“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

Introduction

The purpose of this document and the accompanying Buffmo.py program is to demonstrate how we might implement a Kalman Filter, including truth modeling and consistency tests, when given a dynamical model (or equations of motion) and some model for measurements taken on the dynamics. The problem used will be a minor extension of a simple example from lecture, but the basic steps should be more or less the same when implementing a filter for other, more complicated systems.

Problem Description

Here’s the scenario: several of your classmates were so inspired by the 1D robotic cart system introduced in lecture that they decided to construct their very own autonomous “Buffmo” cart modeled after Ralphie the Bison. Unfortunately, one of the students accidentally submitted a set of model inputs they were simulating to the Buffmo control bus and Robo-Ralphie promptly sprang into action, separating itself from its control station. Even more unfortunately, the Buffmo team had yet to implement remote controls for the autonomous bison, which is now rampaging around campus! They’ve enlisted your help to set up a Kalman filter that estimates Buffmo’s position relative to recent locations reported by terrified students, given the known input model and noisy measurements provided by a small drone operated by the Buffmo team.

"Buffmo", the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

Problem Setup

If we adopt a coordinate system that describes Buffmo's position in terms of easting and northing positions $\boldsymbol{x} = [\xi \quad \dot{\xi} \quad \eta \quad \dot{\eta}]^T$ relative to its last reported location, then the simple kinematic equations of motion for the cart are given by:

$$\dot{x}_1 = \dot{\xi} = x_2, \qquad \dot{x}_3 = \dot{\eta} = x_4$$

$$\dot{x}_2 = \ddot{\xi} = u_1 + \widetilde{w}_1, \qquad \dot{x}_4 = \ddot{\eta} = u_2 + \widetilde{w}_2, \qquad \widetilde{w}_i \sim \mathcal{N}\left(\mathbf{0}, q_W \cdot \begin{bmatrix} 1.0 & 0.1 \\ 0.1 & 1.0 \end{bmatrix}\right) = \mathcal{N}(\mathbf{0}, W)$$

where $\widetilde{w}_i$ are AWGN accelerations due to unpredictable dynamics (e.g., due to gravity on an incline) and the input model submitted to the cart is of the form:

$$u_1(t) = f\cos(\Omega t), \qquad u_2(t) = -f\sin(\Omega t)$$

The CT LTI dynamics for this system are therefore described by:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \widetilde{w}_1 \\ \widetilde{w}_2 \end{bmatrix}$$

$$= A\boldsymbol{x}(t) + B\boldsymbol{u}(t) + \Gamma\widetilde{\boldsymbol{w}}(t)$$

Thanks to the forms of the A and B matrices, conversion to DT is straightforward:

$$\hat{A} = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \Longrightarrow e^{\hat{A}\Delta t} = \begin{bmatrix} F & G \\ 0 & I \end{bmatrix}$$

$$F = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad G = \begin{bmatrix} 0.5 \cdot \Delta t^2 & 0 \\ \Delta t & 0 \\ 0 & 0.5 \cdot \Delta t^2 \\ 0 & \Delta t \end{bmatrix}$$

$$\boldsymbol{x}_{k+1} = F\boldsymbol{x}_k + G\boldsymbol{u}_k + \boldsymbol{w}_k, \qquad \boldsymbol{w}_k \sim \mathcal{N}(\mathbf{0}, Q)$$

where the process noise covariance matrix Q can be found using van Loan's method:

$$Z = \Delta t \cdot \begin{bmatrix} -A & \Gamma W \Gamma^T \\ 0 & A^T \end{bmatrix} \Longrightarrow e^Z = \begin{bmatrix} \ldots & F^{-1}Q \\ 0 & F^T \end{bmatrix}$$

Observations provided by the drone are just direct but noisy measurements of position:

$$\boldsymbol{y}_k = \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \boldsymbol{v}_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \boldsymbol{v}_k = H\boldsymbol{x}_k + \boldsymbol{v}_k$$

$$\boldsymbol{v}_k \sim \mathcal{N}\left(\mathbf{0}, q_R \cdot \begin{bmatrix} 1.0 & 0.1 \\ 0.1 & 1.0 \end{bmatrix}\right) = \mathcal{N}(\mathbf{0}, R)$$

Note that, for this model, the number of dynamical degrees of freedom is $n = 4$ and the number of measurement degrees of freedom is $p = 2$.

In order to start implementing this problem in code, we require some numerical values for the parameters used in this section. To start with, let’s use the following:

Parameter	Δt	t_{max}	Ω	f	q_W	q_R
Value	$0.1\ s$	$100\ s$	$0.75\ s^{-1}$	$0.2\ m\ s^{-2}$	1	1

Table 1: Parameters for problem setup

Note that the q_W parameter provides us a way of tuning the process noise covariance Q.

```
13  # Define parameters
14  tmax = 100
15  dt = 0.1
16  nt = int(tmax/dt)
17  Om = .75
18  f = .2
19  qW = 1
20  qR = 1
21
22  # Define or compute DT inputs & matrices
23  ts = array([dt*t for t in range(1,nt+1)])
24  u = array([f*cos(Om*ts),-f*sin(Om*ts)]).T
25  A = array([[0,1,0,0],[0,0,0,0],[0,0,0,1],[0,0,0,0]])
26  F = array([[1,dt,0,0],[0,1,0,0],[0,0,1,dt],[0,0,0,1]])
27  G = array([[.5*dt**2,0],[dt,0],[0,.5*dt**2],[0,dt]])
28  Gam = array([[0,0],[1,0],[0,0],[0,1]])
29  W = qW*array([[1,.1],[.1,1]])
30  Z = dt*block([[-A,Gam@W@Gam.T],[zeros((4,4)),A.T]])
31  Zhat = expm(Z)
32  Q = F@Zhat[:4,4:]
33  H = array([[1,0,0,0],[0,0,1,0]])
34  R = qR*array([[1,.1],[.1,1]])
```

Figure 1: Python implementation of problem setup

“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

Truth Modeling

In order to ensure the functionality of our Kalman filter, we’ll need to perform many rounds of truth modeling, filtering, and NEES and NIS consistency tests on the results. Note that this means the code excerpts in this and the next section are actually nested within a for-loop over these rounds (see the final section for a full program).

Initialization

Each round of testing must begin with the selection of some initial true state which we’ll then update and perform simulated measurements on according to the equations in the previous section. There aren’t many generally applicable rules governing this selection, but we’d like to ensure that our simulated dynamics are well varied and representative of possible real-world outcomes. To that end, let’s draw our initial truth quantities randomly from uniform distributions with some reasonable ranges:

$$\xi_0, \eta_0 \sim \mathcal{U}(-10, 10), \qquad \dot{\xi}_0, \dot{\eta}_0 \sim \mathcal{U}(-2, 2)$$

That is, we’ll assume Buffmo is within $10\ m$ (in either direction) of where it was last spotted and has velocity components between -2 and $2\ m/s$.

Updates

Having chosen a programmatic way of initializing our truth data, we now need only to update the state according to our dynamics:

$$\boldsymbol{x}_{k+1} = F\boldsymbol{x}_k + G\boldsymbol{u}_k + \boldsymbol{w}_k$$

and simulate measurements according to our measurement model:

$$\boldsymbol{y}_k = H\boldsymbol{x}_k + \boldsymbol{v}_k$$

```
42      # Simulate a set of noisy truth data & measurements
43      xT = zeros((nt+1,4))
44      ySim = zeros((nt,2))
45      xT[0] = array([uni(-10,10),uni(-2,2),uni(-10,10),uni(-2,2)])
46      for t in range(nt):
47          xT[t+1] = F@xT[t]+G@u[t]+mvn(zeros((4)),Q)
48          ySim[t] = H@xT[t+1]+mvn(zeros((2)),R)
```

Figure 2: Python implementation of truth modeling

“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

Kalman Filter

Initialization

Since our only information regarding Buffmo’s position comes from the reports of terrified students (which defines the origin for coordinate system), and we have no a priori information regarding Buffmo’s velocity at all, it is logical for us to simply select the zero vector as our initial estimate. However, we still have no way of predicting how Buffmo may have moved prior to the arrival of the drone, and we should therefore initialize our filter with a relatively large-valued estimation error covariance matrix:

$$\boldsymbol{x}_0 \sim \mathcal{N}(\mathbf{0}, 1 \times 10^3 \cdot I_{4\times4}) = \mathcal{N}(\hat{\boldsymbol{x}}_0^+, P_0^+)$$

Updates

Once the initial estimate and covariance matrix have been defined, we just go through the typical KF update steps, beginning with our prediction step:

$$\hat{\boldsymbol{x}}_k^- = F\hat{\boldsymbol{x}}_{k-1}^+$$
$$P_k^- = FP_{k-1}^+F^T + Q$$
$$K_k = P_k^-H(HP_k^-H^T + R)^{-1}$$

and ending with our update step:

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + K_k(\boldsymbol{y}_k - H\hat{\boldsymbol{x}}_k^-)$$
$$P_k^+ = (I_{4\times4} - K_kH)P_k^-$$

NEES & NIS

The calculation of average NEES and NIS statistics is covered in the next section, but first we need to compute the statistics at every time step in each of the truth model runs. This is best handled alongside the filter updates since many of the constituent quantities for NEES and NIS are shared with the update process. In fact, the only quantity we need that wasn’t already used in the update section is the estimation error, $\boldsymbol{e}_{\boldsymbol{x},k}$. For the sake of clarity, we’ll still define each quantity used for NEES and NIS separately here, beginning with the estimation and measurement errors:

$$\boldsymbol{e}_{\boldsymbol{x},k} = \boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^+$$
$$\boldsymbol{e}_{\boldsymbol{y},k} = \boldsymbol{y}_k - H\hat{\boldsymbol{x}}_k^-$$

Note that the $\boldsymbol{e}_{\boldsymbol{y},k}$ is still calculated relative to the prediction estimate $\hat{\boldsymbol{x}}_k^-$. The next quantity, the effective measurement covariance matrix S_k, was also used to compute K_k:

$$S_k = HP_k^- H^T + R$$

Finally, we compute the covariance weighted square errors for estimates $\boldsymbol{\epsilon}_{\boldsymbol{x},k}$ (NEES) and measurements $\boldsymbol{\epsilon}_{\boldsymbol{y},k}$ (NIS) at this time step and round of modeling:

$$\boldsymbol{\epsilon}_{\boldsymbol{x},k} = \boldsymbol{e}_{\boldsymbol{x},k}^T (P_k^+)^{-1} \boldsymbol{e}_{\boldsymbol{x},k}$$

$$\boldsymbol{\epsilon}_{\boldsymbol{y},k} = \boldsymbol{e}_{\boldsymbol{y},k}^T (S_k)^{-1} \boldsymbol{e}_{\boldsymbol{y},k}$$

```
50      # Initialize the Kalman filter
51      xh = zeros((nt+1,4))
52      P = zeros((nt+1,4,4))
53      xh[0] = array([0,0,0,0])
54      P[0] = 1e3*eye(4)
55
56      # Perform Kalman filter updates, compute NEES/NIS statistics
57      for t in range(nt):
58
59          xhm = F@xh[t]
60          Pm = F@P[t]@F.T+Q
61          Kk = Pm@H.T@inv(H@Pm@H.T+R)
62          xh[t+1] = xhm+Kk@(ySim[t]-H@xhm)
63          P[t+1] = (eye(4)-Kk@H)@Pm
64
65          xerr = xT[t+1]-xh[t+1]
66          yerr = ySim[t]-H@xhm
67          Sk = H@Pm@H.T+R
68          NEES[n,t] = xerr.T@inv(P[t+1])@xerr
69          NIS[n,t] = yerr.T@inv(Sk)@yerr
```

Figure 3: Python implementation of the Kalman filter

“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

Consistency Tests

Average NEES & NIS

Once N rounds of testing have been conducted ($N = 50$ here), we should have an $N \times n_k$ array of NEES and NIS statistics where $n_k = t_{max}/\Delta t$ is the number of time steps in each round. In order to compute the average statistics $\bar{\epsilon}_{x,k}$ and $\bar{\epsilon}_{y,k}$, we just take the average of that array over rounds of testing, leaving an $n_k \times 1$ array of weighted averages of squared estimation and measurement errors at each time step.

NEES & NIS Confidence Bounds

What makes the NEES and NIS so useful is that they’re proportional to the square magnitude of what should be Gaussian random vectors (the errors $\boldsymbol{e}_{x,k}$ and $\boldsymbol{e}_{y,k}$), meaning they should be χ^2 distributed. To test whether this is the case (and thereby test the consistency of our filtering algorithm), we choose a confidence interval and check whether $\bar{\epsilon}_{x,k}$ and $\bar{\epsilon}_{y,k}$ fall between the boundaries of that interval for an appropriate percentage of time steps. The interval is set by the parameter α (here $\alpha = 0.05$), such that we expect $(1-\alpha)\cdot 100\%$ of the statistics to fall between the bounds r_1 and r_2, with $(\alpha/2)\cdot 100\%$ expected to fall into each of the distribution tails to the left of r_1 or right of r_2. To compute these bounds, we therefore need to evaluate the *inverse* CDF (sometimes called the *percent point function* or PPF) of the χ^2 distribution to determine at what points the CDF evaluates to $(\alpha/2)$ and $(1-\alpha/2)$. The parameters of the PPF are these likelihoods and the number of degrees of freedom in the distribution ($N \cdot n = 4N$ for the NEES and $N \cdot p = 2N$ for the NIS here). Finally, we must normalize our bounds for our average statistics by dividing them by the number of testing rounds N.

```
71  # Compute NEES/NIS test quantities
72  alpha = 0.05
73  exbar = NEES.mean(axis=0)
74  eybar = NIS.mean(axis=0)
75  rNEES = array([chi2.ppf(alpha/2,nTMT*4),chi2.ppf(1-alpha/2,nTMT*4)])/nTMT
76  rNIS = array([chi2.ppf(alpha/2,nTMT*2),chi2.ppf(1-alpha/2,nTMT*2)])/nTMT
```

Figure 4: Python implementation of consistency tests

Results

Buffmo Trajectory Estimates

A plot of one model Buffmo trajectory with simulated measurements and Kalman filter estimates is shown in Figure 5. Running the program multiple times (or plotting trajectories for multiple rounds of testing) demonstrates how much random variation can appear in the outcomes of our truth model. This is ideal for ensuring that, if deemed consistent, our filter is robust enough to handle the much less predictable variations that are bound to exist in any real dynamical system.

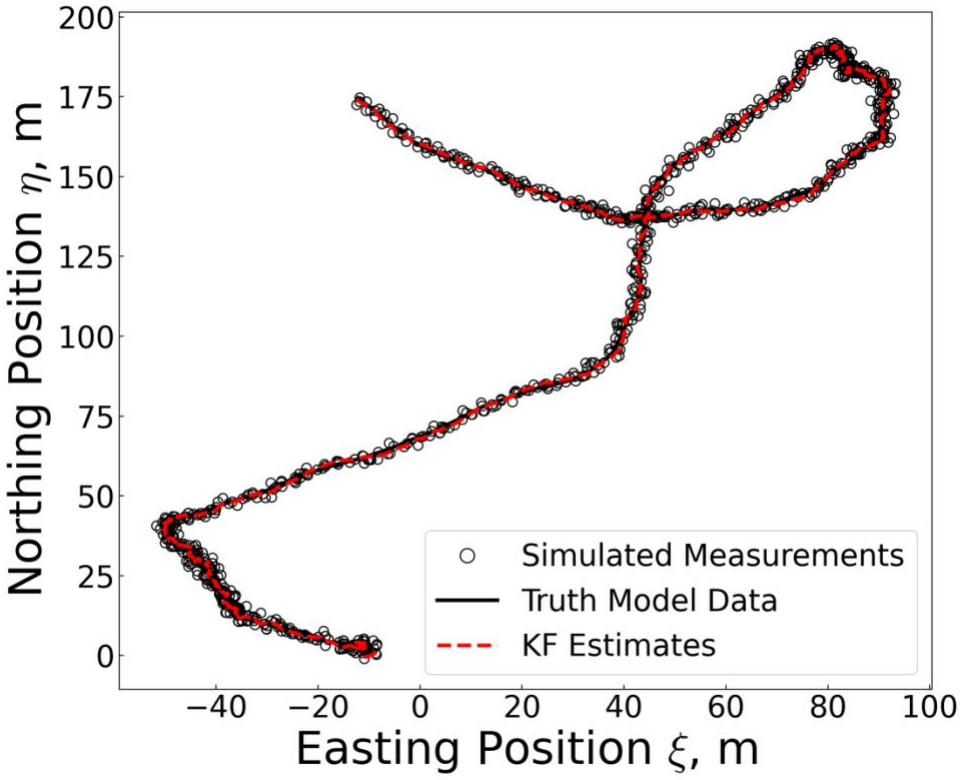

Figure 5: An example Buffmo trajectory with measurements and estimates.

Estimation Errors

A plot of one model set of estimation errors and 2σ error bounds for each state variable is shown in Figure 6. As we would hope, the majority of errors over time fall within the error bounds. As expected, the velocity estimates (for which we get no direct measurements) exhibit higher levels of uncertainty than the position estimates.

“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

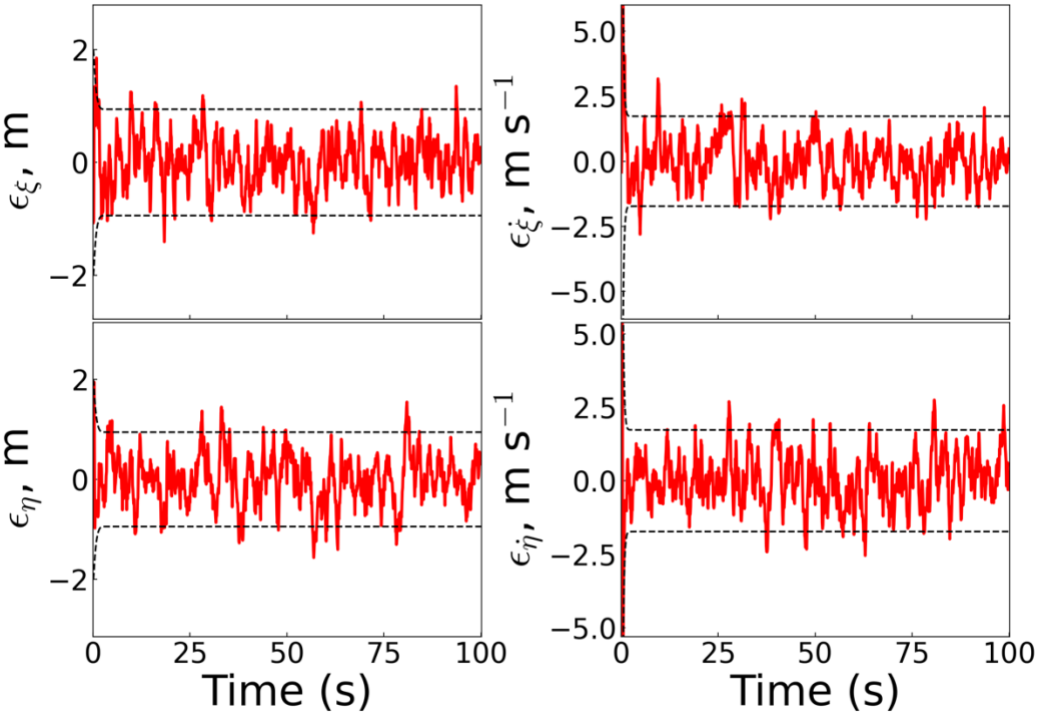

Figure 6: An example set of estimation errors and 2σ error bounds

NEES & NIS Statistics

The average NEES and NIS statistics computed over $N = 50$ rounds of testing are shown in Figures 7 and 8. They are centered around $n = 4$ and $p = 2$, respectively, which are the expected means of their n- and p-degree-of-freedom χ^2 distributions. Moreover, the majority of points (roughly 95% by a quick visual count of the outliers) lie between the r_1 and r_2 bounds for each statistic. So, we can reasonably conclude that $N \cdot \bar{\epsilon}_{\boldsymbol{x},k} \sim \chi^2_{N \cdot n}$ and $N \cdot \bar{\epsilon}_{\boldsymbol{y},k} \sim \chi^2_{N \cdot p}$.

Final Discussion

Given the outcomes of our truth model testing, we cannot declare our filter inconsistent and should therefore accept the null hypothesis that it is in fact consistent, with a significance level of $\alpha = 0.05$. We may still have no way of ending Buffmo’s reign of terror, but we can at least be reasonably confident in our ability to estimate its position and velocity as it mercilessly stalks the unsuspecting students of CU Boulder.

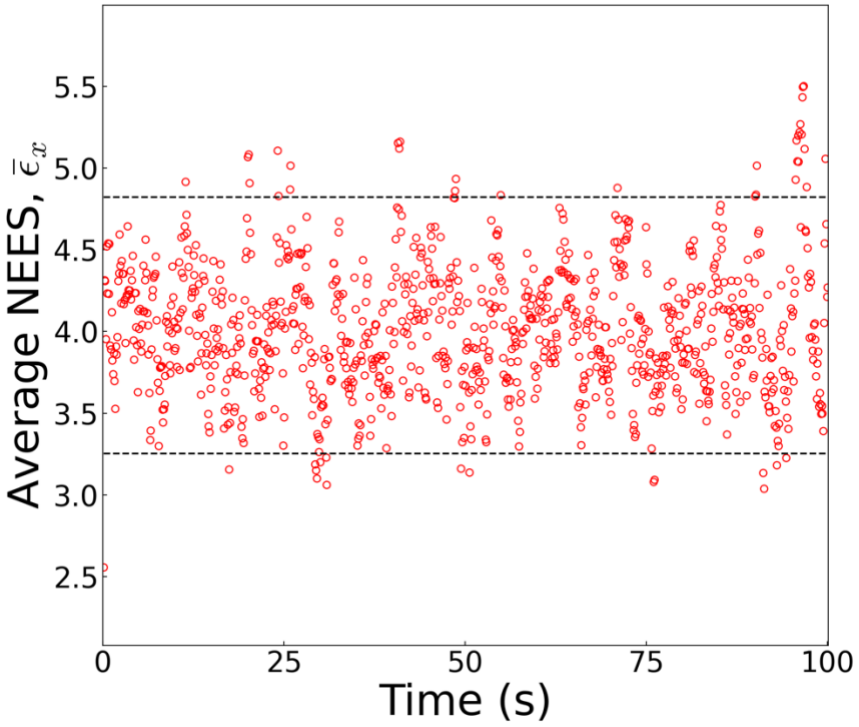

Figure 7: Average NEES statistics for 50 rounds of TMT

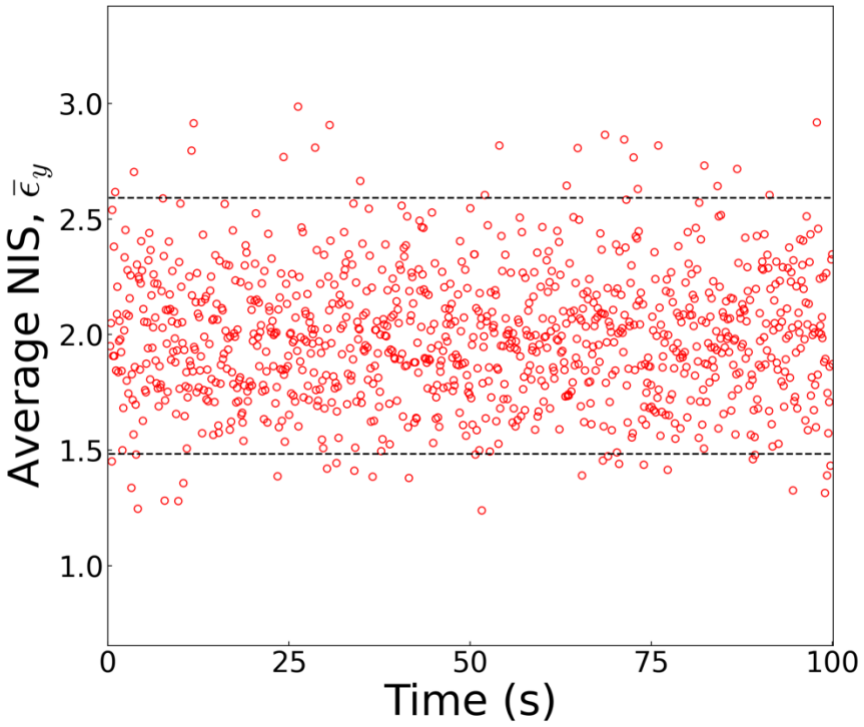

Figure 8: Average NIS statistics for 50 rounds of TMT

“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

Full Program

```
# "Buffmo" the Autonomous Bison
# (Kalman Filter & TMT Example)
# ASEN 5044, Fall 2021

from numpy import array, zeros, sin, cos, block, eye
from numpy.random import multivariate_normal as mvn, uniform as uni
from scipy.linalg import inv, expm
from scipy.stats import chi2
from matplotlib import pyplot as plt
plt.rcParams["mathtext.fontset"] = "cm"
plt.ion()

# Define parameters
tmax = 100
dt = 0.1
nt = int(tmax/dt)
Om = .75
f = .2
qW = 1
qR = 1

# Define or compute DT inputs & matrices
ts = array([dt*t for t in range(1,nt+1)])
u = array([f*cos(Om*ts),-f*sin(Om*ts)]).T
A = array([[0,1,0,0],[0,0,0,0],[0,0,0,1],[0,0,0,0]])
F = array([[1,dt,0,0],[0,1,0,0],[0,0,1,dt],[0,0,0,1]])
G = array([[.5*dt**2,0],[dt,0],[0,.5*dt**2],[0,dt]])
Gam = array([[0,0],[1,0],[0,0],[0,1]])
W = qW*array([[1,.1],[.1,1]])
Z = dt*block([[-A,Gam@W@Gam.T],[zeros((4,4)),A.T]])
Zhat = expm(Z)
Q = F@Zhat[:4,4:]
H = array([[1,0,0,0],[0,0,1,0]])
R = qR*array([[1,.1],[.1,1]])

# Loop over many rounds of truth model testing
nTMT = 50
NEES = zeros((nTMT,nt))
NIS = zeros((nTMT,nt))
for n in range(nTMT):

    # Simulate a set of noisy truth data & measurements
    xT = zeros((nt+1,4))
    ySim = zeros((nt,2))
    xT[0] = array([uni(-10,10),uni(-2,2),uni(-10,10),uni(-2,2)])
    for t in range(nt):
        xT[t+1] = F@xT[t]+G@u[t]+mvn(zeros((4)),Q)
        ySim[t] = H@xT[t+1]+mvn(zeros((2)),R)

    # Initialize the Kalman filter
    xh = zeros((nt+1,4))
    P = zeros((nt+1,4,4))
    xh[0] = array([0,0,0,0])
    P[0] = 1e3*eye(4)

    # Perform Kalman filter updates, compute NEES/NIS statistics
    for t in range(nt):

        xhm = F@xh[t]
        Pm = F@P[t]@F.T+Q
        Kk = Pm@H.T@inv(H@Pm@H.T+R)
        xh[t+1] = xhm+Kk@(ySim[t]-H@xhm)
        P[t+1] = (eye(4)-Kk@H)@Pm

        xerr = xT[t+1]-xh[t+1]
        yerr = ySim[t]-H@xhm
        Sk = H@Pm@H.T+R
        NEES[n,t] = xerr.T@inv(P[t+1])@xerr
        NIS[n,t] = yerr.T@inv(Sk)@yerr

# Compute NEES/NIS test quantities
alpha = 0.05
exbar = NEES.mean(axis=0)
eybar = NIS.mean(axis=0)
rNEES = array([chi2.ppf(alpha/2,nTMT*4),chi2.ppf(1-alpha/2,nTMT*4)])/nTMT
rNIS = array([chi2.ppf(alpha/2,nTMT*2),chi2.ppf(1-alpha/2,nTMT*2)])/nTMT
```

“Buffmo”, the Autonomous Bison
Kalman Filter & Truth Model Testing Example
Originally Created For: ASEN 5044 (CU Boulder, Fall 2021)

```

# Plot simulated measurements & true states vs. time
fig, ax = plt.subplots()
ax.plot(ySim[:,0],ySim[:,1],'ko',ms=7.5,ls='none',mfc='none',\
    label='Simulated Measurements')
ax.plot(xT[1:,0],xT[1:,2],'k-',lw=3,label='Truth Model Data')
ax.plot(xh[1:,0],xh[1:,2],'r--',lw=3,label='KF Estimates')
ax.set_xlabel(r"Easting Position $\xi$, m",fontsize=36)
ax.set_ylabel(r"Northing Position $\eta$, m",fontsize=36)
ax.tick_params(labelsize=24,direction="in")
ax.legend(fontsize=24,markerscale=1.5)
fig.subplots_adjust(left=.12,right=.96,top=.98,bottom=.12)
fig.show()

# Plot estimation errors & 2-sigma bounds vs. time
labels = [r"$\epsilon_{\xi}$, m",r"$\epsilon_{\dot{\xi}}$, m s$^{-1}$",\
    r"$\epsilon_{\eta}$, m",r"$\epsilon_{\dot{\eta} }$, m s$^{-1}$"]
fig, axs = plt.subplots(2,2)
for i in range(2):
    for j in range(2):
        ij = 2*i+j
        ex = xT[1:,ij]-xh[1:,ij]
        hrange = (ex[10:].max()-ex[10:].min())
        mid = ex[10:].mean()
        lime = [mid-1.01*hrange,mid+1.01*hrange]
        axs[i,j].plot(ts,ex,'r-',lw=2.5)
        axs[i,j].plot(ts,2*P[1:,ij,ij]**.5,'k--',lw=1.5)
        axs[i,j].plot(ts,-2*P[1:,ij,ij]**.5,'k--',lw=1.5)
        axs[i,j].axis([0,dt*(nt+1),lime[0],lime[1]])
        axs[i,j].set_xticks([0,25,50,75,100])
        axs[i,j].tick_params(labelsize=24,direction="in")
        if i==1: axs[i,j].set_xlabel(r"Time (s)",fontsize=36)
        else: axs[i,j].set_xticklabels([])
        axs[i,j].set_ylabel(labels[ij],fontsize=36)
fig.subplots_adjust(left=.1,right=.96,top=.98,bottom=.12,wspace=.3,hspace=.01)
fig.show()

# Plot average NEES statistic vs. time & 95% X^2 bounds
hrange = (rNEES[1]-rNEES[0])/2
mid = (rNEES[1]+rNEES[0])/2
lim = [mid-2.5*hrange,mid+2.5*hrange]
fig, ax = plt.subplots()
ax.plot(ts,exbar,'ro',mfc='none')
ax.plot(ts,rNEES[0]+zeros((nt)),'k--',lw=1.5)
ax.plot(ts,rNEES[1]+zeros((nt)),'k--',lw=1.5)
ax.axis([0,dt*(nt+1),lim[0],lim[1]])
ax.tick_params(labelsize=24,direction="in")
ax.set_xticks([0,25,50,75,100])
ax.set_xlabel(r"Time (s)",fontsize=36)
ax.set_ylabel(r"Average NEES, $\bar{\epsilon}_x$",fontsize=36)
fig.subplots_adjust(left=.12,right=.96,top=.98,bottom=.12)
fig.show()

# Plot average NIS statistic vs. time & 95% X^2 bounds
hrange = (rNIS[1]-rNIS[0])/2
mid = (rNIS[1]+rNIS[0])/2
lim = [mid-2.5*hrange,mid+2.5*hrange]
fig, ax = plt.subplots()
ax.plot(ts,eybar,'ro',mfc='none')
ax.plot(ts,rNIS[0]+zeros((nt)),'k--',lw=1.5)
ax.plot(ts,rNIS[1]+zeros((nt)),'k--',lw=1.5)
ax.axis([0,dt*(nt+1),lim[0],lim[1]])
ax.tick_params(labelsize=24,direction="in")
ax.set_xticks([0,25,50,75,100])
ax.set_xlabel(r"Time (s)",fontsize=36)
ax.set_ylabel(r"Average NIS, $\bar{\epsilon}_y$",fontsize=36)
fig.subplots_adjust(left=.12,right=.96,top=.98,bottom=.12)
fig.show()
```