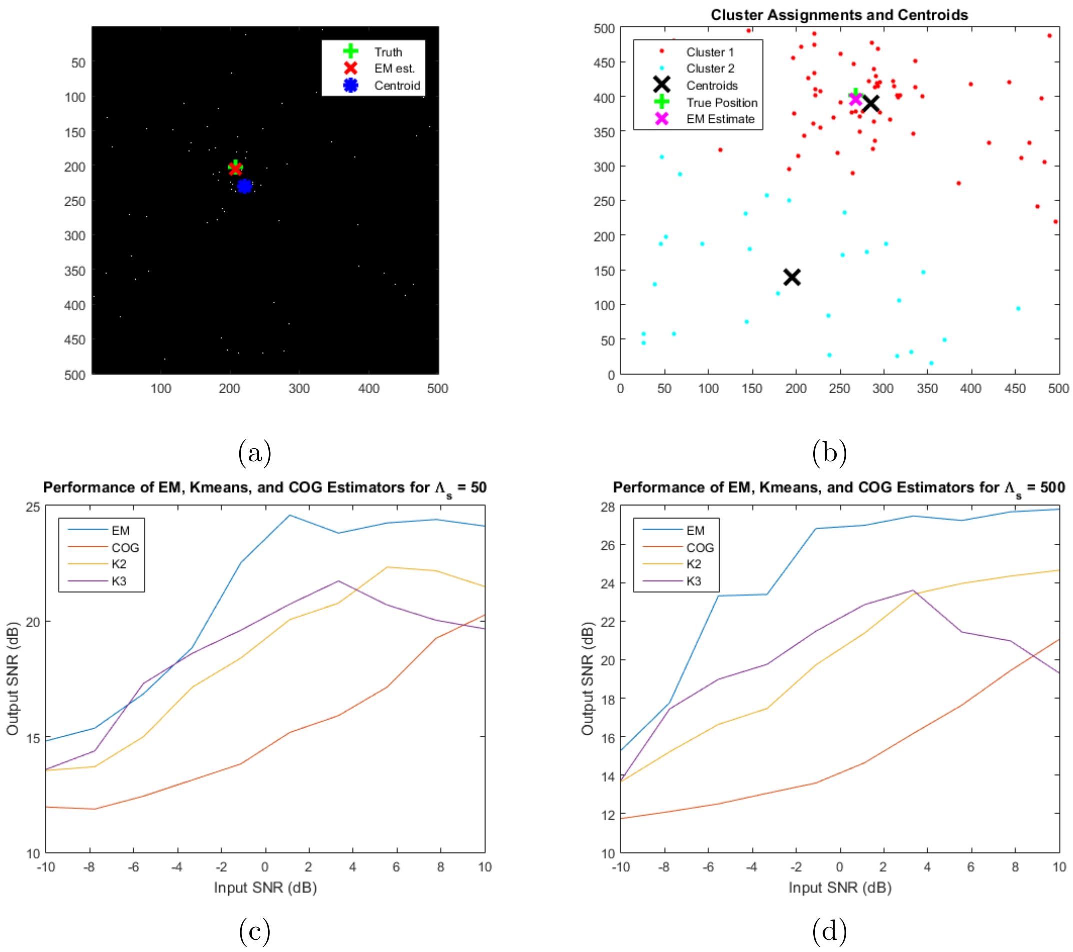


Figure 2: Results from investigations with a single static beam for $\rho = 40$, $\Lambda_s = 50$, $\Lambda_n = 50$: (a) shows a sample image of data collected at detector, (b) shows clustering performed by k-means for $k = 2$. The performance of EM is evaluated for estimation against the centroid and k-means methods (choosing the cluster with the most detections) for (c) $\Lambda_s = 50$ and (d) $\Lambda_s = 500$. The performance closely mirrors that in [1].

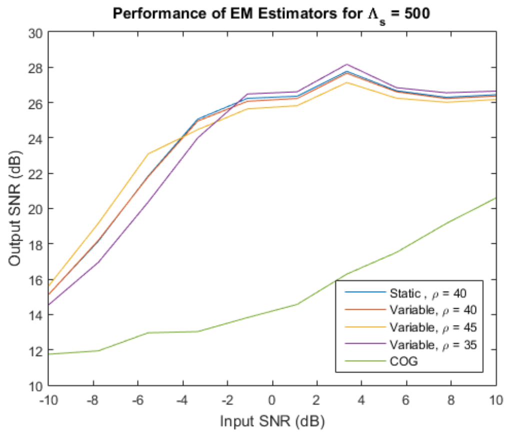


Figure 3: Estimation performance for unknown beam covariance. Adaptively estimating the covariance within the EM algorithm only marginally improves position estimation performance over the "static" algorithm where a circularly symmetric beam is assumed.

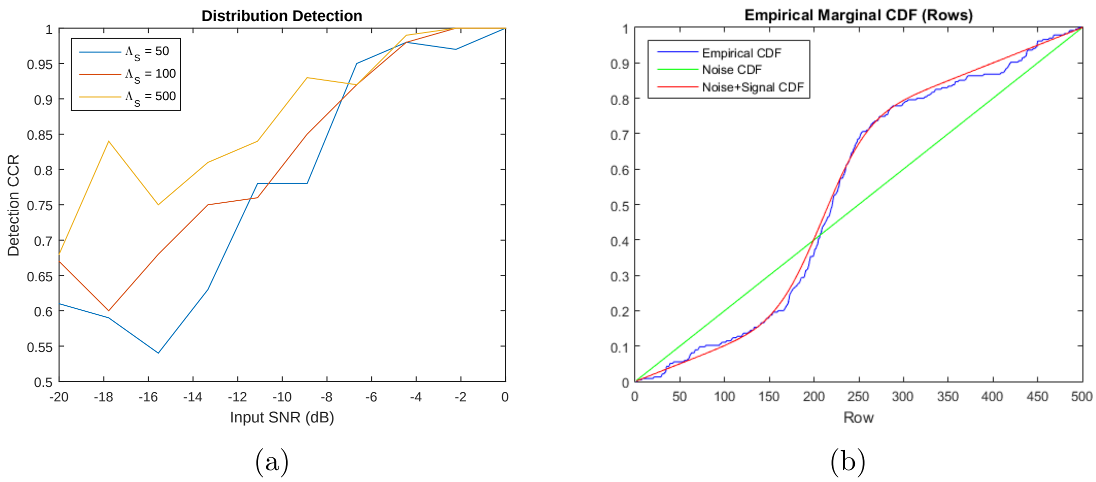
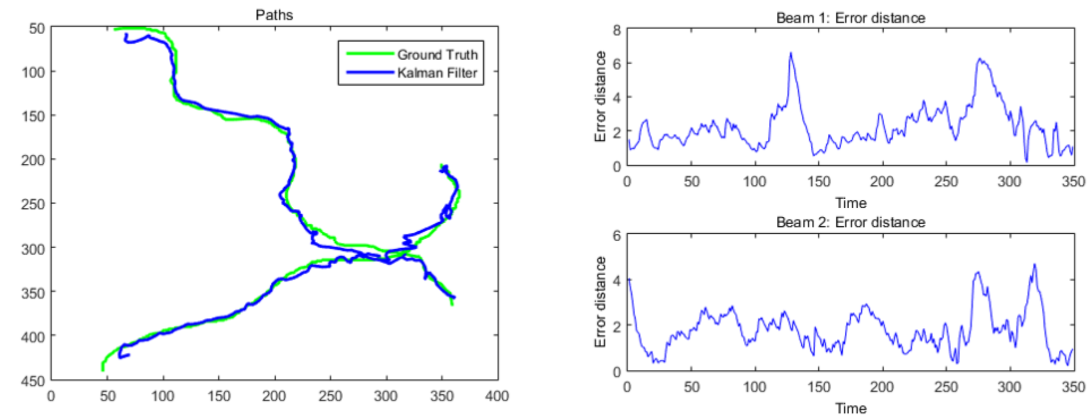
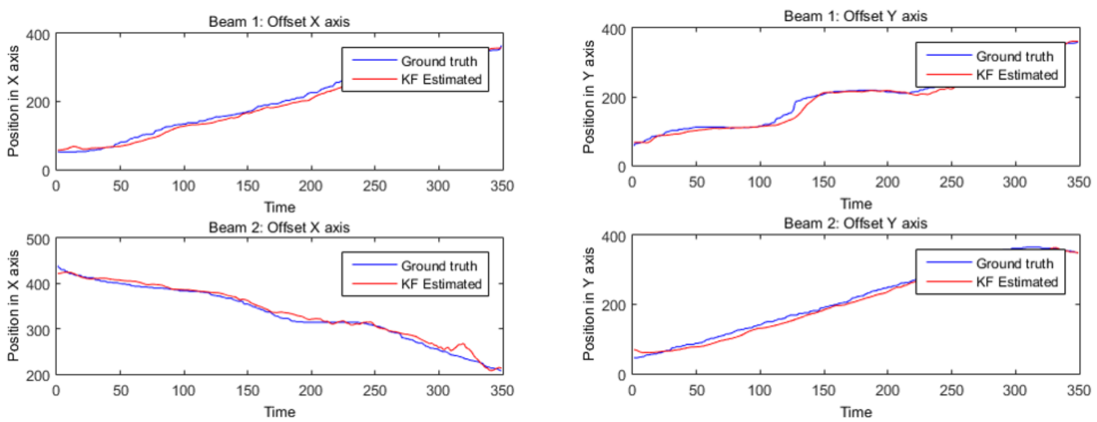


Figure 4: Determining whether data has a signal present: (a) Performance evaluation vs. SNR; (b) Sample processing step, with empirical 1D CDFs compared to theoretical CDFs of possible distributions.



(a) Estimated and true beam paths

(b) Path estimation errors



(c) Estimated path Offsets, X axis

(d) Estimated path offsets, Y axis

Figure 5: Results for motion tracking with two beams moving at constant rates. Even when the two beams pass in close proximity of one another, the Hungarian algorithm successfully matches the position estimate to the correct beam, and the Kalman filter ensures the two beams are still distinguished.

A Algorithms

Algorithm 2 Expectation-Maximization with Prior

Inputs:
  $\mathbf{\Sigma}_s, \Lambda_s, \Lambda_n, A, \mathbf{X}_{\text{uc}} = \{\mathbf{x}_{1,\text{uc}}, \dots, \mathbf{x}_{n,\text{uc}}\}, \hat{\boldsymbol{\mu}}_{s,\text{prev.}}$
Initialize:
  $\hat{\boldsymbol{\mu}}_s^{(0)} = \hat{\boldsymbol{\mu}}_{COG} = \frac{1}{n} \sum_{j=1}^{n} \mathbf{x}_j$
  $\lambda_n = \frac{\Lambda_n}{\|A\|}$
Center Data: $\mathbf{X} = \mathbf{X}_{\text{uc}} - \hat{\boldsymbol{\mu}}_{s,\text{prev.}}$
for $t = 1, \dots, t_{max}$ or $\hat{\boldsymbol{\mu}}_s^{(t)} = \hat{\boldsymbol{\mu}}_s^{(t-1)}$ **do**
  $\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t-1)}) = \frac{\Lambda_s}{2\pi\sqrt{|\mathbf{\Sigma}_s|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})\mathbf{\Sigma}_s^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)$
  $w(\mathbf{x}, \hat{\boldsymbol{\mu}}_s^{(t-1)}) = \frac{\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t-1)})}{\lambda_s(\hat{\boldsymbol{\mu}}_s^{(t-1)}) + \lambda_n}$
  $\hat{\boldsymbol{\mu}}_s^{(t)} = \frac{\sum_{j=1}^{n} w(\mathbf{x}_j, \hat{\boldsymbol{\mu}}_s^{(t-1)})\mathbf{x}_j}{\sum_{j=1}^{n} w(\mathbf{x}_j, \hat{\boldsymbol{\mu}}_s^{(t-1)}) + (\rho/\sigma)^2}$
end for
Shift final estimate back to original coordinate system

Algorithm 3 Expectation-Maximization for Multiple Beams

Inputs:
  $\mathbf{\Sigma}_s, \Lambda_s, \Lambda_n, A, \mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}, k$
Cluster: Apply k-medians algorithm
for cluster $= 1, \dots, k$ **do**
  Apply EM algorithm to cluster
end for

Algorithm 4 Kalman Filter for Single Beam Tracking

Inputs:
  $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}, R, Q, P, A, H, B$
Initialize:
  $\mathbf{x}_0 = \mathbf{z}_1$
for $k = 1, \ldots, numObservations$ **do**
  Prediction:
   $x_k^- = Ax_{k-1} + Bu_k$
   $P_k^- = AP_{k-1}A^T + Q$
  Correction:
   $K_k = P_k^- H^T (HP_K^- H^T + R)^{-1}$
   $x_k = x_k^- + K_k(z_k - Hx_k^-)$
   $P_k = (I - K_k H)P_k^-$
end for

Algorithm 5 Kalman Filter and Hungarian Algorithm for Multi Beam Tracking

Inputs:
  $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$(For each beam)$, R, Q, P, A, H, B$
Initialize:
  $\mathbf{x}_0 = \mathbf{z}_1$
for $k = 1, \ldots, numObservations$ **do**
  for $beam = 1, \ldots, numBeams$ **do**
   Prediction (for each beam):
   $x_k^- = Ax_{k-1} + Bu_k$
   $P_k^- = AP_{k-1}A^T + Q$
  end for
  Hungarian Algorithm (Matching predictions with measurements for each beam)
  for $beam = 1, \ldots, numBeams$ **do**
   Correction (for each beam):
   $K_k = P_k^- H^T (HP_K^- H^T + R)^{-1}$
   $x_k = x_k^- + K_k(z_k - Hx_k^-)$
   $P_k = (I - K_k H)P_k^-$
  end for
end for

Algorithm 6 Hungarian Method

```
1: Inputs:
       Cost Matrix: Assignment of beam (in rows) to measurement (in
       columns)
2: Subtract the smallest entry in each row from all the entries of its row.
3: Subtract the smallest entry in each column from all the entries of its column.
4: Draw lines through appropriate rows and columns so that all the zero entries of
   the cost matrix are covered and the minimum number of such lines is used.
5: Test for Optimality: (i) If the minimum number of covering lines is n, an optimal
   assignment of zeros is possible and we are finished. (ii) If the minimum number
   of covering lines is less than n, an optimal assignment of zeros is not yet possible.
   In that case, proceed to 6.
6: Determine the smallest entry not covered by any line. Subtract this entry from
   each uncovered row, and then add it to each covered column. Return to 4.
```

B Matlab Code

B.1 Data Generation

B.1.1 Generate Data

```
function [ sig_pos, matDetect, listDetect,labels ] = fcn_generate_data( Lr,Lc,rho,Lam_s,Lam_n )
%FCN_GENERATE_DATA takes in parameters about signal and noise detection
%rates and generates a dataset based on the model of a circular Gaussian
%signal and uniform noise.
%
% The output includes both a 2D-detector view of detections (better for visualization)
% as well as a vector of detection coordinates (easier to process).
%
% ********************************************************************************
% Input Parameters
%---------------
% [Lr, Lc] = size of detector array represented by matrix
% rho = signal standard deviation
% sigma = prior standard deviation
% Lam_s = beam photo-conversion rate
% Lam_n Noise photoconversion rate
%
% Output Parameters
% ----------------
% sig_pos = coordinates of true signal position
% matDetect = matrix of signal detections
% listDetect = list of detection coordinates
% ********************************************************************************

sig_pos = [rho+round((Lr-2*rho)*rand), rho+round((Lc-2*rho)*rand)];

numSig = poissrnd(Lam_s);
numNoise = poissrnd(Lam_n);

sigPreDetect = [sig_pos(1)+round(rho*randn(numSig,1)),sig_pos(2)+
```

```
    round(rho*randn(numSig,1))];

sigDetect = sigPreDetect(sigPreDetect(:,1)>0,:);
sigDetect = sigDetect(sigDetect(:,2)>0,:);
sigDetect = sigDetect(sigDetect(:,1)<=Lr,:);
sigDetect = sigDetect(sigDetect(:,2)<=Lc,:);

noiseDetect = [randi(Lr,[numNoise, 1]),randi(Lc,[numNoise, 1])];
listDetect = [sigDetect; noiseDetect];
labels = [ones(length(sigDetect),1);zeros(length(noiseDetect),1)
    ];

matDetect = zeros(Lr,Lc);

for ii= 1:length(listDetect)
    matDetect(listDetect(ii,1),listDetect(ii,2)) = 1;
end
```

B.1.2 Generate Data to Detect Distribution

```
function [sig_pos,matDetect,listDetect,label] =
    fcn_generate_distribution(Lr,Lc,rho,Lam_s,Lam_n)
%FCN_GENERATE_DISTRIBUTION takes in parameters about signal and
    noise detection
%rates and generates a dataset based on the model of a circular
    Gaussian
%signal and uniform noise.
%
% The output includes both a 2D-detector view of detections (
    better for visualization)
% as well as a vector of detection coordinates (easier to process
    ).
%
%
    ***********************************************************************
% Input Parameters
%----------------
% [Lr, Lc] = size of detector array represented by matrix
% rho = signal standard deviation
% sigma = prior standard deviation
% Lam_s = beam photo-conversion rate
% Lam_n Noise photoconversion rate
%
% Output Parameters
% ------------------
% sig_pos = coordinates of true signal position
```

```
% matDetect = matrix of signal detections
% listDetect = list of detection coordinates
% ************************************************************************

numSig = poissrnd(Lam_s);
numNoise = poissrnd(Lam_n);

if randi([0 1])
    sig_pos = [rho+round((Lr-2*rho)*rand), rho+round((Lc-2*rho)*rand)];
    sigPreDetect = [sig_pos(1)+round(rho*randn(numSig,1)),sig_pos(2)+round(rho*randn(numSig,1))];

    sigDetect = sigPreDetect(sigPreDetect(:,1)>0,:);
    sigDetect = sigDetect(sigDetect(:,2)>0,:);
    sigDetect = sigDetect(sigDetect(:,1)<=Lr,:);
    sigDetect = sigDetect(sigDetect(:,2)<=Lc,:);

    noiseDetect = [randi(Lr,[numNoise, 1]),randi(Lc,[numNoise, 1])];
    listDetect = [sigDetect; noiseDetect];
    label = 1;
else
    sig_pos = [0,0];
    numDetect = numSig+numNoise;
    listDetect = [randi(Lr,[numDetect, 1]),randi(Lc,[numDetect, 1])];
    label = 0;
end

matDetect = zeros(Lr,Lc);

for ii= 1:length(listDetect)
    matDetect(listDetect(ii,1),listDetect(ii,2)) = matDetect(listDetect(ii,1),listDetect(ii,2))+1;
end
```

B.1.3 Generate Data with Unknown Covariance Matrix

```
function [sig_pos,Sigma_cov,matDetect,listDetect,labels] = ...
    fcn_generate_correlated_data( Lr,Lc,sig_hat,Lam_s,Lam_n )
%FCN_GENERATE_DATA takes in parameters about signal and noise detection
%rates and generates a dataset based on the model of a circular Gaussian
```

```
%signal and uniform noise.
%
% The output includes both a 2D-detector view of detections (better for visualization)
% as well as a vector of detection coordinates (easier to process).
%
% ***********************************************************************
% Input Parameters
%----------------
% [Lr, Lc] = size of detector array represented by matrix
% rho = signal standard deviation
% sigma = prior standard deviation
% Lam_s = beam photo-conversion rate
% Lam_n Noise photoconversion rate
%
% Output Parameters
% ---------------------
% sig_pos = coordinates of true signal position
% matDetect = matrix of signal detections
% listDetect = list of detection coordinates
% ***********************************************************************

sig_pos = [sig_hat+round((Lr-2*sig_hat)*rand), sig_hat+round((Lc-2*sig_hat)*rand)];

numSig = poissrnd(Lam_s);
numNoise = poissrnd(Lam_n);

sig1 = randi([sig_hat/2,3*sig_hat/2]);
sig2 = randi([sig_hat/2,3*sig_hat/2]);
rho = -1+2*rand(1);
Sigma_cov = [sig1^2, rho*sig1*sig2; rho*sig1*sig2, sig2^2];
sigPreDetect = round(mvnrnd(sig_pos,Sigma_cov,numSig));

%sigPreDetect = [sig_pos(1)+round(sig_hat*randn(numSig,1)), sig_pos(2)+round(sig_hat*randn(numSig,1))];

sigDetect = sigPreDetect(sigPreDetect(:,1)>0,:);
sigDetect = sigDetect(sigDetect(:,2)>0,:);
sigDetect = sigDetect(sigDetect(:,1)<=Lr,:);
sigDetect = sigDetect(sigDetect(:,2)<=Lc,:);
```

```
noiseDetect = [randi(Lr,[numNoise, 1]),randi(Lc,[numNoise, 1])];
listDetect = [sigDetect; noiseDetect];
labels = [ones(length(sigDetect),1);zeros(length(noiseDetect),1)
    ];

matDetect = zeros(Lr,Lc);

for ii= 1:length(listDetect)
    matDetect(listDetect(ii,1),listDetect(ii,2)) = matDetect(
        listDetect(ii,1),listDetect(ii,2))+1;
end
```

B.2 EM Algorithms

B.2.1 Basic EM

```
function xhats = staticEM(detector_data,nonzero_coords,rho,LS,LN,
    itmax)
% use the expectation maximization algorithm to estimate the
    position on an
% optical beam on a 2D detector array
%
% Inputs:
%
% detector_data - an MxN matrix of photoevent counts at each grid
    point
%
% nonzero_coords - Lx2 matrix of coordinate pairs [row col; row
    col; row
% col; ...] where L is the number of locations where at least one
    photo count occured
%
% rho - beam spatial variance (scalar double) (probably in units
    of num
% grid points)
%
% LS - mean # signal photoconversions
%
% LN - mean # noise photoevents
%
% itmax - maximum number of iterations allowed (default Inf)
%
% Outputs:
%
% xhat - the position estimate [row position column position]

```

```
if nargin == 5
    itmax = Inf;
end

L = size(nonzero_coords,1);
num_row = size(detector_data,1);
num_col = size(detector_data,2);
A = num_row*num_col;
R = diag([rho^2 rho^2]);
Rinv = inv(R);

% get initial estimate (COG estimate)
cogsum = 0;
for nzpair = 1:L
    row = nonzero_coords(nzpair,1);
    col = nonzero_coords(nzpair,2);
    rc_pair = [row col];
    counts = detector_data(row,col);
    cogsum = cogsum + counts*rc_pair;
end
xhat = cogsum / sum(detector_data(:));
xhat = round(xhat)';

% apply EM
lambdaN = LN/A;
itnum = 1;
xhats{1} = xhat;
while true

    num_sum = 0;
    denom_sum = 0;

    for nzpair = 1:L

        row = nonzero_coords(nzpair,1);
        col = nonzero_coords(nzpair,2);
        d = [row; col];
        lambdaS = LS/(2*pi*rho^2) * exp(-0.5*(d - xhat)'*Rinv*(d
            - xhat));
        w = lambdaS/(lambdaS + lambdaN);

        counts = detector_data(row,col);
        num_sum = num_sum + counts * w * d;
        denom_sum = denom_sum + counts * w;

    end
```

```

    xhat_tminus1 = xhat;
    xhat = round(num_sum/denom_sum);

    if isequal(xhat,xhat_tminus1) || itnum == itmax
        break;
    end
    itnum = itnum + 1;
    xhats{itnum} = xhat;
end
```

B.2.2 EM Unknown Covariance

```
function [xhats, Rhats] = variableEM(matDetect,listDetect,
    sigma_hat,Lam_S,Lam_N,itmax)
% use the expectation maximization algorithm to estimate the
    position on an
% optical beam on a 2D detector array
%
% Inputs:
%
% matDetect - an MxN matrix of photoevent counts at each grid
    point
%
% nonzero_coords - Lx2 matrix of coordinate pairs [row col; row
    col; row
% col; ...] where L is the number of locations where at least one
     photo count occured
%
% LS - mean # signal photoconversions
%
% LN - mean # noise photoevents
%
% itmax - maximum number of iterations allowed (default Inf)
%
% Outputs:
%
% xhat - the position estimate [row position column position]

if nargin == 5
    itmax = Inf;
end

numDetect = size(listDetect,1);
num_row = size(matDetect,1);
num_col = size(matDetect,2);
Area = num_row*num_col;
```

```

%% Initialization
% x estimate
cogsum = 0;
for nzpair = 1:numDetect
    row = listDetect(nzpair,1);
    col = listDetect(nzpair,2);
    rc_pair = [row col];
    counts = matDetect(row,col);
    cogsum = cogsum + counts*rc_pair;
end
xhat = cogsum / sum(matDetect(:));
xhat = round(xhat)';

% Covariance estimate
if sigma_hat==0
    Rhat = ((listDetect-ones(numDetect,1)*xhat')'*(listDetect-ones(numDetect,1)*xhat'))/numDetect;
else
    Rhat = diag([sigma_hat^2;sigma_hat^2]);
end
Rinv = inv(Rhat);

% apply EM
lambdaN = Lam_N/Area;
itnum = 1;
xhats{1} = xhat;
Rhats{1} = Rhat;
while true
    sigma_hat = sqrt(det(Rhat));

    num_sum = 0;
    denom_sum = 0;
    R_num_sum = 0;

    for nzpair = 1:numDetect

        row = listDetect(nzpair,1);
        col = listDetect(nzpair,2);
        d = [row; col];
        lambdaS = Lam_S/(2*pi*sigma_hat) * exp(-0.5*(d - xhat)'*Rinv*(d - xhat));
        weight = lambdaS/(lambdaS + lambdaN);

        counts = matDetect(row,col);
        num_sum = num_sum + counts * weight * d;
```

```
            denom_sum = denom_sum + counts * weight;

            R_num_sum = R_num_sum+ weight*(d - xhat)*(d - xhat)';

        end

        xhat_tminus1 = xhat;
        xhat = round(num_sum/denom_sum);
        Rhat = R_num_sum/denom_sum;

        if isequal(xhat,xhat_tminus1) || itnum == itmax
            break;
        end
        itnum = itnum + 1;
        xhats{itnum,1} = xhat;
        Rhats{itnum,1} = Rhat;
end
```

B.2.3 EM with Prior for Dynamic Tracking

```
function xhat = dynamicEM(detector_data,nonzero_coords,prev_xhat,
    sigma,rho,LS,LN)
% use the expectation maximization algorithm to estimate the
    position on an
% optical beam on a 2D detector array
%
% Inputs:
%
% detector_data - an MxN matrix of photoevent counts at each grid
    point
%
% prev_xhat - the most recent position estimate (2x1 or 1x2
    vector)
%
% rho - beam spatial variance (scalar double) (probably in units
    of num
% grid points)
%
% LS - mean # signal photoconversions
%
% LN - mean $ noise photoevents
%
% Outputs:
%
% xhat - the position estimate [row position column position]

if size(prev_xhat) == size(ones(1,2))
```

```
    prev_xhat = prev_xhat';
end
L = size(nonzero_coords,1);
num_row = size(detector_data,1);
num_col = size(detector_data,2);
A = num_row*num_col;
Rinv = inv(diag([rho^2 rho^2]));

xhat = prev_xhat;

% determine sigma

static_xhats = staticEM(detector_data,nonzero_coords,rho,LS,LN);
staticX = static_xhats{end};

if isempty(sigma)
    sigma = 0.25*norm(staticX - prev_xhat);
end

% make prev_xhat the origin of the grid, get estimate and adjust using prev_xhat to get value with true
% origin
lambdaN = LN/A;
while true

    num_sum = 0;
    denom_sum = 0;
    xhat = xhat - prev_xhat;
    for nzpair = 1:L
        row = nonzero_coords(nzpair,1);
        col = nonzero_coords(nzpair,2);

        d = [row; col] - prev_xhat; % set previous xhat as origin
        lambdaS = LS/(2*pi*rho^2) * exp(-0.5*(d - xhat)'*Rinv*(d-xhat));
        w = lambdaS/(lambdaS + lambdaN);

        counts = detector_data(row,col);
        num_sum = num_sum + counts * w * d;
        denom_sum = denom_sum + counts * w;
    end

    xhat_tminus1 = xhat + prev_xhat;
    xhat = round(num_sum/(denom_sum + (rho/sigma)^2));
    xhat = xhat + prev_xhat; % reset origin
```

```

    if xhat == xhat_tminus1
%     if norm(xhat - xhat_tminus1) <= 0
        break;
    end

end
```

B.2.4 EM for Multiple Beams

```
function xhat = multibeamEM(detector_data,nonzero_coords,rho,LS,LN,...
    prev_xhats,sigma,num_beams)
% use the expectation maximization algorithm to estimate the position on an
% optical beam on a 2D detector array
%
% need to  assign cluster to prev_xhat entry based on centroid and
% prev_xhat position
%
% Inputs:
%
% detector_data - an MxN matrix of photoevent counts at each grid point
%
% nonzero_coords - Lx2 matrix of coordinate pairs [row col; row col; row
% col; ...] where L is the number of locations where at least one photo count occured
%
% rho - beam spatial variance (scalar double) (probably in units of num
% grid points)
%
% LS - mean # signal photoconversions
%
% LN - mean # noise photoevents
%
% prev_xhats - most recent position estimates (cell array, one cell per beam) (use [] for static case)
%
% sigma - standard deviation for beam locationn(use[] for static case)
%
% num_beams - number of beams
%
```

```
% Outputs:
%
% xhats - the position estimates for each beam cell array of cell arrays

[clusters,Cs] = kmeans(nonzero_coords,num_beams,'Replicates',20,'Distance','cityblock');

% assign clusters to prev_xhats
cluster_prev_xhats = cell(1,length(prev_xhats)); % idx 1 corresponds to cluster 1, idx 2 to cluster 2, etc.
xhat_choices = prev_xhats;
if ~isempty(prev_xhats)
    for cluster_num = 1:size(Cs,1)
        cent = Cs(cluster_num,:)'; % centroid of cluster
        xhatDiffs = cellfun(@(x) norm(x-cent),xhat_choices); % find distance for each xhat to centroid
        [~,idx] = min(xhatDiffs); % index of the unassigned xhat that is closest to centroid
        cluster_prev_xhats{cluster_num} = xhat_choices{idx}; % assign nearest unassigned xhat to cluster
        xhat_choices(idx) = []; % remove assigned xhat
    end
end


xhat = cell(1,num_beams);
for beam_num = 1:num_beams
    beam_data = nonzero_coords(clusters == beam_num,:);
    if isempty(prev_xhats)
        xhats = staticEM(detector_data,beam_data,rho,LS,LN);
        xhat{beam_num} = xhats{end};
    elseif ~isempty(prev_xhats)
        prev_xhat = cluster_prev_xhats{beam_num};
        xhat{beam_num} = dynamicEM(detector_data,beam_data,prev_xhat,sigma,rho,LS,LN);
    end
end
```

B.3 Beam Tracking Functions

B.3.1 Kalman Filter

```
function [estPath] = kalman2D(observ, Lr, Lc)
%% System parameters
dt      = 1; % sampling interval
t       = 1; % starting frame
```

```
u         = .005; % control input
x_init    = [observ(t,1); observ(t,2); 0; 0]; % Initial Conditions
x_est     = x_init; % state estimate
noise     = .1; % process noise intensity
noise_x = 1; % noise for x and y are
noise_y = 1; % chosen by user and the same
visualize = 0; % visualize the tracking
numObserv = length(observ);

%% Kalman Filter params
R         = [noise_x 0; ...
             0 noise_y]; %coviarance of the noise
Q         = [dt^4/4 0 dt^3/2 0; ...
             0 dt^4/4 0 dt^3/2; ...
             dt^3/2 0 dt^2 0; ...
             0 dt^3/2 0 dt^2] .* noise^2; % Covariance of the observation noise
P         = Q; % Estimate of initial state
A         = [1 0 dt 0; 0 1 0 dt; 0 0 1 0; 0 0 0 1];  %State transition model
H         = [1 0 0 0; 0 1 0 0];  % Observation model
B         = [(dt^2/2); (dt^2/2); dt; dt];  % Control-input model
z         = []; % The measurements of the node state
x_sta_est = []; %  Initial state estimate
v_est     = []; % Initial velocity estimate
P_est     = P; % Initial covariance matrix

%% Perform Kalman Filter
% figure
for i = t:numObserv
    img       = ones(Lr, Lc, 3); % Create a blank image for visualization
    z(i,:)    = [observ(i,1) observ(i,2)]; % Current measurement coordinates

    % Time Update
    x_est     = A * x_est + B * u; % Project the state ahead
    P         = A * P * A' + Q; % Project the error covariance ahead
    % Measurement Update
    K         = P * H' / (H * P * H' + R); % Compute the Kalman Gain
    if ~isnan(z(i,:))
        x_est = x_est + K * (z(i,:)' - H * x_est); % Update estimate with measurement
    end
```

```
    P         = (eye(4) - K * H) * P; % Update error covariance

    x_sta_est   = [x_sta_est; x_est(1:2)'];
    v_est       = [v_est; x_est(3:4)'];

    x_estimation(i) = x_est(1); %estimation in horizontal position
    y_estimation(i) = x_est(2); %estimation in vertical position

    if visualize==1
        r = 5;
        ang=0:.01:2*pi; %parameters of nodes
        imagesc(img);
        set(gca,'YDir','normal')
%       axis off
        hold on
        plot(r * cos(ang) + ground(i,1), r * sin(ang) + ground(i,2), '.g'); % Ground truth motion
        plot(r * cos(ang) + z(i,1), r * sin(ang) + z(i,2), '.b'); % The measurement motion
        plot(r * cos(ang) + x_est(1), r * sin(ang) + x_est(2), '.r'); % The kalman filtered motion
        hold off
        legend('Ground truth', 'Measurement', 'Kalman Filter')
        pause(0.05)
    end
end

x_estimation = x_estimation';
y_estimation = y_estimation';

estPath = [x_estimation y_estimation];
```

B.3.2 Kalman Filtering Multiple Beams

```
function [estPaths] = multitrack2D(X,Y,Lr,Lc,numBeams)
%% System parameters
dt          = 1; % Sampling interval
startFrame  = 1; % Starting frame
u           = 0; % Control input
noise       = .1; % process noise intensity
noise_x     = 1;  % measurement noise in the horizontal direction (x axis).
noise_y     = 1;  % measurement noise in the horizontal direction (y axis).
%% Kalman parameters
R   = [noise_x 0; ...
```

```
          0 noise_y]; % Covariance of the noise
Q     = [dt^4/4 0 dt^3/2 0; ...
          0 dt^4/4 0 dt^3/2; ...
          dt^3/2 0 dt^2 0; ...
          0 dt^3/2 0 dt^2].*noise^2; % Covariance of the observation noise
P     = Q; % Estimate of initial state
A     = [1 0 dt 0; 0 1 0 dt; 0 0 1 0; 0 0 0 1]; % State transition model
B     = [(dt^2/2); (dt^2/2); dt; dt]; % Control-input model
H     = [1 0 0 0; 0 1 0 0];  % Observation model
%% Multi tracking parameters
beamObserv       = [X{startFrame} Y{startFrame} zeros(length(X{startFrame}),1) zeros(length(X{startFrame}),1)]';
beamEstimation   = nan(4,2000);
beamEstimation(:, 1:size(beamObserv, 2)) = beamObserv;  % Initial estimate
beamXestimation = nan(2000); % X estimate
beamYestimation = nan(2000); % Y estimate
P_est            = P; % Initial covariance matrix
trackLost        = zeros(1,2000); % How many times a track was not assigned
numDetect        = size(X{startFrame},1); % Initial number of detections
numBeam          = find(isnan(beamEstimation(1, :)) == 1, 1) - 1; % Initial number of track estimates

%% Start the multi-tracking
for t = startFrame:length(X)
    beamMeasurement = [X{t} Y{t}]; % Matrix with current measurements
    %% Perform Kalman Filter
    % Time Update (Prediction of state for all the beams)
    numDetect = size(X{t},1); % How many detections in current time
    for beam = 1:numBeam
        beamEstimation(:,beam) = A * beamEstimation(:,beam) + B * u;
    end
    P = A * P* A' + Q;

    %% Perform Hungarian Algorithm
    % Create the distance matrix between all the detections
    % For the matrix, it is assigned: rows = tracks & columns = detections
    distMatrix = pdist([beamEstimation(1:2, 1:numBeam)';
```

```
        beamMeasurement]);
    distMatrix = squareform(distMatrix); % Create the squared distance matrix
    distMatrix = distMatrix(1:numBeam, numBeam+1:end) ; % Do only matching for the tracks detected

    [assignment, cost] = assignmentoptimal(distMatrix); % Hungarian Algorithm
    assignment = assignment';

    % Check exceptions where matching must be ignored and just estimate
    % In those casses assignment = 0
    % Detection far from observation
    rejected = [];
    for beam = 1:numBeam
        if assignment(beam) > 0
            rejected(beam) =  distMatrix(beam,assignment(beam)) < 50 ;
        else
            rejected(beam) = 0;
        end
    end
    assignment = assignment .* rejected;
    % Done with matching

    % Measurement Update (Correction of state for all the beams)
    K = P * H' / ( H * P * H' + R);
    k = 1;
    for beam = 1:length(assignment)
        if assignment(beam) > 0
            beamEstimation(:, k) = beamEstimation(:, k) + K * ( beamMeasurement(assignment(beam), :)' - H * beamEstimation(:, k));
        end
        k = k + 1;
    end
    P = (eye(4)- K * H) * P; % Update error covariance

    %% Store data
    beamXestimation(t,1:numBeam) = beamEstimation(1,1:numBeam);
    beamYestimation(t,1:numBeam) = beamEstimation(2,1:numBeam);

    % Assigning new detections and lost trackings
    % For new detections: If it wasn't assigned means a new beam
    newTracks = beamMeasurement(~ismember(1: size(beamMeasurement
```

```
            , 1), assignment), :)';
        if ~isempty(newTracks)
            beamEstimation(:, numBeam + 1:numBeam + size(newTracks,
                2)) = ...
                [newTracks; zeros(2, size(newTracks, 2))];
            % Number of estimated beams including new ones
            numBeam = numBeam + size(newTracks, 2);
        end

        % If a tracking didn't get matched with a detection,
        % a counter will start
        noTrackInList = find(assignment == 0);
        if ~isempty(noTrackInList)
            trackLost(noTrackInList) = trackLost(noTrackInList) + 1;
        end

        % If a track has a counter greater than 6, the tracking will
            be deleted
        % and reseted to NaN
        bad_trks = find(trackLost > 6);
        beamEstimation(:,bad_trks) = NaN;

        %% Visualization
        %{
        clf
        img = ones(500, 500, 3); % Create a blank image for
            visualization
        imagesc(img);
        hold on;
        plot(Y{t}(:),X{t}(:),'or'); % Plot measurements
        colours = ['r', 'b', 'g', 'c', 'm', 'k'];
        for nB = 1:numBeam
            if ~isnan(beamXestimation(t,nB))
                cIdx = mod(nB,6) + 1; %pick color
                tempX = beamXestimation(1:t,nB);
                tempY = beamYestimation(1:t,nB);
                plot(tempY, tempX, '.-', 'MarkerSize', 3, 'Color',
                    colours(cIdx), 'LineWidth',3)
                axis off
            end
        end
        pause(0.05)
        %}
%       t
end
```

```
% Creating the estimated paths for each beam
for i=1:numBeams
    estPaths{i} = [beamXestimation(1:length(X),i) beamYestimation(1:length(Y),i)];
end
```

B.3.3 Hungarian matching algorithm

```
function [assignment, cost] = assignmentoptimal(distMatrix)
%ASSIGNMENTOPTIMAL    Compute optimal assignment by Munkres algorithm
%   ASSIGNMENTOPTIMAL(DISTMATRIX) computes the optimal assignment (minimum
%   overall costs) for the given rectangular distance or cost matrix, for
%   example the assignment of tracks (in rows) to observations (in
%   columns). The result is a column vector containing the assigned column
%   number in each row (or 0 if no assignment could be done).
%
%   [ASSIGNMENT, COST] = ASSIGNMENTOPTIMAL(DISTMATRIX) returns the
%   assignment vector and the overall cost.
%
%   The distance matrix may contain infinite values (forbidden
%   assignments). Internally, the infinite values are set to a very large
%   finite number, so that the Munkres algorithm itself works on
%   finite-number matrices. Before returning the assignment, all
%   assignments with infinite distance are deleted (i.e. set to zero).
%
%   A description of Munkres algorithm (also called Hungarian algorithm)
%   can easily be found on the web.
%
%   <a href="assignment.html">assignment.html</a>  <a href="http://www.mathworks.com/matlabcentral/fileexchange/6543">File Exchange</a>  <a href="https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=EVW2A4G2HBVAU">Donate via PayPal</a>
%
%   Markus Buehren
%   Last modified 05.07.2011

% save original distMatrix for cost computation
```

```
originalDistMatrix = distMatrix;

% check for negative elements
if any(distMatrix(:) < 0)
  error('All matrix elements have to be non-negative.');
end

% get matrix dimensions
[nOfRows, nOfColumns] = size(distMatrix);

% check for infinite values
finiteIndex   = isfinite(distMatrix);
infiniteIndex = find(~finiteIndex);
if ~isempty(infiniteIndex)
  % set infinite values to large finite value
  maxFiniteValue = max(max(distMatrix(finiteIndex)));
  if maxFiniteValue > 0
    infValue = abs(10 * maxFiniteValue * nOfRows * nOfColumns);
  else
    infValue = 10;
  end
  if isempty(infValue)
    % all elements are infinite
    assignment = zeros(nOfRows, 1);
    cost       = 0;
    return
  end
  distMatrix(infiniteIndex) = infValue;
end

% memory allocation
coveredColumns = zeros(1,       nOfColumns);
coveredRows    = zeros(nOfRows, 1);
starMatrix     = zeros(nOfRows, nOfColumns);
primeMatrix    = zeros(nOfRows, nOfColumns);

% preliminary steps
if nOfRows <= nOfColumns
  minDim = nOfRows;

  % find the smallest element of each row
  minVector = min(distMatrix, [], 2);

  % subtract the smallest element of each row from the row
  distMatrix = distMatrix - repmat(minVector, 1, nOfColumns);

```

```
  % Steps 1 and 2
  for row = 1:nOfRows
    for col = find(distMatrix(row,:)==0)
      if ~coveredColumns(col)%~any(starMatrix(:,col))
        starMatrix(row, col) = 1;
        coveredColumns(col)  = 1;
        break
      end
    end
  end

else % nOfRows > nOfColumns
  minDim = nOfColumns;

  % find the smallest element of each column
  minVector = min(distMatrix);

  % subtract the smallest element of each column from the column
  distMatrix = distMatrix - repmat(minVector, nOfRows, 1);

  % Steps 1 and 2
  for col = 1:nOfColumns
    for row = find(distMatrix(:,col)==0)'
      if ~coveredRows(row)
        starMatrix(row, col) = 1;
        coveredColumns(col)  = 1;
        coveredRows(row)     = 1;
        break
      end
    end
  end
  coveredRows(:) = 0; % was used auxiliary above
end

if sum(coveredColumns) == minDim
  % algorithm finished
  assignment = buildassignmentvector(starMatrix);
else
  % move to step 3
  [assignment, distMatrix, starMatrix, primeMatrix,
     coveredColumns, coveredRows] = ...
      step3(distMatrix, starMatrix, primeMatrix, coveredColumns,
        coveredRows, minDim); %#ok
end

% compute cost and remove invalid assignments
```

```
[assignment, cost] = computeassignmentcost(assignment, originalDistMatrix, nOfRows);


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function assignment = buildassignmentvector(starMatrix)

[maxValue, assignment] = max(starMatrix, [], 2);
assignment(maxValue == 0) = 0;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [assignment, cost] = computeassignmentcost(assignment, distMatrix, nOfRows)

rowIndex    = find(assignment);
costVector = distMatrix(rowIndex + nOfRows * (assignment(rowIndex)-1));
finiteIndex = isfinite(costVector);
cost = sum(costVector(finiteIndex));
assignment(rowIndex(~finiteIndex)) = 0;

% Step 2: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [assignment, distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows] = ...
   step2(distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows, minDim)

% cover every column containing a starred zero
maxValue = max(starMatrix);
coveredColumns(maxValue == 1) = 1;

if sum(coveredColumns) == minDim
  % algorithm finished
  assignment = buildassignmentvector(starMatrix);
else
  % move to step 3
  [assignment, distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows] = ...
   step3(distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows, minDim);
```

```
end

% Step 3: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [assignment, distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows] = ...
    step3(distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows, minDim)

zerosFound = 1;
while zerosFound

  zerosFound = 0;
  for col = find(~coveredColumns)
    for row = find(~coveredRows')
      if distMatrix(row,col) == 0

        primeMatrix(row, col) = 1;
        starCol = find(starMatrix(row,:));
        if isempty(starCol)
          % move to step 4
          [assignment, distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows] = ...
            step4(distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows, row, col, minDim);
          return
        else
          coveredRows(row)        = 1;
          coveredColumns(starCol) = 0;
          zerosFound              = 1;
          break % go on in next column
        end
      end
    end
  end
end

% move to step 5
[assignment, distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows] = ...
    step5(distMatrix, starMatrix, primeMatrix, coveredColumns, coveredRows, minDim);

% Step 4: %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [assignment, distMatrix, starMatrix, primeMatrix,
    coveredColumns, coveredRows] = ...
     step4(distMatrix, starMatrix, primeMatrix, coveredColumns,
        coveredRows, row, col, minDim)

newStarMatrix          = starMatrix;
newStarMatrix(row,col) = 1;

starCol = col;
starRow = find(starMatrix(:, starCol));

while ~isempty(starRow)

  % unstar the starred zero
  newStarMatrix(starRow, starCol) = 0;

  % find primed zero in row
  primeRow = starRow;
  primeCol = find(primeMatrix(primeRow, :));

  % star the primed zero
  newStarMatrix(primeRow, primeCol) = 1;

  % find starred zero in column
  starCol = primeCol;
  starRow = find(starMatrix(:, starCol));

end
starMatrix = newStarMatrix;

primeMatrix(:) = 0;
coveredRows(:) = 0;

% move to step 2
[assignment, distMatrix, starMatrix, primeMatrix, coveredColumns,
     coveredRows] = ...
     step2(distMatrix, starMatrix, primeMatrix, coveredColumns,
        coveredRows, minDim);


% Step 5:
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [assignment, distMatrix, starMatrix, primeMatrix,
    coveredColumns, coveredRows] = ...
```

```
    step5(distMatrix, starMatrix, primeMatrix, coveredColumns,
        coveredRows, minDim)

% find smallest uncovered element
uncoveredRowsIndex    = find(~coveredRows');
uncoveredColumnsIndex = find(~coveredColumns);
[s, index1] = min(distMatrix(uncoveredRowsIndex,
    uncoveredColumnsIndex));
[s, index2] = min(s); %#ok
h = distMatrix(uncoveredRowsIndex(index1(index2)),
    uncoveredColumnsIndex(index2));

% add h to each covered row
index = find(coveredRows);
distMatrix(index, :) = distMatrix(index, :) + h;

% subtract h from each uncovered column
distMatrix(:, uncoveredColumnsIndex) = distMatrix(:,
    uncoveredColumnsIndex) - h;

% move to step 3
[assignment, distMatrix, starMatrix, primeMatrix, coveredColumns,
    coveredRows] = ...
    step3(distMatrix, starMatrix, primeMatrix, coveredColumns,
        coveredRows, minDim);
```

B.4 Performance Evaluations

B.4.1 Static Single Beam Evaluation

```
%% Explore EM Performance
% Joshua Rapp
% April 23, 2016

clear; close all; clc;

%%
numS = 3;
numN = 10;
numTrials = 100;

Lr = 500; Lc = 500;
Lam_s = [50,100,500];
rho = 40;
itmax = 100;

meanCOG = zeros(numS,numN);
```

```
meanEM = zeros(numS,numN);
meanKMEANS2 = zeros(numS,numN);
meanKMEANS3 = zeros(numS,numN);
meanSPECT = zeros(numS,numN);

for jj = 1:numS
    lam_s = Lam_s(jj);
    disp(num2str(lam_s));
    Lam_n = round(logspace(log10(lam_s/10),log10(10*lam_s),numN));

    for kk = 1:numN
        lam_n = Lam_n(kk);
        disp(num2str(lam_n));

        EMdist = zeros(numTrials,1);
        COGdist = zeros(numTrials,1);
        K2MEANSdist = zeros(numTrials,1);
        K3MEANSdist = zeros(numTrials,1);
        SPECTdist = zeros(numTrials,1);

        for t = 1:numTrials
            [ sig_pos, matDetect, listDetect,labels  ] = fcn_generate_data(Lr,Lc,rho,lam_s,lam_n);
            numDetect = length(labels);

%             figure; plot(listDetect(labels==1,1),listDetect(labels==1,2),...
%                 'r.',listDetect(labels==0,1),listDetect(labels==0,2),'b.');

            % EM Estimate
            xhats = staticEM(matDetect,listDetect,rho,lam_s,lam_n,itmax);
            EM_est = xhats{end};

            % COG Estimate
            COG_est = mean(listDetect);

            % kmeans estimate, k=2
            KMEANS2_est = kmeans_estimate( listDetect, 2);

            % kmeans estimate, k=3
            KMEANS3_est = kmeans_estimate( listDetect, 3 );

            % Spectral Clustering
```

```
            %SPECT_est = kmeans_estimate( listDetect , 2, 'spectral', 1);

            % Euclidean Distance
            EMdist(t) = sqrt((EM_est(1)-sig_pos(1))^2+(EM_est(2)-sig_pos(2))^2);
            COGdist(t) = sqrt((COG_est(1)-sig_pos(1))^2+(COG_est(2)-sig_pos(2))^2);
            K2MEANSdist(t) = sqrt((KMEANS2_est(1)-sig_pos(1))^2+(KMEANS2_est(2)-sig_pos(2))^2);
            K3MEANSdist(t) = sqrt((KMEANS3_est(1)-sig_pos(1))^2+(KMEANS3_est(2)-sig_pos(2))^2);

        end

        meanEM(jj,kk) = mean(EMdist);
        meanCOG(jj,kk) = mean(COGdist);
        meanKMEANS2(jj,kk) = mean(K2MEANSdist);
        meanKMEANS3(jj,kk) = mean(K3MEANSdist);

    end
end

save('slocumb_validation4.mat');

%%
SNRi = 10*log10(fliplr(logspace(-1,1,numN)));

BW = rho*sqrt(log(4));
SNRo_EM = 10*log10(BW^2./meanEM);
SNRoCOG = 10*log10(BW^2./meanCOG);
SNRoKMEANS2 = 10*log10(BW^2./meanKMEANS2);
SNRoKMEANS3 = 10*log10(BW^2./meanKMEANS3);
SNRoSPECT = 10*log10(BW^2./meanSPECT);

for ii = 1:numS
    figure; plot(SNRi,SNRo_EM(ii,:),SNRi,SNRoCOG(ii,:),SNRi,SNRoKMEANS2(ii,:),SNRi,SNRoKMEANS3(ii,:));
    xlabel('Input SNR (dB)');
    ylabel('Output SNR (dB)');
    legend('EM','COG','K2','K3','Location','northwest');
    title(['Performance of EM, Kmeans, and COG Estimators for \Lambda_s = ' num2str(Lam_s(ii))]);
end
```

B.4.2 Beam Presence Detection Evaluation

```
%% Explore EM Performance
% Joshua Rapp
% April 23, 2016

clear; close all; clc;

%%
numS = 3;
numN = 10;
numTrials = 100;

Lr = 500; Lc = 500;

noise_min = 1;
noise_max = 100;

Lams = [50,100,500];
Lam_s = [];
Lam_n = [];

for ii = 1:numS
    Lam_s = [Lam_s,Lams(ii)*ones(1,numN)];
    Lam_n = [Lam_n,round(logspace(log10(Lams(ii)*noise_min),...
        log10(noise_max*Lams(ii)),numN))];
end

rho = 40;
itmax = 100;

rows = 1:Lr;
cols = 1:Lc;

%% Theoretical Marginals - Uniform (Noise Only)
unif_rows = ones(Lr,1)/Lr;
marg_unif_rows = cumsum(unif_rows)/sum(unif_rows);
unif_cols = ones(Lc,1)/Lc;
marg_unif_cols = cumsum(unif_cols)/sum(unif_cols);

CCRs = zeros(numS*numN,1);

parfor jj = 1:numS*numN
    lam_s = Lam_s(jj);
    lam_n = Lam_n(jj);
    disp(['Sig: ' num2str(lam_s) ', Noise: ' num2str(lam_n)]);

    true_labels = zeros(numTrials,1);
```

```
    predict_labels = zeros(numTrials,1);

    for t = 1:numTrials
        [ sig_pos, matDetect,listDetect,label] =
           fcn_generate_distribution(Lr,Lc,rho,lam_s,lam_n);
        true_labels(t) = label;

        % EM Gaussian Center Estimation
        xhats = staticEM(matDetect,listDetect,rho,lam_s,lam_n,
           itmax);
        xest = xhats{end};

        % Theoretical Marginal - Gaussian + Uniform (Noise and
           Signal)
        gauss_rows = (lam_s*normpdf(rows,xest(1),rho)+lam_n*
           unif_rows')/(lam_s+lam_n);
        marg_gauss_rows = cumsum(gauss_rows)/sum(gauss_rows);
        gauss_cols = (lam_s*normpdf(cols,xest(2),rho)+lam_n*
           unif_cols')/(lam_s+lam_n);
        marg_gauss_cols = cumsum(gauss_cols)/sum(gauss_cols);

        % Compute Marginals
        data_rows = sum(matDetect,2);
        marg_cdf_rows = cumsum(data_rows)/sum(data_rows);
        data_cols = sum(matDetect,1)';
        marg_cdf_cols = cumsum(data_cols)/sum(data_cols);

        MSE_Noise_cols = mean((marg_cdf_cols-marg_unif_cols).^2);
        MSE_Noise_rows = mean((marg_cdf_rows-marg_unif_rows).^2);
        MSE_Signal_cols = mean((marg_cdf_cols-marg_gauss_cols')
           .^2);
        MSE_Signal_rows = mean((marg_cdf_rows-marg_gauss_rows')
           .^2);

        [~,dist_predict] = min([MSE_Noise_cols+MSE_Noise_rows,
           MSE_Signal_cols+MSE_Signal_rows]);
        predict_labels(t) = dist_predict -1;

    end

    CCRs(jj) = sum(true_labels==predict_labels)/numTrials;
end

save('distribution_detection3.mat');

%%
```

```
SNRi = 10*log10(fliplr(logspace(log10(1/noise_max),log10(
    noise_min),numN)));
CCRs = reshape(CCRs,numN,numS);

figure; plot(SNRi,CCRs);
xlabel('Input SNR (dB)');
ylabel('Detection CCR');
legend('\Lambda_S = 50','\Lambda_S = 100','\Lambda_S = 500','
    Location','northwest');
title('Distribution Detection');
```

B.4.3 Variable Beam Size Evaluation

```
%% Performance vs SNR, Unknown spot size
% Joshua Rapp
% April 23, 2016

clear; close all; clc;

%%
numS = 3;
numN = 10;
numTrials = 100;

Lr = 500; Lc = 500;
Lams = [50,100,500];

rho_hat1 = 40;
rho_hat2 = 45;
rho_hat3 = 35;

itmax = 100;

meanCOG = zeros(numS,numN)';
meanStaticEM = zeros(numS,numN)';
meanVar1 = zeros(numS,numN)';
meanVar2 = zeros(numS,numN)';
meanVar3 = zeros(numS,numN)';

noise_min = 0.1;
noise_max = 10;

Lams = [50,100,500];
Lam_s = [];
Lam_n = [];

for ii = 1:numS
```

```
    Lam_s = [Lam_s,Lams(ii)*ones(1,numN)];
    Lam_n = [Lam_n,round(logspace(log10(Lams(ii)*noise_min) ,...
        log10(noise_max*Lams(ii)),numN))];
end
%%
parfor jj = 1:numS*numN
    lam_s = Lam_s(jj);
    lam_n = Lam_n(jj);
    disp(num2str(lam_n));

    StaticDist = zeros(numTrials,1);
    Var1Dist = zeros(numTrials,1);
    Var2Dist = zeros(numTrials,1);
    Var3Dist = zeros(numTrials,1);
    COGdist = zeros(numTrials,1);

    for t = 1:numTrials
        [sig_pos,~,matDetect,listDetect,labels] = ...
            fcn_generate_correlated_data( Lr,Lc,rho_hat1,lam_s,
                lam_n );
        numDetect = length(labels);

        % EM Estimate
        xhats = staticEM(matDetect,listDetect,rho_hat1,lam_s,
            lam_n,itmax);
        EM_est = xhats{end};

        x_var1 = variableEM(matDetect,listDetect,rho_hat1,lam_s,
            lam_n,itmax);
        Var1_est = x_var1{end};

        x_var2 = variableEM(matDetect,listDetect,rho_hat2,lam_s,
            lam_n,itmax);
        Var2_est = x_var2{end};

        x_var3 = variableEM(matDetect,listDetect,rho_hat3,lam_s,
            lam_n,itmax);
        Var3_est = x_var3{end};

        % COG Estimate
        COG_est = mean(listDetect);

        % Euclidean Distance
        StaticDist(t) = sqrt((EM_est(1)-sig_pos(1))^2+(EM_est(2)-
            sig_pos(2))^2);
        Var1Dist(t) = sqrt((Var1_est(1)-sig_pos(1))^2+(Var1_est
```

```
(2)-sig_pos(2))^2);
            Var2Dist(t) = sqrt((Var2_est(1)-sig_pos(1))^2+(Var2_est(2)-sig_pos(2))^2);
            Var3Dist(t) = sqrt((Var3_est(1)-sig_pos(1))^2+(Var3_est(2)-sig_pos(2))^2);
            COGdist(t) = sqrt((COG_est(1)-sig_pos(1))^2+(COG_est(2)-sig_pos(2))^2);

        end

        meanStaticEM(jj) = mean(StaticDist);
        meanVar1(jj) = mean(Var1Dist);
        meanVar2(jj) = mean(Var2Dist);
        meanVar3(jj) = mean(Var3Dist);
        meanCOG(jj) = mean(COGdist);


end

save('slocumb_variable5.mat');

%%
SNRi = 10*log10(fliplr(logspace(-1,1,numN)));

BW = rho_hat1*sqrt(log(4));
SNRo_Static = 10*log10(BW^2./meanStaticEM);
SNRo_Var1 = 10*log10(BW^2./meanVar1);
SNRo_Var2 = 10*log10(BW^2./meanVar2);
SNRo_Var3 = 10*log10(BW^2./meanVar3);
SNRoCOG = 10*log10(BW^2./meanCOG);

for ii = 1:numS
    figure; plot(SNRi,SNRo_Static(:,ii),SNRi,SNRo_Var1(:,ii),...
        SNRi,SNRo_Var2(:,ii),SNRi,SNRo_Var3(:,ii),SNRi,SNRoCOG(:,ii));
    xlabel('Input SNR (dB)');
    ylabel('Output SNR (dB)');
    legend('Static, \rho = 40',['Variable, \rho = ' num2str(rho_hat1)],...
        ['Variable, \rho = ' num2str(rho_hat2)],...
        ['Variable, \rho = ' num2str(rho_hat3)],...
        'COG','Location','southeast');
    title(['Performance of EM Estimators for \Lambda_s = ' num2str(Lams(ii))]);
end
```

B.5 Test Scripts

B.5.1 Test Basic Data Generation and EM Estimation

```matlab
%% Test script for data generation
% Joshua Rapp
% Boston University
% EC 503

clear; close all; clc;

%% Static Data
Lr = 500; Lc = 500;
rho = 40;
Lam_s = 50;
Lam_n = 50;

[ sig_pos, matDetect, listDetect,labels ] = fcn_generate_data(
    Lr,Lc,rho,Lam_s,Lam_n );
centroid = mean(listDetect);
%% Plot Detections as Image
figure; imagesc(matDetect); axis ij image; colormap(gray);
hold on;
plot(sig_pos(2),sig_pos(1),'g+',...
      'MarkerSize',10,'LineWidth',3)

euclid_dist = sqrt(sum((sig_pos-centroid).^2));

%% Apply EM
xhats = staticEM(matDetect,listDetect,rho,Lam_s,Lam_n,20);
xest = xhats{end};
figure; imagesc(matDetect); axis ij image; colormap(gray);
hold on;
plot(sig_pos(2),sig_pos(1),'g+','MarkerSize',10,'LineWidth',3)
plot(xest(2),xest(1),'rx',...
      'MarkerSize',10,'LineWidth',3)

%% Apply k-means
numClusters = 2;
[idx,C,sumd] = kmeans(listDetect,numClusters);
CCR = mean((2-labels)==idx);

cmap = hsv(numClusters);

figure;
plot(listDetect(idx==1,1),listDetect(idx==1,2),'r.','MarkerSize'
    ,12)
```

```
hold on

for ii = 2:numClusters
    plot(listDetect(idx==ii,1),listDetect(idx==ii,2),'.','Color',cmap(ii,:),'MarkerSize',12);
end

plot(C(:,1),C(:,2),'kx','MarkerSize',15,'LineWidth',3)
plot(sig_pos(1),sig_pos(2),'g+','MarkerSize',10,'LineWidth',3)
plot(xest(1),xest(2),'mx','MarkerSize',10,'LineWidth',3)
legend('Cluster 1','Cluster 2','Centroids',...
       'True Position','EM Estimate','Location','NW')
title 'Cluster Assignments and Centroids'
hold off

EMerror = norm(sig_pos-xest');
KMeansError = norm(sig_pos-C(1,:));
%% Tracking Motion
% numFrames = 10;
% speed = 40;
%
% [sig_pos, matDetect, listDetect,labels] = ...
%     fcn_generate_motion_data(Lr,Lc,rho,Lam_s,Lam_n,numFrames,speed);
%
% for ii = 1:numFrames
%     figure; imagesc(matDetect(:,:,ii)); axis image; colormap(gray);
%     hold on;
%     plot(sig_pos(ii,2),sig_pos(ii,1),'g*',...
%      'MarkerSize',10,'LineWidth',3)
% end

% implay(matDetect,1);
```

B.5.2 Test Beam Presence Detection

```
%% Test script for data generation
% Joshua Rapp
% Boston University
% EC 503

clear; close all; clc;

%% Static Data
Lr = 500; Lc = 500;
rho = 40;
```

```
Lam_s = 50;
Lam_n = 100;

[ sig_pos , matDetect , listDetect , label ] = fcn_generate_distribution(Lr,Lc,rho,Lam_s,Lam_n);
centroid = mean(listDetect);

%% Apply EM
xhats = staticEM(matDetect,listDetect,rho,Lam_s,Lam_n,20);
xest = xhats{end};
figure; imagesc(matDetect); axis ij image; colormap(gray);
hold on;
plot(sig_pos(2),sig_pos(1),'g+','MarkerSize',10,'LineWidth',3);
plot(xest(2),xest(1),'rx','MarkerSize',10,'LineWidth',3);
plot(centroid(2),centroid(1),'b*','MarkerSize',10,'LineWidth',3);

legend('Truth','EM est.','Centroid');
%%
rows = 1:Lr;
cols = 1:Lc;

% Theoretical Marginals
unif_rows = ones(Lr,1)/Lr;
marg_unif_rows = cumsum(unif_rows)/sum(unif_rows);
unif_cols = ones(Lc,1)/Lc;
marg_unif_cols = cumsum(unif_cols)/sum(unif_cols);

gauss_rows = (Lam_s*normpdf(rows,xest(1),rho)+Lam_n*unif_rows')/(Lam_s+Lam_n);
marg_gauss_rows = cumsum(gauss_rows)/sum(gauss_rows);
gauss_cols = (Lam_s*normpdf(cols,xest(2),rho)+Lam_n*unif_cols')/(Lam_s+Lam_n);
marg_gauss_cols = cumsum(gauss_cols)/sum(gauss_cols);

% Compute Marginals
data_rows = sum(matDetect,2);
marg_cdf_rows = cumsum(data_rows)/sum(data_rows);
figure; plot(rows,marg_cdf_rows,rows,marg_unif_rows,rows,marg_gauss_rows);
title('Empirical Marginal CDF (Rows)');
xlabel('Row');
legend('Empirical CDF','Noise CDF','Noise+Signal CDF','Location','northwest');

data_cols = sum(matDetect,1)';
marg_cdf_cols = cumsum(data_cols)/sum(data_cols);
```

```
figure; plot(cols,marg_cdf_cols,cols,marg_unif_cols,cols,marg_gauss_cols);
title('Empirical Marginal CDF (Columns)');
xlabel('Column');
legend('Empirical CDF','Noise CDF','Noise+Signal CDF','Location','northwest');
%%
MSE_Noise_cols = mean((marg_cdf_cols-marg_unif_cols).^2);
MSE_Noise_rows = mean((marg_cdf_rows-marg_unif_rows).^2);

MSE_Signal_cols = mean((marg_cdf_cols-marg_gauss_cols').^2);
MSE_Signal_rows = mean((marg_cdf_rows-marg_gauss_rows').^2);

[~,dist_predict] = min([MSE_Noise_cols+MSE_Noise_rows,MSE_Signal_cols+MSE_Signal_rows]);
```

B.5.3 Test Variable-Size Position Estimation

```
%% Test script for data generation
% Joshua Rapp
% Boston University
% EC 503

clear; close all; clc;

%% Static Data
Lr = 500; Lc = 500;
rho_hat = 40;

Lam_s = 100;
Lam_n = 1000;

[sig_pos,Sigma_cov,matDetect,listDetect,labels] = ...
    fcn_generate_correlated_data( Lr,Lc,rho_hat,Lam_s,Lam_n );
centroid = mean(listDetect);

% Apply EM
[x_var, Rvar] = variableEM(matDetect,listDetect,0,Lam_s,Lam_n);
x_var_est = x_var{end};
R_var_est = Rvar{end};

x_stat = staticEM(matDetect,listDetect,rho_hat,Lam_s,Lam_n);
x_stat_est = x_stat{end};

Err_var = sqrt(sum((sig_pos-x_var_est').^2));
Err_stat = sqrt(sum((sig_pos-x_stat_est').^2));

```

```
disp(['Variable Improvement: ' num2str(Err_stat-Err_var)]);

%% Plot

figure; imagesc(matDetect); axis ij image; colormap(gray);
hold on;
plot(sig_pos(2),sig_pos(1),'g+','MarkerSize',10,'LineWidth',3)
plot(x_var_est(2),x_var_est(1),'rx','MarkerSize',10,'LineWidth'
    ,3)
plot(x_stat_est(2),x_stat_est(1),'mp','MarkerSize',10,'LineWidth'
    ,3)
plot(centroid(2),centroid(1),'b*','MarkerSize',10,'LineWidth',3)
legend('Truth','Variable EM','Static EM','Centroid');
```