|  |
| --- |
| [ASEN 5044]  Statistical Estimation of Dynamical Systems Progress Report 2 Fall 2020  Group Members:  Joshua Malburg  Junior Sundar |

# Question #4

## Implementing Linearized Kalman Filter (LKF)

To implement the LKF for this Cooperative Localization problem we define an LKF function which performs the prediction [Equation ( 1 )] and correction [Equation ( 2 )] step.

The function takes in the ground truth values (state dynamics and measurements), the nominal state and measurement trajectories without process or measurement noise, the process noise covariance () for tuning, and initial values for and . The function outputs the state estimates (), measurement estimates and estimation errors ( and ).

|  |  |  |
| --- | --- | --- |
|  |  | ( 1 ) |

|  |  |  |
| --- | --- | --- |
|  |  | ( 2 ) |

This function is included in a Monte Carlo run which feeds the ground truth models, nominal state and measurement trajectories, and the DT state space matrices linearized around the nominal trajectories () according to the methods from Question #2 and Question #3.

The truth model is a simulated run of the nonlinear model with the process and measurement noise generated using the covariances uploaded on Canvas. We seed for the ground truth values for every Monte Carlo run from and then and then for every subsequent values the process noise is obtained from covariance matrix ‘Qtrue’.

## Truth Model Testing (TMT) for LKF

For the TMT we do 50 Monte Carlo () runs. This is an appropriate number of Monte Carlo runs as we will have enough data sets for performing an unbiased NEES and NIS test.

Additionally, for the multiple runs in the TMT we feed the LKF the following initial values:

|  |  |  |
| --- | --- | --- |
|  |  | ( 3 ) |

These initial conditions were selected for the following reasons: we are given the nominal and truth state trajectories, and hence the initial perturbation can be derived from those values; furthermore, we have a sufficient degree of certainty of our initial state perturbation such that the state perturbation covariance is finite ().

In addition to this, in the LKF, the covariance matrix for the measurement noise () is set to be the one uploaded on canvas, i.e. Rtrue. As that information is generally known to us from the sensors being used.

## Tuning

There are two aspects that we need to keep in mind while tuning the Kalman filter. First, we need to make sure that the Kalman filter ‘works properly’, i.e. the estimate error averages at 0, and the bounds are converging. Second, we also look at the NEES and NIS chi-square tests and make sure they are within the confidence intervals.

We calculate the confidence intervals on MATLAB with significance level . We chose this significance level to provide a less stringent condition for proving whether the LKF is doing its job (i.e. having a low enough false-alarm probability). This was because we found that the LKF is not good enough for estimating this nonlinear system. The bounds are calculated using the MATLAB function ‘chi2inv’. As a result, for Monte Carlo runs, states, measured values, the chi-square confidence bounds are:

|  |  |  |
| --- | --- | --- |
|  |  | ( 4 ) |

While tuning the LKF we vary our predicted process noise covariance matrix () until the two conditions in the beginning of this subsection are satisfied. We first start with and run the truth model tests and check if the KF works and the NEES and NIS tests are satisfied. The result showed that the neither of the conditions were met, hence we moved to then to and tweaked the main diagonal parameters in this way until the conditions were sufficiently satisfied.

In the end, the process noise covariance matrix for the LKF that best satisfied the conditions was found to be:

|  |  |  |
| --- | --- | --- |
|  |  | ( 5 ) |

Before we move forward, there are some points that need to be highlighted regarding the result. First, the NEES test COULD NOT be satisfied for any selection of regardless of the magnitude of the covariance matrix or selection of initial conditions. The KF was debugged thoroughly, and we could not find any method to alleviate this problem. Second, the NIS test could be satisfied, however periodically at every ~25 seconds the chi-square values exceed the bounds. Although, we managed to bring most of the points for within the bounds in Equation ( 5 ). Finally, it was also found that regardless of the tuning methods employed, the state estimate errors corresponding to North position and East position of the UGV (i.e. and ) are unstable and their bounds do not converge.

The best results obtained after tuning the LKF are shown in the following plots. Figure 1 is a plot showing the NEES and NIS chi-square tests for the Monte Carlo runs with the LKF. Figure 2 is a plot of the state estimate errors from a sample Monte Carlo run with LKF under the same conditions and Figure 3 shows the actual state estimates and the state ground truths for the very same sample run.

First looking at Figure 1, we can see that the NIS test is sufficiently satisfied except for every ~25 seconds where the points are outside the bounds. Upon further investigation, this was found due to the covariance matrix used for the LKF. In the matrix, the elements directly corresponding to the aircraft’s North position and East position values are greater than the other elements that are a combination of the UGV’s and UAV’s together. For instance, elements . And the UAV has a control input of which corresponds to a period of 25 seconds. Furthermore, the NEES test is not satisfied and the problem could not be alleviated regardless of the tuning method. This issue was found to be because the LKF could not accurately ascertain the state evolution of the UGV ( and ).

Also note, that the state estimate errors for and are sufficiently low, except of the time points where the state peaks, and the state estimation covariance () also converges for the UAV’s states. The estimate errors for and also tend to increase at points where the angles are wrapped to within the bounds.

Referring to Figure 2 and Figure 3, we can see that and estimates are barely close in periodic intervals. The LKF fails in estimating these two states. Furthermore, another observation that is that the covariance for the state estimates (i.e. ) for and are oscillating and not converging. Selecting a that alleviates this problem affected the other states adversely, and hence it was left this way. The oscillation also has a period of ~25 seconds and the bounds for is out of phase with .

## Comment on LKF for Estimating Nonlinear System

The tuning procedure was arduous and complicated. However, after multiple attempts, we can conclude that the LKF is not a good enough filter to estimate the nonlinear system for cooperative localization.

We derived this conclusion because we were not able to tune the LKF to satisfy the two conditions: the LKF ‘works properly’ and satisfies the NEES and NIS chi-square tests.

This section went into the methodology used to implement the LKF and the process used to tune it. We also went into the issues faced when tuning and the compromises and assumptions made.

The LKF failed the NEES test and barely passed the NIS test, indicating that the state estimates obtained from this filter are not close enough to the ground truths. This was even after multiple attempts in tuning the filter.

Referring to the UAV’s states, it seems that the LKF starts to mis-estimate its states when the aircraft is moving near the boundaries of its domain in and . However, at all other points, the LKF provides a decent estimate for the UAV’s state. However, the LKF completely fails to successfully estimate the states for UGV. This is credited to the fact that we can directly observe the UAV’s states, while the UGV’s states are only derived indirectly through trigonometric equations.

The LKF fails to estimate the states and for an extended periods and this in and of itself disqualifies the LKF.

Finally, the bounds, or covariance, for and fail to converge over time, which also disqualifies the LKF. We tried to fix this problem by modifying the initial conditions and elements, however in the cases where we got the bounds to converge, the bounds for and exhibit the same oscillating property, and the NEES and NIS tests were worse off. Hence we selected the conditions in Equation ( 3 )( 4 )( 5 ) as a compromise.

Finally, we also tested the filter by checking whether angle wrapping between operations effected or improved the results (i.e. wrapping the nominal and ground truths before feeding into LKF). The results were slightly worse off, because the filter would fail to estimate the states at the point where the angle wrapped back to within the intervals. Hence angles are not wrapped until after the estimate is obtained through the filter.

The MATLAB codes for the LKF implementation, the Monte Carlo runs, and the NEES and NIS tests are attached in Appendix D.

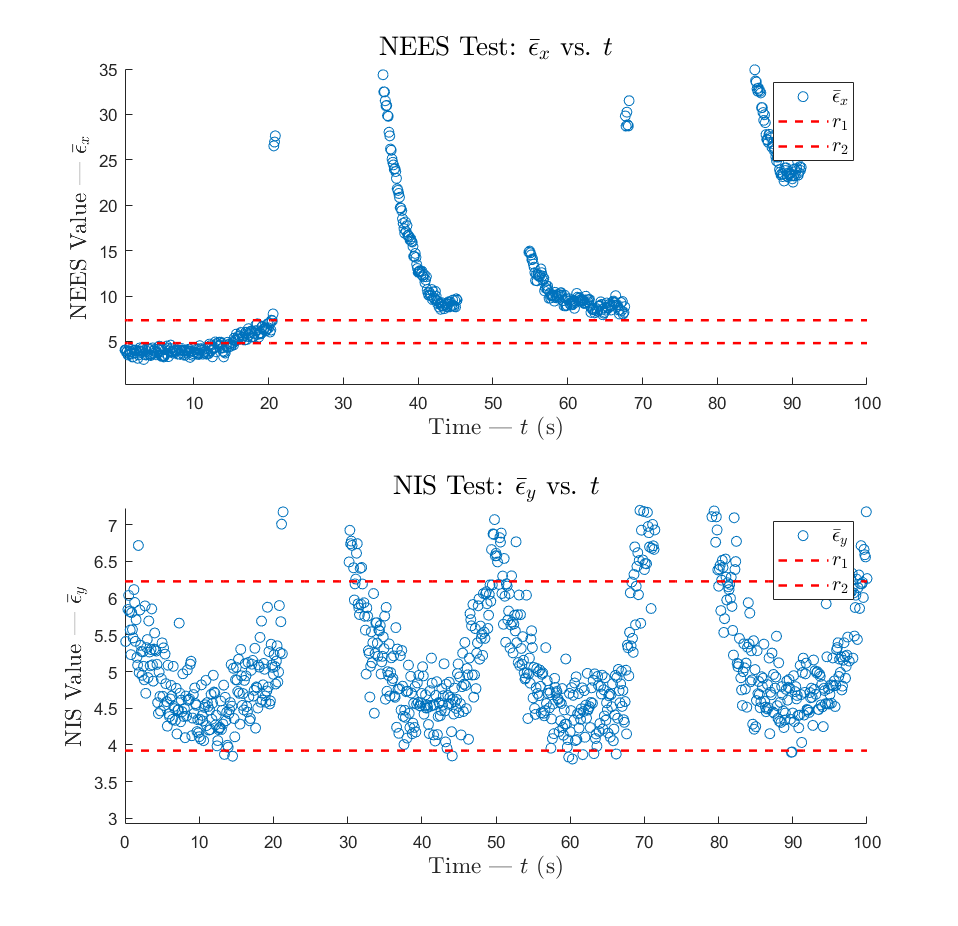


Figure 1 – NEES and NIS chi-square test for LKF Monte Carlo runs with Equations ( 3 )( 4 )( 5 ) applied

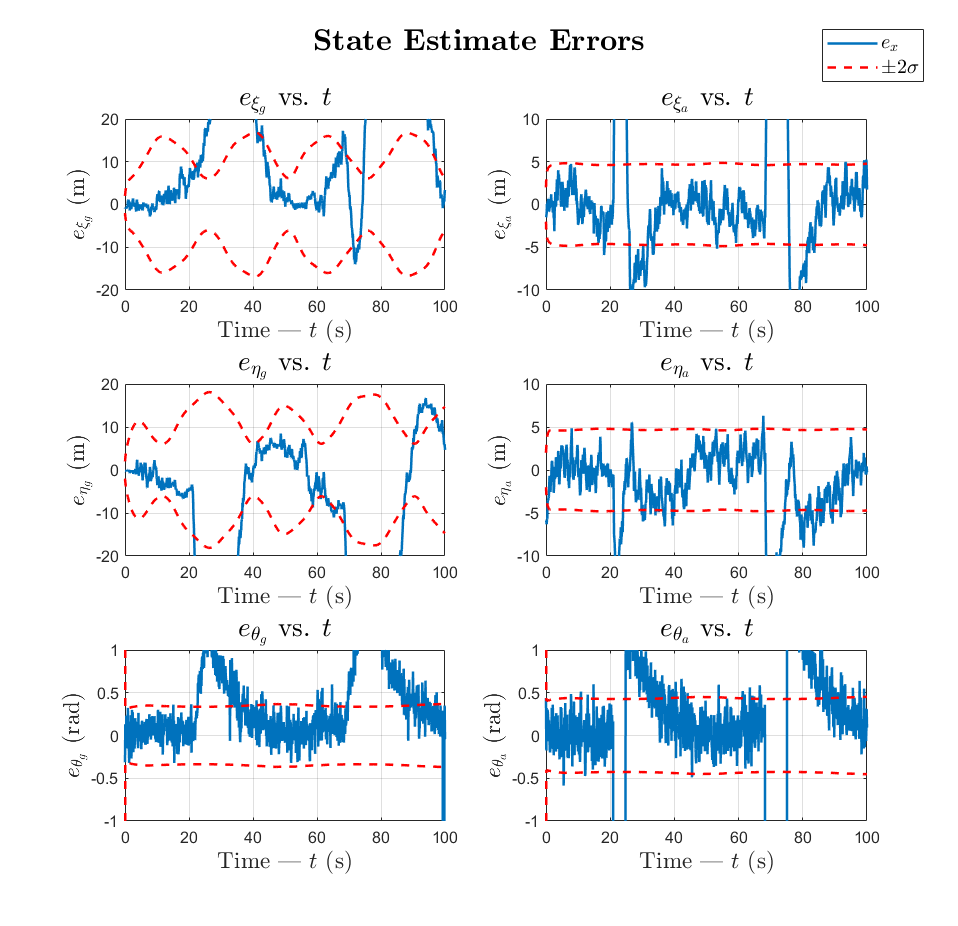


Figure 2 – State estimate errors and bounds of sample LKF Monte Carlo run with Equations ( 3 )( 4 )( 5 ) applied

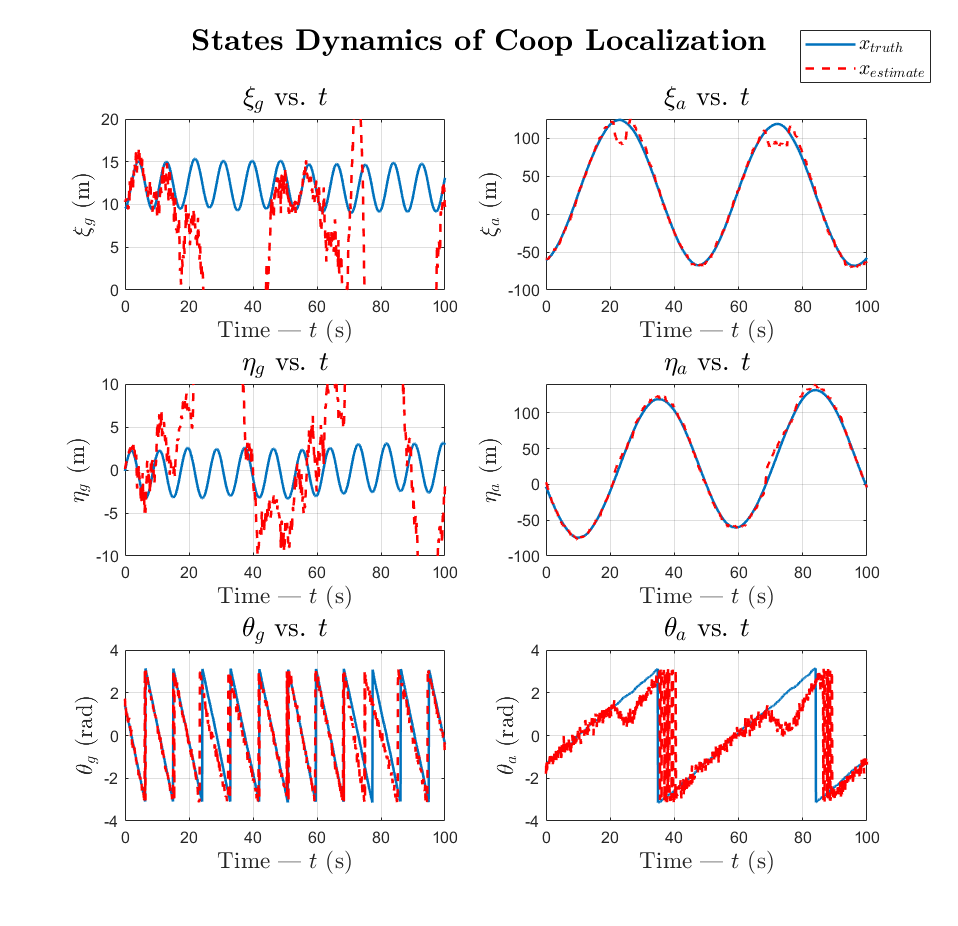


Figure 3 – State estimates and state ground truth of sample LKF Monte Carlo run with Equations ( 3 )( 4 )( 5 ) applied

# Question #5

## Extended Kalman Filter (EKF)

The EKF starts off at time-step zero (0) with an estimate of the total state and covariance. At each time-step k, during the prediction step the EKF uses the non-linear equations to calculate the estimated state which in turn is used to approximate the predicted covariance matrix.

where

During the correction step the EKF calculates the innovation vector () using the measurement estimate derived from the non-linear measurement equation.

The filter then generates a new linearized DT System (H) matrix, linearized about the new state estimate, to calculate the Kalman gain, update the total state estimate and update the covariance matrix.

MATLAB code for implementation and tuning of the EKF can be found in Appendix E.

## Tuning Approach

Tuning the filter involved generating randomized ground truth data from many Monte Carlo simulations and calculating the mean NEES and NIS values at each time step. The EKF’s process noise matrix Q was then tweaked until the NEES and NIS consistency tests were within the desired accuracy bounds (alpha = ±5%). The initial state was also randomized to demonstrate the filter wasn’t biased to a specific nominal trajectory.

## Typical Simulation Plots

The section shows plots from a typical simulation run. Figure 4captures an example of the randomly generated UGV and UAV state data.



Figure 4 - Simulation Run - Ground Truth States

Figure 5 shows an example of the randomly generated noisy measurement data.



Figure 5 - Simulation Run - Simulated Measurement Data



Figure 6 – Estimated State Errors



Figure 7 - Estimated Measurement Errors

## Chi-Square Test Results

This section shows the final NEES and NIS test results.



Figure 8 - Final NEES results



Figure 9 - Final NIS Results

**Appendix D – Codes accompanying Question 4**

|  |
| --- |
| function [x\_est,y\_est,dx,P,S,e\_x,e\_y] = LKF(dx\_init,P\_init,x\_nom,y\_nom,x,y,Fk,Hk,Ok,Q,R)  n = size(Fk(:,:,1),2);  p = size(Hk(:,:,1),1);  steps = length(x\_nom(1,:))-1;  dx = zeros(n,steps+1);  P = zeros(n,n,steps+1);  S = zeros(p,p,steps+1);  K = zeros(n,p,steps+1);  dx(:,1) = dx\_init;  dy = y-y\_nom;  P(:,:,1) = P\_init;  I = eye(n);  e\_y = zeros(p,steps+1);    e\_y(:,1) = y(:,1) - Hk(:,:,1)\*(x\_nom(:,1) + dx(:,1));    y\_est = zeros(5,steps+1);  y\_est(:,1) = Hk(:,:,1)\*(x\_nom(:,1) + dx(:,1));    S(:,:,1) = Hk(:,:,1)\*P(:,:,1)\*Hk(:,:,1)' + R;  K(:,:,1) = P(:,:,1)\*Hk(:,:,1)'\*inv(S(:,:,1));    for i=1:steps  % Prediction Step  dx(:,i+1) = Fk(:,:,i)\*dx(:,i);  P(:,:,i+1) = Fk(:,:,i)\*P(:,:,i)\*Fk(:,:,i)' + Ok(:,:,i)\*Q\*Ok(:,:,i)';  % Correction Step  S(:,:,i+1) = Hk(:,:,i+1)\*P(:,:,i+1)\*Hk(:,:,i+1)' + R;  K(:,:,i+1) = P(:,:,i+1)\*Hk(:,:,i+1)'\*inv(S(:,:,i+1));    y\_est(:,i+1) = Hk(:,:,i+1)\*(x\_nom(:,i+1) + dx(:,i+1));  e\_y(:,i+1) = y(:,i+1) - y\_est(:,i+1);    dx(:,i+1) = dx(:,i+1) + K(:,:,i+1)\*(dy(:,i+1) - Hk(:,:,i+1)\*dx(:,i+1));  P(:,:,i+1) = (I - K(:,:,i+1)\*Hk(:,:,i+1))\*P(:,:,i+1);  end    x\_est = x\_nom + dx;    e\_x = x - x\_est;    end |
| function [x\_truth,y\_truth,x\_est,y\_est,dx,P,S,e\_x,e\_y] = LKF\_MonteCarlo(Q,R,steps)  load("cooplocalization\_finalproj\_KFdata.mat");    x\_nom = [10 0 pi/2 -60 0 -pi/2]';  u\_nom = [2 -pi/18 12 pi/25]';  x\_pert = [0 1 0 0 0 0.1]';  Dt = 0.1;    n = size(x\_nom,1);  P0 = eye(n).\*0;    t = 0:Dt:steps\*Dt;    % Generate truth model outputs for nominal trajectories  [x\_truth, y\_truth] = GenerateTruth(x\_nom, u\_nom, P0, Qtrue, Rtrue, Dt, steps);    % Generate nominal trajectories  [~,x\_NL] = ode45(@(t,x) NL\_DynModel(t,x,u\_nom,zeros(6,1)),t,x\_nom);  x\_NL = x\_NL';  y\_NL = zeros(5,length(t));  for i=1:length(t)  y\_NL(:,i) = NL\_MeasModel(x\_NL(:,i),zeros(5,1));  end    % Generate DT matrices along nominal trajectory  x\_nominal = x\_NL;  y\_nominal = y\_NL;    Fk = zeros(6,6,length(t));  Hk = zeros(5,6,length(t));  Ok = zeros(6,6,length(t));    for i=1:length(t)  [A\_t, B\_t, C\_t] = Linearize(x\_nominal(:,i),u\_nom);  [Fk(:,:,i), ~ , Hk(:,:,i)] = Discretize(A\_t,B\_t,C\_t, Dt);  Ok(:,:,i) = eye(6);  end    % Run LKF on Data  dx\_init = x\_pert;  P\_init = eye(6);    [x\_est,y\_est,dx,P,S,e\_x,e\_y] = LKF(dx\_init,P\_init,x\_nominal,y\_nominal,x\_truth,y\_truth,Fk,Hk,Ok,Q,R);  end |
| clc  clear  load('cooplocalization\_finalproj\_KFdata.mat')  seed = 100;  rng(seed);  Dt = 0.1;  steps = 1000;  n = 6; p = 5; t = 0:Dt:steps\*Dt;    N = 50; % No. of Monte Carlo runs  NEES\_all = zeros(N,steps+1);  NIS\_all = zeros(N,steps+1);    % Tuning parameters  % Q(1,1) = 0.5;  % Q(2,2) = 0.5;  % Q(3,3) = 5;  % Q(4,4) = 5;  % Q(5,5) = 5;  % Q(6,6) = 5;    Q(1,1) = 1;  Q(2,2) = 1;  Q(3,3) = 3;  Q(4,4) = 1;  Q(5,5) = 1;  Q(6,6) = 3;  % Q = 100\*eye(6);    R = Rtrue;    for i=1:N  disp(i)  NEES = zeros(1,steps+1);  NIS = zeros(1,steps+1);    [x\_truth,y\_truth,x\_est,y\_est,dx,P,S,e\_x,e\_y] = LKF\_MonteCarlo(Q,R,steps);    for k=1:steps+1  NEES(:,k) = e\_x(:,k)'\*inv(P(:,:,k))\*e\_x(:,k);  NIS(:,k) = e\_y(:,k)'\*inv(S(:,:,k))\*e\_y(:,k);  end    NEES\_all(i,:) = NEES;  NIS\_all(i,:) = NIS;  end    NEES\_bar = zeros(1,steps+1);  NIS\_bar = zeros(1,steps+1);    for i=1:steps+1  NEES\_bar(1,i) = mean(NEES\_all(:,i));  NIS\_bar(1,i) = mean(NIS\_all(:,i));  end    alpha = 0.01;  rx1 = (chi2inv(alpha/2,N\*n))./N;  rx2 = (chi2inv(1-alpha/2,N\*n))./N;  ry1 = (chi2inv(alpha/2,N\*p))./N;  ry2 = (chi2inv(1-alpha/2,N\*p))./N; |

**Appendix E – Codes accompanying Question 5**

|  |
| --- |
| % Question 5 - Tune EKF  close all, clearvars, clc  load("cooplocalization\_finalproj\_KFdata.mat");    x0 = [10 0 pi/2 -60 0 -pi/2]';  u0 = [2 -pi/18 12 pi/25]';  Dt = 0.1;  n = size(x0,1);  steps = 1000;  seed = 100;  rng(seed);    Q = diag([.0015, .0015, 0.01, 0.001, 0.005, 0.01]);  P0 = diag([1 1 0.025 1 1 0.025]);    runs = 100;  EX = zeros(n, steps+1, runs);  p = 5;  EY = zeros(p, steps+1, runs);  PS = zeros(n, n, steps+1, runs);  SS = zeros(p, p, steps+1, runs);  fig1 = figure(1);  fig2 = figure(2);  fig3 = figure(3);  fig4 = figure(4);  enablePlotDuring = true;  for run = 1:runs  disp(['run #', num2str(run)]);    % generate truth for run  [x, y] = GenerateTruth(x0, u0, P0, Qtrue, Rtrue, Dt, steps, true);  t = (0:(length(x)-1))\*Dt;    % assume we can get exact measurement noise from  % specifications of sensors  R = Rtrue;    % Run filter for all time-steps of run #k  [x\_est, y\_est, P, S] = EKF(x0, P0, u0, y, Q, R, Dt);    % wrap angle diff too!!  ex = x - x\_est;  ex(3,:) = angdiff(x\_est(3,:),x(3,:));  ex(6,:) = angdiff(x\_est(6,:),x(6,:));  ey = y - y\_est;  ey(1,:) = angdiff(y\_est(1,:),y(1,:));  ey(3,:) = angdiff(y\_est(3,:),y(3,:));    % Plot error during monte carlo runs  if enablePlotDuring == true  PlotStates(fig1,t,ex, ['State Errors, Run ',num2str(run)], P);  PlotMeasurements(fig2,t,y,'Ground Truth Measurements');  PlotStates(fig3,t,x,'Ground Truth States');  PlotMeasurements(fig4,t,ey,['Ground Truth Measurement Errors, Run ',num2str(run)]);  end    % save run data from NEES/NIS tests  EX(:, :, run) = ex;  EY(:, :, run) = ey;  PS(:, :, :, run) = P;  SS(:, :, :, run) = S;  end    %% Calculate NEES and NIS statistics  [NEES\_bar, NIS\_bar] = CalcStats(EX, EY, PS, SS);    %--------------------------------------------------------------------------  % Plots for (a)  PlotStates(fig1,t,x - x\_est, ['State Errors, Run ',num2str(run)], P);  PlotMeasurements(fig2,t,y,'Ground Truth Measurements');  PlotStates(fig3,t,x,'Ground Truth States');  PlotMeasurements(fig4,t,y - y\_est,['Ground Truth Measurement Errors, Run ',num2str(run)]);    %--------------------------------------------------------------------------  % Plots for (b)  fig5 = figure(5);  alpha = 0.05;  PlotNees(fig5, NEES\_bar, runs, n, alpha);  %--------------------------------------------------------------------------  % Plots for (c)  fig6 = figure(6);  PlotNis(fig6, NIS\_bar, runs, p, alpha); |
| function [x\_est, y\_est, P, S] = EKF(x0, P0, u, y, Q, R, Dt)    % set simulation tolerances for ode45  opt = odeset('RelTol',1e-6,'AbsTol',1e-6);    n = size(x0, 1); % number of states  p = size(R, 1); % number of measurements  steps = size(y,2); % number of time steps    x\_est = zeros(n, steps); % state estimate vector  y\_est = zeros(p, steps); % measurement estimate vector  P = zeros(n, n, steps); % coveriance  S = zeros(p, p, steps);    % start with initial estimate of total state  % and covariance  x\_p = x0;  P\_p = P0;    for i=1:steps  %----------------------------------------------------------  % Prediction Step  % use full NL equations to estimate state at next time step  % using state at previous time step; since Wk is AWGN,  % its expected value is zero, set input to zero  wk = zeros(1,n);  [~, x\_m] = ode45(@NL\_DynModel, [0.0 Dt], x\_p', opt, u', wk);  x\_m = x\_m(end,:)';    x\_m(3) = wrapToPi(x\_m(3));  x\_m(6) = wrapToPi(x\_m(6));    % to calculate covariance, linearize "online"  % about current state estimate  [A\_t,B\_t,C\_t] = Linearize(x\_m, u);  [F, ~, H] = Discretize(A\_t, B\_t ,C\_t, Dt);  P\_m = F\*P\_p\*F' + Q;    x\_m(3) = wrapToPi(x\_m(3));  x\_m(6) = wrapToPi(x\_m(6));    % uese estimated state from NL ODEs; since Wk is AWGN,  % its expected value is zero, set to zero  vk = zeros(p,1);  y\_est(:,i) = NL\_MeasModel(x\_m, vk);  y\_est(1,i) = wrapToPi(y\_est(1,i));  y\_est(3,i) = wrapToPi(y\_est(3,i));    % calculate innovation vector  e\_y = y(:,i) - y\_est(:,i);  e\_y(1) = wrapToPi(e\_y(1));  e\_y(3) = wrapToPi(e\_y(3));    %----------------------------------------------------------  % Correction Step  % calculate gain using linearized measurement  % matrix H and covariance from Prediction Step  S\_p = H\*P\_m\*H' + R;  K = P\_m\*H'/S\_p;    % calculate posterior state estimate and covariance  x\_p = x\_m + K\*e\_y;  P\_p = (eye(n) - K\*H)\*P\_m;    x\_p(3) = wrapToPi(x\_p(3));  x\_p(6) = wrapToPi(x\_p(6));    % for each time-step, save estimate and covariance  x\_est(:,i) = x\_p;  P(:,:,i) = P\_p;  S(:,:,i) = 0.5\*(S\_p + S\_p');  end    end |
| function [x, y] = GenerateTruth(x0, u, P0, Q, R, Dt, steps, wrapOn)  opt = odeset('RelTol',1e-6,'AbsTol',1e-6);  useChol = true;  n = size(x0,1);  p = size(R,1);    x = zeros(n,steps+1);  y = zeros(p,steps+1);    % set initial conditions  dx = mvnrnd(zeros(1,n),P0);  x(:,1) = x0 + dx';  x(3,1) = wrapToPi(x(3,1));  x(6,1) = wrapToPi(x(6,1));    for i = 2:steps+1    % generate noisy state  if useChol==true  wk = chol(Q)\*randn(n,1);  else  wk = mvnrnd(zeros(1,n),Q)';  end  [~,next\_x] = ode45(@NL\_DynModel, [0 Dt], x(:,i-1)', opt, u', wk);    if wrapOn == true  % wrap angle to [-pi pi]  next\_x(3) = wrapToPi(next\_x(3));  next\_x(6) = wrapToPi(next\_x(6));  end  x(:,i) = next\_x(end,:)';  end    for i = 1:steps+1  % generate noisy measurement  if useChol==true  vk = chol(R)\*randn(p,1);  else  vk = mvnrnd(zeros(1,p),R)';  end  y(:,i) = NL\_MeasModel(x(:,i), vk);    if wrapOn == true  % wrap angle to [-pi pi]  y(1,i) = wrapToPi(y(1,i));  y(3,i) = wrapToPi(y(3,i));  end  end    end |
| function [NEES, NIS] = CalcStats(EX, EY, P, S)    steps = size(EX, 2);  runs = size(EX, 3);    NEES\_all = zeros(runs,steps);  NIS\_all = zeros(runs,steps);  NEES = zeros(1,steps);  NIS = zeros(1,steps);    for run=1:runs  for step=1:steps  NEES(step) = EX(:,step, run)' / P(:,:,step, run) \* EX(:,step, run);  NIS(step) = EY(:,step, run)' / S(:,:,step, run) \* EY(:,step, run);  end    NEES\_all(run,:) = NEES;  NIS\_all(run,:) = NIS;  end    % calculate mean at each time step  for i=1:steps  NEES(i) = mean(NEES\_all(:,i));  NIS(i) = mean(NIS\_all(:,i));  end  end |
| function PlotMeasurements(hdl,t,y, title)  figure(hdl)  ftSize = 10;  sgtitle(title,'FontSize',ftSize+2,'Interpreter','latex')  subplot(3,2,1)  plot(t,y(1,:))  ylabel('$\gamma\_{ag}$ (rad)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(y(1,:)) max(y(1,:))])  grid on    subplot(3,2,2)  plot(t,y(2,:))  ylabel('$\rho\_{ga}$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(y(2,:)) max(y(2,:))])  grid on    subplot(3,2,3)  plot(t,y(3,:))  ylabel('$\gamma\_{ga}$ (rad)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(y(3,:)) max(y(3,:))])  grid on    subplot(3,2,4)  plot(t,y(4,:))  ylabel('$\xi\_a$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(y(4,:)) max(y(4,:))])  grid on    subplot(3,2,[5,6])  plot(t,y(5,:))  ylabel('$\eta\_a$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(y(5,:)) max(y(5,:))])  grid on  end |
| function PlotNees(hdl, epsNEESbar, Nsimruns, n, alpha)    figure(hdl);  Nnx = Nsimruns\*n;    %%compute intervals:  r1x = chi2inv(alpha/2, Nnx )./ Nsimruns;  r2x = chi2inv(1-alpha/2, Nnx )./ Nsimruns;    figure(hdl)  plot(epsNEESbar,'ro','MarkerSize',6,'LineWidth',2),hold on  plot(r1x\*ones(size(epsNEESbar)),'r--','LineWidth',2)  plot(r2x\*ones(size(epsNEESbar)),'r--','LineWidth',2)  ylabel('NEES statistic, $\bar{\epsilon}\_x$','FontSize',14,'Interpreter','latex')  xlabel('time step, k','FontSize',14)  title('NEES Estimation Results','FontSize',14)  legend('NEES @ time k', 'r\_1 bound', 'r\_2 bound')  end |
| function PlotNis(hdl, epsNISbar, Nsimruns, p, alpha)    figure(hdl);    Nny = Nsimruns\*p;    %%compute intervals:  r1y = chi2inv(alpha/2, Nny )./ Nsimruns;  r2y = chi2inv(1-alpha/2, Nny )./ Nsimruns;    plot(epsNISbar,'bo','MarkerSize',6,'LineWidth',2),hold on  plot(r1y\*ones(size(epsNISbar)),'b--','LineWidth',2)  plot(r2y\*ones(size(epsNISbar)),'b--','LineWidth',2)  ylabel('NIS statistic, $\bar{\epsilon}\_y$','FontSize',14,'Interpreter','latex')  xlabel('time step, k','FontSize',14)  title('NIS Estimation Results','FontSize',14)  legend('NIS @ time k', 'r\_1 bound', 'r\_2 bound')  end |
| function PlotStates(hdl,t,x, title, P)    if nargin > 4  p = zeros(size(x));  for ind = 1:size(x,2)  p(:,ind) = 2\*sqrt(diag(P(:,:,ind)));  end  displayError = true;  else  displayError = false;  end    figure(hdl)  ftSize = 10;  sgtitle(title,'FontSize',ftSize+2,'Interpreter','latex')  subplot(3,2,1)  state = 1;  plot(t,x(state,:))  if displayError == true  hold all, plot(p(state,:),'b--'), plot(-p(state,:),'b--'), hold off  end  ylabel('$\xi\_g$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(x(state,:)) ...  max(x(state,:))])  grid on    subplot(3,2,3)  state = state + 1;  plot(t,x(state,:))  if displayError == true  hold all, plot(p(state,:),'b--'), plot(-p(state,:),'b--'), hold off  end  ylabel('$\eta\_g$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(x(state,:)) ...  max(x(state,:))])  grid on    subplot(3,2,5)  state = state + 1;  plot(t,wrapToPi(x(state,:)))  if displayError == true  hold all, plot(p(state,:),'b--'), plot(-p(state,:),'b--'), hold off  end  ylabel('$\theta\_g$ (rad)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(wrapToPi(x(state,:))) ...  max(wrapToPi(x(state,:)))])  grid on    subplot(3,2,2)  state = state + 1;  plot(t,x(state,:))  if displayError == true  hold all, plot(p(state,:),'b--'), plot(-p(state,:),'b--'), hold off  end  ylabel('$\xi\_a$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(x(state,:)) ...  max(x(state,:))])  grid on    subplot(3,2,4)  state = state + 1;  plot(t,x(state,:))  if displayError == true  hold all, plot(p(state,:),'b--'), plot(-p(state,:),'b--'), hold off  end  ylabel('$\eta\_a$ (m)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(x(state,:)) ...  max(x(state,:))])  grid on    subplot(3,2,6)  state = state + 1;  plot(t,wrapToPi(x(state,:)))  if displayError == true  hold all, plot(p(state,:),'b--'), plot(-p(state,:),'b--'), hold off  end  ylabel('$\theta\_a$ (rad)','FontSize',ftSize,'Interpreter','latex')  xlabel('Time(s)','FontSize',ftSize,'Interpreter','latex')  axis([min(t) max(t) min(wrapToPi(x(state,:))) ...  max(wrapToPi(x(state,:)))])  grid on  end |