TENET SIMULATOR USER'S GUIDE

July 2001

Stanton H. Musick
Air Force Research Laboratory
AFRL/SNAT
2241 Avionics Circle
Wright-Patterson AFB, OH 45433

John Greenewald
Veridian Engineering
5200 Springfield Pike, Suite 200
Dayton, OH 45431

TABLE OF CONTENTS

FOREWARD

The Air Force Research Laboratory (AFRL) has issued a Challenge Problem to promote the development of effective numerical methods for nonlinear tracking applications. This Challenge Problem has the acronym TENET, which stands for TEchniques for Nonlinear Estimation of Tracks. As part of the TENET Challenge Problem, we have developed a simulator and associated documentation in the form of this User's Guide. The simulator is itself comprised of a target motion generator, a sensor emulator, two baseline solutions, and postprocessing software for evaluating metrics.

Also in conjunction with the TENET Challenge Problem project, AFRL hosted a Workshop in Feb 2001 (Reference [1]) and has written conference papers on results with the simulator (References [2] and [3]). All TENET materials are publicly available, some at the public web site (https://www.tenet.vdl.afrl.af.mil/) where they are free.

The TENET Simulator is a suite of MATLAB functions written to provide a nonlinear filtering evaluation testbed to the research community. TENET contains two tracker implementations, one based on Particle Filter (PF) methods and the other on Alternating Direction Implicit (ADI) finite difference methods. These baseline implementations illustrate use of the TENET Simulator in concrete terms. The TENET Simulator is written to easily accommodate new trackers. New tracking algorithms should be compared to the two baseline methods according to the provided metrics.

The TENET Simulator is written for easy integration and evaluation of new filtering methods. This user's manual describes the existing simulator, and explains what steps are required to add a new filter for performance comparison.

All TENET Simulator code was developed using MATLAB®. You must have Version 5.3 or later of MATLAB to run the TENET Simulator.

Dr. Keith Kastella and Mr. Chris Kreucher of Veridian Systems developed the ADI and Particle Method MATLAB methods as well as an initial version of the TENET test program. Mr. John Greenewald of Veridian Engineering wrote the TENET Simulator and contributed to this User's Guide.

For the Air Force Research Laboratory, Mr. Stanton Musick (AFRL/SNAT) was the Project Engineer who guided this Challenge Problem effort. Mr. Musick also contributed to this User's Guide.

1 INTRODUCTION AND SUMMARY

1.1 Overview of TENET Simulator

The TENET Simulator is a complete suite of MATLAB functions for studying target tracking problems where track-before-detect solution methods are required, as they are at low signal-to-noise ratios (SNR). The primary functions of the simulator are to generate target truth trajectories and synthetic image data, estimate target tracks, and evaluate tracker performance.

The TENET Simulator consists of six principle modules, each implemented as a MATLAB function. A brief statement about the purpose of each module follows:

- *GenMonteCarlo* – The executive program, generates an ensemble of Monte Carlo simulation runs using GenTrueTrack, GenScenes, RunFilter and CalcMetrics.
- *GenTrueTrack* – Generates a different randomized target trajectory for each Monte Carlo run given initial conditions and process noise strength
- *GenScenes* – Generates pixilated sensor images with embedded target pixels at specified SNR values
- *RunFilter* – Employs the selected tracker (ADI, PF, or user supplied) to process the image data and estimate target position and velocity
- *CalcMetrics* –Calculate performance metrics
- *PostProcess* – Generates plots to assist in analyzing performance

1.2 Guide to this Report

This User's Guide instructs the user on how to install and use the TENET Simulator to develop and compare new algorithms in nonlinear filtering. Section 2 details the steps to download, install and checkout the software. Section 3 describes the parameters that control target motion and sensor imaging. This section also describes a simple user interface to add new filters to the simulator. The goal is to provide maximum flexibility to the user and an efficient data structure that retains all relevant data. Section 4 describes the TENET Simulator software organization. Section 5 details the methods we employed to simulate and track targets using low SNR images. Section 6 contains a list of references on this topic.

1.3 Software Rights

All information and data at the TENET website https://www.tenet.vdl.afrl.af.mil/ is **approved for public release, distribution unlimited.** This means that TENET materials are free of copyrights and restrictive licensing agreements, and may be downloaded for free by anyone with Internet capabilities. In addition, these materials may be employed as the user desires. We of course hope that these materials will foster new research efforts in nonlinear filtering methods.

In order to maintain a traceable connection to the original TENET project, we request that you do not remove the TENET identifier that appears at the top of each MATLAB function:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   TENET Simulator, Version 1.0                                        %
%   https://www.tenet.vdl.afrl.af.mil/                                  %
%                                                                       %
%   AFRL/SNAT                                                           %
%   Sensors Directorate, AF Research Laboratory                         %
%   Wright-Patterson Air Force Base, Ohio 45433                         %
%                                                                       %
%   Software approved for public release; distribution unlimited.       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Observing this request will help to ensure that subsequent users of TENET programs will be able to find the TENET website, which may then prompt them to obtain an update. As TENET matures, we expect that additional solution methods and improved versions of the software will emerge and be made available at the website.

2 INSTALLING THE TENET SIMULATOR

This section provides instructions for acquiring and installing the TENET Software, and then for checking its operation.

2.1 System Requirements

TENET software is MATLAB source code that is designed to execute on any computer that runs MATLAB Version 5.3 or later. This guide illustrates TENET use assuming a personal computer running Windows 2000, but any computer/operating-system that supports MATLAB is acceptable. When installed, TENET source codes occupy less than 100 kilobytes of permanent storage on your hard drive.

2.2 Download and Install Procedure

Follow these steps to download and install the TENET Simulator on your computer.

1. Create a directory on the hard drive of your local computer to receive the TENET software, e.g. c:\TENET.

2. Log on to the TENET website at https://www.tenet.vdl.afrl.af.mil. Click “yes” to proceed when prompted at the Security Alert prompt.

3. Select *Download TENET Simulator M-Files* to download a zip file containing all TENET Simulator MATLAB files. When prompted where to save the file, browse to your TENET directory and click *Save*.

2.3 Checking the Software

This section discusses some simple runs that will help to verify that the downloaded software is operating correctly. Both single- and multi-run tests are discussed.

2.3.1 Single Run Simulation

Run a single-run test of the TENET Simulator to see an example simulation and evaluation. It will probably take a few minutes for this run to finish. Begin MATLAB and change to your TENET directory (**cd c:\TENET**). From the MATLAB command line type:

```
MC = GenMonteCarlo(1);
```

Figures with images should appear on the screen simulating a sensor with a target under track. See Figures 1-3 for an example. Note that Figure 1 shows the particle values and a 1-

sigma ellipse around the estimate. Figure 1 and Figure 2 show the velocity pdf for the ADI method. The simulation completes after twenty frames (20 sec) of data are processed.

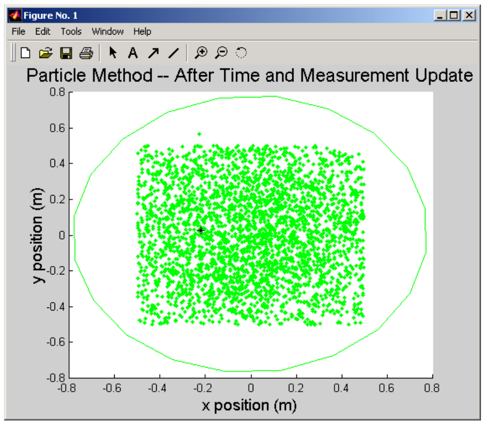


Figure 1 – Particle Method density after time and measurement update

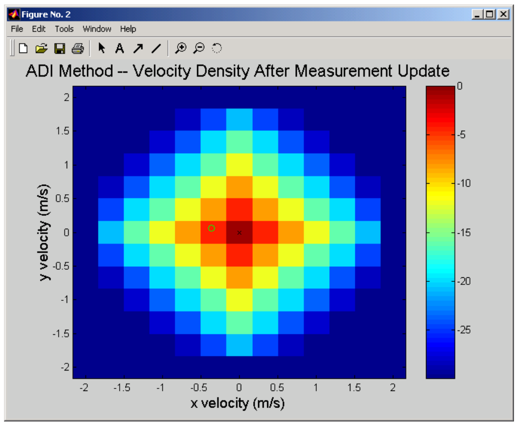


Figure 2 - ADI Method density after time and measurement update

The data from this run is saved to the structure MC and to an automatically generated file. The data may be postprocessed to produce additional useful plots. The saved file name has the following format: MC*Date_Description_Count*.mat, where *Date* is in YYYYMMDD format, *Description* is a underscore delimited list of the Filters field (in GenMonteCarlo.m) and *Count* is a counter to distinguish the results of other simulations.

To postprocess the data type:

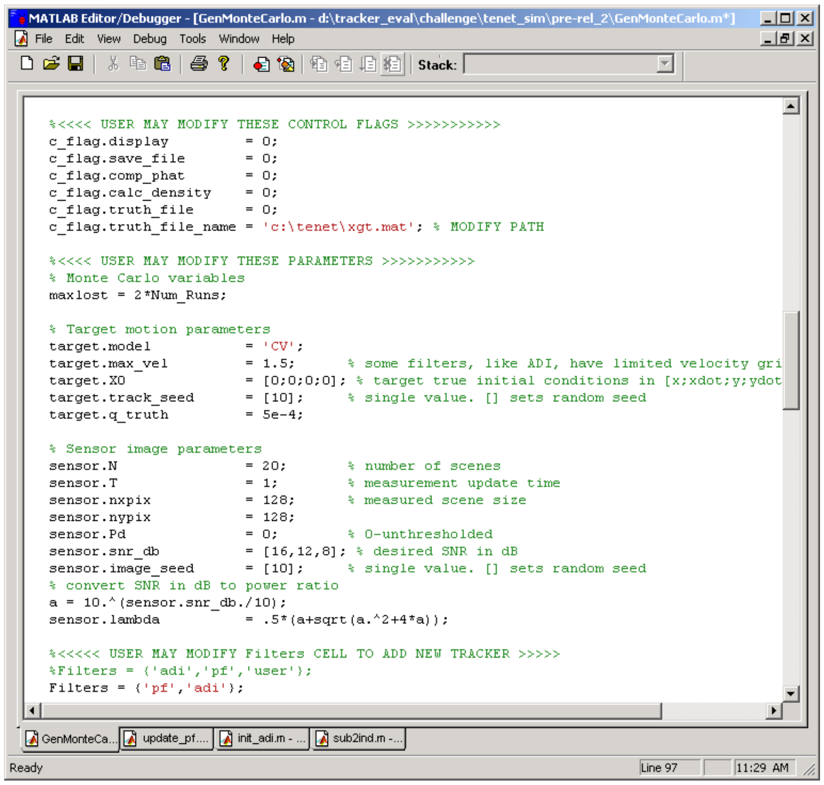
PostProcess(MC);

Plots showing tracker position and velocity errors should appear.

2.3.2 Multiple Run Simulation

In this checkout test we run the TENET Simulator for five Monte Carlo runs of 20 sec each to see an example of ensemble processing. Again this will take about 40 minutes to complete, depending on processor speed. A wait bar will show progress over the three SNR choices.

Begin MATLAB and move to your TENET directory (**cd c:\TENET**). Edit file GenMonteCarlo.m and change its code to be the same as that shown in Figure 3. Set each of the c_flags to zero. In Sensor Image, set snr_db = [16,12,8] to simulate three different SNR levels in the image, and make IMAGE_SEED = [10]. Finally in Tracker, choose TRACK_SEED = [10].


MATLAB Editor/Debugger - [GenMonteCarlo.m - d:\tracker_eval\challenge\tenet_sim\pre-rel_2\GenMonteCarlo.m*]
File Edit View Debug Tools Window Help
Stack:
GenMonteCa... update_pf.... init_adi.m - ... sub2ind.m - ...
Ready
Line 97
11:29 AM

```
%<<<< USER MAY MODIFY THESE CONTROL FLAGS >>>>>>>>>>>
c_flag.display         = 0;
c_flag.save_file       = 0;
c_flag.comp_phat       = 0;
c_flag.calc_density    = 0;
c_flag.truth_file      = 0;
c_flag.truth_file_name = 'c:\tenet\xgt.mat'; % MODIFY PATH

%<<<< USER MAY MODIFY THESE PARAMETERS >>>>>>>>>>>
% Monte Carlo variables
maxlost = 2*Num_Runs;

% Target motion parameters
target.model           = 'CV';
target.max_vel         = 1.5;      % some filters, like ADI, have limited velocity gri
target.X0              = [0;0;0;0]; % target true initial conditions in [x;xdot;y;ydot
target.track_seed      = [10];     % single value. [] sets random seed
target.q_truth         = 5e-4;

% Sensor image parameters
sensor.N               = 20;       % number of scenes
sensor.T               = 1;        % measurement update time
sensor.nxpix           = 128;      % measured scene size
sensor.nypix           = 128;
sensor.Pd              = 0;        % 0-unthresholded
sensor.snr_db          = [16,12,8]; % desired SNR in dB
sensor.image_seed      = [10];     % single value. [] sets random seed
% convert SNR in dB to power ratio
a = 10.^(sensor.snr_db./10);
sensor.lambda          = .5*(a+sqrt(a.^2+4*a));

%<<<<< USER MAY MODIFY Filters CELL TO ADD NEW TRACKER >>>>>
%Filters = {'adi','pf','user'};
Filters = {'pf','adi'};
```

Figure 3 – File GenMonteCarlo.m after editing

Then from the MATLAB command line type

MC = GenMonteCarlo(5);

If the c_flag.display flag is set to 1 then figures with images should appear on the screen simulating a sensor with a target under track. See Figures 1-2 for an example. Note that Figure 1 shows the particle values and a 1 sigma ellipse around the estimate. Figures 2 and 3 show the velocity pdf for the ADI method. The simulation will last twenty frames and then be complete.

To postprocess the data type:

PostProcess(MC);

For the multiple run case, the user is prompted to set an error threshold for each filter and SNR. The MATLAB command window prompts the user with the following message:

Set Outlier Threshold: Click at threshold and press ENTER

The user should place the cursor in the figure until the cross hairs appear as in Figure 5. Position the cursor to the desired threshold and left click the mouse button or press ENTER. If all of the data is to be used then click ENTER to bypass the data threshold.

3 RUN PROCEDURES

As a user you will be able to configure and run Monte Carlo simulations of existing nonlinear filters, examine results in a variety of ways, and postprocess ensembles of runs for evaluation and analysis. This section provides detailed instructions for using the TENET Simulator to perform Monte Carlo studies and postprocess their results.

3.1 Running TENET Simulator

From the MATLAB command prompt, move to the TENET directory (i.e. type **cd c:\TENET**). Then type the command

MC=GenMonteCarlo(50);

to generate 50 Monte Carlo runs. Parameter values, which are hardwired into m-file GenMonteCarlo.m, determine the course of the runs. The resulting data, which comprises a 50-run ensemble, is stored in a data structure named MC and an automatically generated file. The saved file name has the following format: MC*Date_Description_Count*.mat, where *Date* is in YYYYMMDD format and *Description* is the first letter in the Filters field (in GenMonteCarlo.m) and *Count* is a counter to distinguish the results of other simulations.

For a list of the elements of MC refer to Section 3.4.

3.2 Setting TENET Simulator Parameters

Listed below are the TENET Simulator parameters with their MATLAB variable names, in bold type, and range of values, if any, listed within brackets {}. These variables are listed in the file GenMonteCarlo.m and may be modified to run specific studies.

Target motion model

- Initial conditions, **XO {x,vx,y,vy}**^{T}
- Maximum target velocity, **max_vel**
- Constant velocity (CV) or constant acceleration (CA) Singer model, **truth_model {CV, CA}**
- Process noise variance for the target motion, **q_truth {>0, 5e-4 for near linear motion}**
- Probability of detection, **Pd {0 to 1}**
- SNR in dB, **snr_db**

Sensor image parameters

- Image size in pixels, x dimension: **nxpix**, y dimension: **nypix {64 to 1024}** *depending on run length and available memory*
- Number of scenes in the run, **N**
- Update rate, **T {positive integer}**.

Nonlinear Filtering

- Filter choice, Filters {'pf', 'adi'}
- Process noise variance, q {>0, 5e-4 for near linear motion} performance usually improves with q = 10*q_truth

3.3 The Interface for New Filters

The TENET Simulator is designed to accommodate new filtering methods in two "easy" Steps. Step 1 is to add new user track estimation code; this is by far the more demanding Step. The PF and ADI implementations are examples of appropriate code. Step 2 is to add the user filter name to the Filters list in the nonlinear filtering parameter initialization segment of file GenMonteCarlo.m. TENET will accommodate any number of trackers, as long as there are separate "init_" and "update_" files for each. Module GenMonteCarlo will run each one named in the Filters list. The remainder of this section describes the interface for any new track estimation code.

The first task in Step 1 is merely to choose the name of the new filter. Names previously used were pf and adi. Henceforth this name will be *user* for convenience.

The new user-supplied filter must perform two functions in two separate m-files. The first m-file initializes the new filter and must be named 'init_*user*.m'. The second m-file contains a function to perform time and measurement updating and is named 'update_*user*.m'. Each function must use the variables provided in the interface.

3.3.1 Init Function

The call from GenMonteCarlo to this function assumes this form:

[data] = init_*user* (X0, q, T, NumFrames);

The required inputs and outputs to **init_***user* are listed next.

Inputs:

- **X0** is the initial state estimate
- **q** is the modeled process noise
- **T** is the time between measurement updates
- **NumFrames** is the number of observations for the entire simulation. This is provided to save overall processing time by initializing parameters since it is costly in time resources to sequentially update the data structures in MATLAB.

Outputs:

All outputs are delivered in "**data**" which is a standard MATLAB data structure with the following required fields:

- **data.desc** is a string of characters to describe the nonlinear filtering method on post processing plots.
- **data.Xhat** is the state estimate [x;xdot;y;ydot], size is 4xNumFrames. Initialize using: data.Xhat = zeros(4,NumFrames);
- **data.index** indexes through the observations. Initialize to 0.
- **data.q** is the provided model of the process noise variance.

3.3.2 Update Function

The call from RunFilter to the "update" function assumes this form:

[data] = update_*user*(data, FOV, FOV_offset, T, XGT);

The required inputs and outputs to **update_***user* are listed next.

Inputs:

- **data** is the nonlinear filter data structure
- **FOV** is the field of view output of the imaging sensor. FOV size is determined by the imaging parameter in GenMonteCarlo.m.
- **FOV_offset** is the offset to control the sensor to move the image to keep the target in the FOV.
- **T** is the time between measurement updates
- **XGT** is the ground truth state estimate [x;xdot;y;ydot], size is 4xNumFrames.

Outputs:

- **data** is a MATLAB data structure with the following **required** fields:
- **data.desc** is a string of characters to describe the nonlinear filtering method on post processing plots.
- **data.Xhat** is the state estimate [x;xdot;y;ydot], size is 4xNumFrames. Initialize using: Data.Xhat = zeros(4,NumFrames);
- **data.Phat** is the covariance estimate.
- **data.index** indexes through the observations. Initialize to 0.
- **data.q** is the provided model of the process noise variance.

Optional data structure fields:

- **Pos_Density** is the estimate of the position density after update. Pos_Density is a three dimensional array where the size is a fixed density window size in each x and y by NumFrames.
- **Pos_DensityOffset** is the offset in x/y of the position density window, size is NumFrames by 2.
- **Vel_Density** is the estimate of the velocity density after update. Vel_Density is a three dimensional array where the size is a fixed velocity density window size in each x and y by NumFrames.
- **Vel_DensityOffset** is the offset in x/y of the velocity density window, size is NumFrames by 2.

Users should review init_pf.m, init_adi.m, update_pf.m and update_adi.m for examples of initialization and update functions.

3.4 Data Structure

Data structures are relaxed except for a few required fields to aid in the automation. The user is free to create any other fields deemed necessary. However, without user modification the intermediate data will not be saved.

The output data structure (MC) from the GenMonteCarlo function is of the following architecture:

MC.Results is an array (size = #SNR cases by #Monte Carlo runs) of each filter under test

3.5 Example Commands

Plot the true position in XY coordinates for the first SNR and the first simulation:

```
Plot(MC.Results(1,1).TrueTrack(1,:),MC.Results(1,1).TrueTrack(3,:))
```

Plot the position estimate from the ADI filter in XY coordinates for the first SNR and the first simulation:

```
Plot(MC.Results(1,1).adi.Xhat(1,:),MC.Results(1,1).adi.Xhat(3,:))
```

Plot the true velocity in XY coordinates for the first SNR and the first simulation:

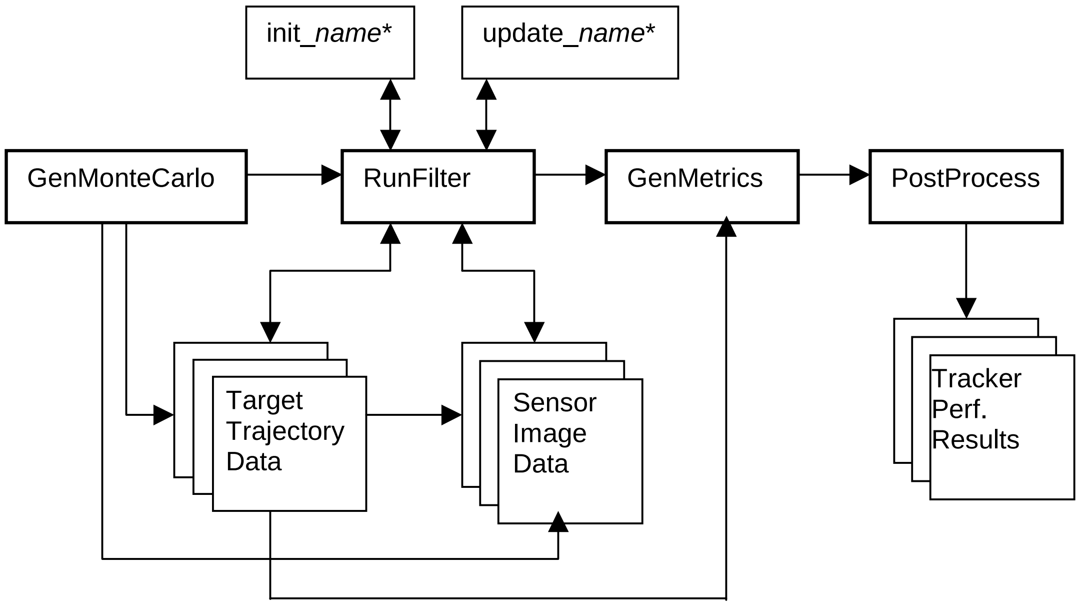
```
Plot(MC.Results(1,1).TrueTrack(2,:),MC.Results(1,1).TrueTrack(4,:))
```

Plot the velocity in XY

```
Plot(MC.Results(1,1).adi.Xhat(2,:),MC.Results(1,1).adi.Xhat(4,:))
```

4 TENET SOFTWARE ORGANIZATION

4.1 Software Flowchart



* *name* is identified in parameter
Filters, e.g. pf and adi.

4.2 MATLAB m-File Description

GenMonteCarlo: The function GenMonteCarlo initiates all calls to sub functions based on the user input parameters. The parameter list determines what scenarios will be run and in what order for compilation into tracker performance results.

5 METHOD

5.1 Bayesian Nonlinear Filtering

To track a target from a sequence of measurements $\mathbf{y}_k$ made at discrete times t_k, let the target dynamics be described by an Ito stochastic differential equation [3] for the time-dependent target state $\mathbf{x}_t$

$$d\mathbf{x}_t = \mathbf{f}(\mathbf{x}_t, t)dt + \mathbf{G}(\mathbf{x}_t, t)d\beta_t, \quad t \geq t \tag{1}$$

where $\mathbf{x}_t$ and $\mathbf{f}$ are n-vectors, $\mathbf{G}$ is an $n \times r$ matrix function, and $\{\beta_t, t \geq t_0\}$ is an r-vector Brownian motion process with $E\{d\beta_t d\beta_t^{\mathrm{T}}\} = \mathbf{Q}(t)dt$.

The observations up to time τ are denoted

$$Y_\tau = \{\mathbf{y}_l : t_l \leq \tau\} \tag{2}$$

Between observations the evolution of the conditional density is determined by the target dynamics as characterized by the Ito equation. The time evolution of the joint density between measurements is the solution to the Fokker-Planck equation (FPE)

$$\frac{\partial p}{\partial t}(\mathbf{x}_t \mid Y_{t_k}) = L(p), \quad t_k \leq t < t_{k+1}, \tag{3}$$

where

$$L(p) \equiv -\sum_{i=1}^{n} \frac{\partial(\mathbf{f}_i p)}{\partial \mathbf{x}_i} + \frac{1}{2}\sum_{i,j=1}^{n} \frac{\partial^2\left(p\left(\mathbf{GQG}^{\mathrm{T}}\right)_{ij}\right)}{\partial \mathbf{x}_i \partial \mathbf{x}_j} \tag{4}$$

with initial condition given by $p(\mathbf{x}_{t_k} \mid Y_{t_k})$.

Then given a new observation $\mathbf{y}_k$, the updated conditional density $p(\mathbf{x}_{t_k} \mid Y_{t_k})$ is obtained from the predicted density $p(\mathbf{x}_{t_k} \mid Y_{t_{k-1}})$ using Bayes' formula:

$$p(\mathbf{x}_{t_k} \mid Y_{t_k}) = \frac{p(\mathbf{y}_k \mid \mathbf{x}_{t_k})\, p(\mathbf{x}_{t_k} \mid Y_{t_{k-1}})}{\int p(\mathbf{y}_k \mid \mathbf{x}'_{t_k})\, p(\mathbf{x}'_{t_k} \mid Y_{t_{k-1}}) d\mathbf{x}'_{t_k}} \tag{5}$$

The minimum mean square error target state estimate $\hat{\mathbf{x}}_t$ is

$$\hat{\mathbf{x}}_t = \int \mathbf{x}_t \; p(\mathbf{x}_t \mid Y_t) d\mathbf{x}_t \qquad (6)$$

In this version of the challenge problem, we focus on the effect of measurement nonlinearity due to low SNR, and use a *linear* motion model, the so-called "nearly constant velocity" model with $\mathbf{x} = (x, \dot{x}, y, \dot{y})^{\mathrm{T}}$, $\mathbf{f}(\mathbf{x}) = (\dot{x}, 0, \dot{y}, 0)^{\mathrm{T}}$, and $\mathbf{G}(t)$ and $\mathbf{Q}(t)$ given by

$$\mathbf{G} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \mathbf{Q} = \begin{pmatrix} q & 0 \\ 0 & q \end{pmatrix}. \qquad (7)$$

The resulting FPE is

$$\frac{\partial p}{\partial t} = -\dot{x}\frac{\partial p}{\partial x} - \dot{y}\frac{\partial p}{\partial y} + \frac{q}{2}\frac{\partial^2 p}{\partial \dot{x}^2} + \frac{q}{2}\frac{\partial^2 p}{\partial \dot{y}^2} \qquad (8)$$

5.2 Sizing Signal-to-Noise Ratio

While signal-to-noise ratio (SNR) is often used to gauge likely performance for filter processes, it is only rigorously related to performance in the case of Gaussian signals. The Kullback-Leibler discrimination, a more general quantity that is related to detection and estimation performance, is defined by

$$L(q_0; q_1) \equiv \int q_0(y) \ln\big(q_0(y) / q_1(y)\big) dy \qquad (9)$$

Its symmetrized relative, the divergence, is defined by

$$\overleftrightarrow{L}(q_0; q_1) \equiv L(q_0; q_1) + L(q_1; q_0) \qquad (10)$$

The divergence is a convenient measure of effective SNR. For a Gaussian signal given by

$$p_1(y) = \frac{1}{\sqrt{2\pi\lambda}} \exp\big(-(y-\lambda)^2 / 2\lambda\big) \qquad (11)$$

and

$$p_0(y) = \frac{1}{\sqrt{2\pi\lambda}} \exp\big(-y^2 / 2\lambda\big) \qquad (12)$$

it is straightforward to find that

$$\overleftrightarrow{L}^{Gauss}(p_0\,;\,p_1)=\lambda \qquad (13)$$

For Rayleigh distributed measurements with

$$p_1(y)=\frac{y}{1+\lambda}\exp\left(-y^2/2(1+\lambda)\right)$$

$$p_0(y)=y\exp\left(-y^2/2\right)$$

we have this expression for the divergence:

$$\overleftrightarrow{L}^{Ray}(p_0;p_1)=\frac{\lambda^2}{1+\lambda}\,. \qquad (14)$$

Note that in the large λ limit, the Rayleigh and Gaussian divergences are the same, while they differ significantly for small values of λ.

5.3 Target Dynamics

We simulate target dynamics using an Ito stochastic differential equation [3] for the time-dependent target state $\mathbf{x}_t$

$$d\mathbf{x}_t=\mathbf{f}(\mathbf{x}_t,t)dt+\mathbf{G}(\mathbf{x}_t,t)d\beta_t\,,\quad t\geq t \qquad (15)$$

where $\mathbf{x}_t$ and $\mathbf{f}$ are n-vectors, $\mathbf{G}$ is an $n\times r$ matrix function, and $\{\beta_t, t\geq t_0\}$ is an r-vector Brownian motion process with $E\{d\beta_t d\beta_t^{\mathrm{T}}\}=\mathbf{Q}(t)dt$. We use a *linear* motion model, the so-called "constant velocity" model with $\mathbf{x}=(x,\dot{x},y,\dot{y})^{\mathrm{T}}$, $\mathbf{f}(\mathbf{x})=(\dot{x},0,\dot{y},0)^{\mathrm{T}}$,

$$\mathbf{G}=\begin{pmatrix}0&0\\1&0\\0&0\\0&1\end{pmatrix}\text{ and }\mathbf{Q}=\begin{pmatrix}q&0\\0&q\end{pmatrix}. \qquad (16)$$

5.4 Sensor Model

An example of how to incorporate the likelihood function into NLF is presented by pixelized data such as point-target image-tracking applications. Envelope-detected radar data takes a similar form. In this case an image contains M pixels labeled $i=1,\ldots,M$. The measurement consists of the pixel output vector $\mathbf{y}_k=[y_{k,1},\ldots,y_{k,M}]^{\mathrm{T}}$ where the $y_{k,i}$ can be positive, real or complex, depending on the details of the sensor and signal processing. For point targets, pixel outputs are conditionally independent and

$$p(\mathbf{y}_k \mid \mathbf{x}_k) = \prod_{i=1}^{M} p(y_{k,i} \mid \mathbf{x}_k) \tag{17}$$

When there is no target in a pixel, its output statistics are determined by the background rate $p_0(y_{k,i})$. Depending on the nature of the imager, this may be modeled by Rayleigh, Poisson, or more complicated distributions. Further, it may vary with pixel index and time, depending on the clutter statistics. When the target projects into a pixel i, its output statistics will be given by $p_1(y_{k,i})$, which, again, may be pixel and time dependent and will depend on the detailed nature of the sensor and target.

Using Bayes' formula directly to update the density is cumbersome because it involves M factors in the product for each discretized value of the target state vector $\mathbf{x}_{t_k}$. This can be simplified by defining the target-space to pixel-space projection:

$$h_i(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} \text{ projects to pixel } i \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

This projection is a binary mapping from the target state to the pixel space. Note that this is a highly nonlinear function of the target state. For Doppler-sensitive sensors such as radar or ladar, this projection will depend on the target velocity as well as its location. Using the target-space to pixel-space projection,

$$\begin{aligned} p(\mathbf{y}_k \mid \mathbf{x}_k) &= \prod_{i=1}^{M} \left(p_0(y_{k,i}) + h_i(\mathbf{x}_k)\left(p_1(y_{k,i}) - p_0(y_{k,i}) \right) \right) \\ &= \kappa p_1(y_{k,i_{\mathbf{x}_k}}) / p_0(y_{k,i_{\mathbf{x}_k}}) \end{aligned} \tag{19}$$

where $i_{\mathbf{x}}$ is the target-containing pixel (i.e. $h_{i_{\mathbf{x}_k}}(\mathbf{x}_k) = 1$) and

$$\kappa = \prod_{i=1}^{M} p_0(y_{k,i}) \tag{20}$$

is a constant that can be dropped in the Bayes' formula update. With this expression only the measurement likelihood ratio

$$l(y_{k,i_{\mathbf{x}_k}}) = p_1(y_{k,i_{\mathbf{x}_k}}) / p_0(y_{k,i_{\mathbf{x}_k}}) \tag{21}$$

needs to be evaluated for each discretized value of the target state vector $\mathbf{x}_k$, a significant saving in computation.

To illustrate, for a Rayleigh target with SNR parameter λ, the probability distribution for intensity $y_{i_{\mathbf{x}_k}}$ of the target-containing pixel is

$$p_1(y_{i_{\mathbf{x}_k}}) = \frac{y_{i_{\mathbf{x}_k}}}{1+\lambda}\exp\left(-y_{i_{\mathbf{x}_k}}{}^2/2(1+\lambda)\right) \tag{22}$$

The distribution for the background pixels is given by the same expression with $\lambda = 0$. A little algebra shows that up to an irrelevant constant that can be dropped, the likelihood is

$$l(y_{i_{\mathbf{x}_k}}) = \exp\left(\frac{\lambda y_{i_{\mathbf{x}_k}}{}^2}{2(1+\lambda)}\right) \tag{23}$$

5.5 Algorithm for Particle Filter Method

Particle methods are a collection of Monte Carlo techniques in which the probability density is represented by a collection of N independent and identically distributed random samples, $\{\mathbf{x}_k^{(i)}; i = 1,\dots,N\}$. The samples $\mathbf{x}_k^{(i)}$, referred to as particles, are distributed according to $\mathbf{x}_k^{(i)} \sim p(\mathbf{x}_k \mid Y_k)$. Then the conditional likelihood is approximated by

$$p(\mathbf{x}_k \mid Y_k) \approx \frac{1}{N}\sum_{i=1}^{N}\delta\left(\mathbf{x}_k - \mathbf{x}_k^{(i)}\right) \tag{24}$$

where $\delta(\mathbf{x})$ is the Dirac δ-function in target state space.

To implement a particle filter using the technique called Sampling, Importance, Resampling (SIR), we mechanize prediction and measurement update using the approximating density above. For prediction at time k, we propagate each particle $\mathbf{x}_{k-1}^{(i)}$ by producing a single draw from $p\left(\mathbf{x}_{k+1} \mid \mathbf{x}_k^{(i)}\right)$. The predicted particles are denoted $\mathbf{x}_k^{(i)}$ and the predicted density is approximated by

$$p(\mathbf{x}_k \mid Y_{k-1}) \approx \frac{1}{N}\sum_{i=1}^{N}\delta\left(\mathbf{x}_k - \mathbf{x}_k^{(i)}\right) \tag{25}$$

To perform measurement update, we can approximate (up to a normalization constant κ)

$$
\begin{aligned}
p(\mathbf{x}_k \mid Y_k) &\approx \kappa p(y_k \mid \mathbf{x}_k) \sum_{i=1}^{N} \delta\left(\mathbf{x}_k - \mathbf{x}_k^{(i)}\right) \\
&= \kappa \sum_{i=1}^{N} p\left(y_k \mid \mathbf{x}_k^{(i)}\right) \delta\left(\mathbf{x}_k - \mathbf{x}_k^{(i)}\right)
\end{aligned}
\tag{26}
$$

where each particle is weighted by its measurement likelihood $p\left(y_k \mid \mathbf{x}_k^{(i)}\right)$. In the so-called Sequential Importance Sampling (SIS), this process is simply iterated. The weight for particle i after K steps is $w_K^i \propto \prod_{k=1}^{K} p(y_k \mid \mathbf{x}_k^{-(i)})$. SIS is not a very effective algorithm since the particle trajectories are not influenced by the measurements at all. As a result, they quickly diffuse away from the region of high likelihood and very poor estimation performance results.

This problem with SIS is corrected in the Sampling Importance Resampling (SIR) algorithm. Here a new set of particles is generated from the predicted set by resampling with likelihood $p\left(y_k \mid \mathbf{x}_k^{(i)}\right)$. Thus particles with high likelihood are sampled many times while particles with low likelihood are unlikely to appear in the resampled set. With this, we have the SIR algorithm:

Particle Filter Algorithm (SIR)

1. *Initialization*, $k = 0$:

   ♦ For $i = 1, \ldots, N$ sample $\mathbf{x}_0^{(i)} \sim p(\mathbf{x}_0)$ and set $k = 1$.

2. *Prediction step:*

   ♦ For $i = 1, \ldots, N$ sample $\tilde{\mathbf{x}}_k^{(i)} \sim p\left(\mathbf{x}_k \mid \mathbf{x}_{k-1}^{(i)}\right)$.

3. *Resampling step:*

   ♦ Compute importance weights, $w_k^{(i)} = p(y_k \mid \tilde{\mathbf{x}}_k^{(i)})$.

   ♦ Normalize the weights according to

   $w_k^{\max} = \max_i w_k^{(i)}$; $w_k^{(i)} \leftarrow w_k^{(i)} / w_k^{\max}$.

   ♦ Generate N resampled particles $\left\{\mathbf{x}_t^{(i)}; i = 1, \ldots, N\right\}$

   with $\Pr\left(\mathbf{x}_k^{(i)} = \tilde{\mathbf{x}}_k^{(j)}\right) = \frac{1}{N} w_k^{(j)}$.

   ♦ Set $k \leftarrow k+1$ and go to Step 2.

5.6 Algorithm for ADI Method

For the finite difference filter, we used the Alternating Direction Implicit (ADI) scheme to solve the Fokker-Planck Equation. Space and time are discretized on a uniform grid with time resolution Δt and spatial resolution $\Delta\mathbf{x} = (\Delta x, \Delta\dot{x}, \Delta y, \Delta\dot{y})^{\mathrm{T}}$. In this subsection p denotes a continuum solution to the FPE while g denotes a function defined on the grid that approximates p. Defining the sub-operators

$$A_1 = -\dot{x}\frac{\partial}{\partial x} \tag{27}$$

$$A_2 = \frac{q}{2}\frac{\partial^2}{\partial\dot{x}^2} \tag{28}$$

$$A_3 = -\dot{y}\frac{\partial}{\partial y} \tag{29}$$

$$A_4 = \frac{q}{2}\frac{\partial^2}{\partial\dot{y}^2} \tag{30}$$

the FPE can be written as

$$\frac{\partial p}{\partial t} = \sum_i A_i p \tag{31}$$

Discretizing in time but not in space, for now, we abbreviate $p^k = p(\mathbf{x}, t_k)$. The implicit Euler scheme [9] for the FPE is obtained by using a Taylor series in time for $p(\mathbf{x}, t_k + \Delta t)$, leading to

$$\frac{p^{k+1} - p^k}{\Delta t} = \sum_i A_i p^{k+1} + O(\Delta t) \tag{32}$$

Rearranging terms leads to

$$\left(1 - \Delta t \sum_i A_i\right) p^{k+1} = p^k + O\left((\Delta t)^2\right) \tag{33}$$

In principle, this expression can be solved for p^{k+1} by inverting the operator $1 - \Delta t \sum_i A_i$, but direct inversion is computationally expensive. An expression that is equivalent to $O\left((\Delta t)^3\right)$ but much simpler to invert is obtained by using the operator identity

$$\prod_i (1 - \Delta t A_i) \cong 1 - \Delta t \sum_i A_i + \Delta t^2 \sum_{i<j} A_i A_j \tag{34}$$

It can be shown that, if $A_{i\Delta\mathbf{x}}$ is an $O(\Delta\mathbf{x})$ discretization of A_i, then

$$\prod_i (1-\Delta t A_{i\Delta\mathbf{x}})p^{k+1} \cong p^k \tag{35}$$

and the grid function g defined by

$$\prod_i (1-\Delta t A_{i\Delta\mathbf{x}})g^{k+1} = g^k \tag{36}$$

approximates p up to the error terms proportional to $O(\Delta\mathbf{x})+O(\Delta t)$ (higher order approximations can be constructed, at some additional computational cost).

To propagate the density, we solve Eq. (36) for g^{k+1} using

$$g^{k+i/4} = (1-\Delta t A_{i\Delta\mathbf{x}})^{-1} g^{k+(i-1)/4} A, i = 1,\ldots,4 \tag{37}$$

The essential point to note is that each factor $(1-\Delta t A_{i\Delta\mathbf{x}})$ in Eq. (36) is inverted separately, simplifying the calculation. To discretize A_i, abbreviate $q(k\Delta t, x\pm\Delta x, \dot{x}, y, \dot{y}, \omega) = q_{x\pm1}$ with similar definitions for $q_{\dot{x}\pm1},\ldots,q_{\omega\pm1}$. Using upwind differencing for the first order spatial derivatives, we have

$$A_{1\Delta\mathbf{x}} g = -\frac{\dot{x}}{\Delta x}\begin{cases} q_x - q_{x-1}, & \dot{x} > 0 \\ q_{x+1} - q_x, & \dot{x} < 0 \end{cases} \tag{38}$$

$$A_{2\Delta\mathbf{x}} q = \frac{q}{2\Delta\dot{x}^2}(q_{\dot{x}+1} - 2q + q_{\dot{x}-1}) \tag{39}$$

with similar expressions for $A_{3\Delta\mathbf{x}}$ and $A_{4\Delta\mathbf{x}}$. With this discretization, each operator $(1-\Delta t A_{i\Delta\mathbf{x}})$ is tridiagonal and can be inverted using Thomas's algorithm [9].

To completely specify the FPE solution, it must be restricted to a finite domain leading to an initial-boundary value problem. The finite grid domain consists of the points $((i+i_0)\Delta x, (j+j_0)\Delta\dot{x}, (k+k_0)\Delta y, (l+l_0)\Delta\dot{y})^{\mathrm{T}}$, $i = 0,\ldots,N_x$, $j = 0,\ldots,N_{\dot{x}}$, $k = 0,\ldots,N_y$, $l = 0,\ldots,N_{\dot{y}}$, where $i_0,\ldots,m_0$ are offsets used to translate the origin. The total number of grid nodes is $(N_x+1)(N_{\dot{x}}+1)(N_y+1)(N_{\dot{y}}+1)$ while the number of unknowns is $N = (N_x-1)(N_{\dot{x}}-1)(N_y-1)(N_{\dot{y}}-1)$. Boundary conditions must be specified on this hyper-cube to determine the solution to the FPE uniquely. We assume that the target signal-to-noise ratio is sufficiently high that the target has been localized. Then the density will be concentrated in some small region and decay exponentially far from this region. We assumed that the grid was large enough that the density was small on its boundary. With this motivation we used a homogenous Dirichlet condition with the solution held at 0 on the boundary.

To reduce the size of the grid required to represent the target joint density and thereby save computations, the grid was translated after each measurement to approximately maintain the target's location near its center. After each measurement update the target position estimate $(\hat{x}_t, \hat{y}_t)$ was evaluated and the grid was shifted to center of the grid near this position. This was achieved by placing the lower left corner of the spatial grid at (i_0, j_0) where

$$i_0 = \left[\hat{x}/\Delta x - N_x/2\right] \tag{40}$$

$$j_0 = \left\lfloor\hat{y}/\Delta y - N_y/2\right\rfloor \tag{41}$$

and $[x]$ denotes rounding to the nearest integer. This always translated the grid by an integral multiple of $(\Delta x, \Delta y)$. Grid nodes outside the intersection of the original and translated grids were set to 0.

5.7 Metrics

The RMS values of position and velocity error were chosen as metrics to illustrate performance in this paper. Error is defined as the difference between estimate and truth, $\mathbf{e}_t = \hat{\mathbf{x}}_t - \mathbf{x}_t$, where the estimate $\hat{\mathbf{x}}_t$ is computed as the mean of the estimated density $p(\mathbf{x}_t \mid \mathbf{Y}_t)$. Although RMS accuracy alone does not provide a thorough characterization of performance in a general tracking problem, it would seem to be appropriate in this special case where there is just one target (the usual multi-track metrics degenerate or disappear) and track state initialization is nearly perfect (e.g. convergence times would be artificially small).

As noted previously, it is expensive and unnecessary to compute the joint density over the entire range of motion. Instead, a computational gate is established on which to produce a solution, this gate being a small fixed-size subset of the motion region. The gate must be translated using estimates of target motion as inputs to a translation control algorithm. If the filter estimate drifts so far from the truth that the target exits the gate, *lost lock* occurs. With target observations lost, the filter diverges quickly and seldom recovers. The gate translation control problem was challenging anytime filter estimates were inaccurate, e.g. at startup or at the lowest SNRs. Lost lock events were logged and their frequency computed.

5.8 Experiments

The estimation problem is to track a single dim target moving in a 2D space using intensity images of the track area for measurements. True target motion is generated using a nearly constant velocity (NCV) model of target dynamics, a model based on the assumption that acceleration is a white noise process, $a(t) = w(t)$. The white noise in the NCV model imposes randomness in the motion so that a different true trajectory is produced on every Monte Carlo run. Target motion is represented in each filter by the same NCV model. This decision to match filter to truth avoids most mis-modeling issues. For the results that follow, even the noise strength of the filter was matched to the truth.

The simulated sensor images the entire target motion region to produce a scene of 256 x 256 pixels. The intensity in each pixel is governed by a Rayleigh distribution with noise power one, Eq. (22). In the pixel holding the target, the intensity is adjusted for the SNR of the study, Eq. (21). Identical simulated sensor images are input to each filter as measurements, but only the portion of the scene in the instantaneous gate contributes to the joint density estimate.

The initial state of each filter is chosen to approximately match the truth. The initial density of each filter is uniform in each of the four dimensions $(x, \dot{x}, y, \dot{y})$, and extends over the space in the initial gate. For the results that follow, the gate was fixed at 10x10 pixels in (x, y) space.

In the case of a lost lock event, accuracy degrades precipitously and the run is effectively ruined. When this occurs, data from that run is removed from the study ensemble, and a new run is made to replace the spoiled one.

Experimental results are based on studies of 50 Monte Carlo runs each. Studies were conducted for ADI and PF separately, at 2 dB intervals in the range 4-20 dB effective SNR, Eq. (23). Altogether, 18 studies (9 each for PF and ADI) contributed to the results reported next.

6 REFERENCES

[1] *Workshop on Nonlinear Filtering Methods for Tracking*, Proceedings on Veridian CD: DAAA01C13D, sdms_help@mbvlab.wpafb.af.mil, Dayton Ohio, 21-22 Feb 2001.

[2] Musick, Greenewald, Kreucher, and Kastella, "Comparison of Particle Method and Finite Difference Nonlinear Filters for Low SNR Target Tracking", Conference Proceedings of *Fusion2001*, 6-10 August 2001.

[3] Musick, Greenewald, Kastella, and Kreucher, "Comparing Numerical Methods for Nonlinear Tracking Applications", *Defense Applications of Signal Processing 2001 Workshop*, held in Adelaide, Australia from 16-20 September 2001.

[4] A. H. Jazwinski, Stochastic Processes and Filtering Theory, Academic Press, New York, 1970.