UNIVERSITY OF CALIFORNIA, LOS ANGELES
MAE 271B: Stochastic Estimation

Estimation of Missile State

Name: Jiahui Lu
UID: 204945099
Instructor: Prof. Rezaei Ali
March 5, 2018

Abstract

In the problem estimation missile state, the dynamic states are an essential part for paying attention. To help the estimation of missile state, the position, velocity and acceleration should be estimated. Some work need to be done so as to deal with the noisy measurement and even with complicated telegraph signals. In this course project, continuous time Kalman filter algorithm is applied combined with measurement. Also, the dynamics of the system is built up assuming the acceleration is a Gauss-Markov process. After carrying out the continuous time Kalman filter algorithm, the result of the actual position, velocity and acceleration of the truth model are compared with the result in the continuous time Kalman filter algorithm together with the outcome of Kalman gains history and RMS as the error in the 10 seconds time after the missile launched. Then the Monte Carlo simulation for 5000 times of realization gives the constructions to find the ensemble averages over a set of realizations, which tends to have the variance of error for position, velocity and acceleration as a result as well as the error being very small showing the right implementation of continuous time Kalman filter algorithm. At last, random telegraph signal for Kalman filter with parameters that are set up is implemented so as to show the rightness of the continuous time Kalman filter algorithm.

1 INTRODUCTION

As for the problem of missile targeting, it is rather essential to maintain a good estimation of position, velocity and acceleration, which may not be easy. However, with the help of kalman filter algorithm, the state can be well estimated with reasonable assumptions of system dynamics as well as measurements.

In this project, the state of a missile needs to be estimated, which includes the relative position, velocity and acceleration. One important assumption is that the measurement is available in continuous manner. The plot of the structure of the problem of be found in Fig.1 below.

In the process, the assumptions are that the acceleration of the target is deemed as a stochastic differential equation and we linearize the angle of the pursuer to the target. Thereby, the based linear continuous-time state space form is set up to be utilized in the algorithm for continuous time Kalman filter.

Kalman filter algorithm is wildly implemented to optimally blend the measurements. With the assumption that the additive noise is Gaussian, the filter becomes a continuous-time conditional mean

estimator. If the additive noise is uncorrelated, the best linear filter is obtained and is equivalent to the Kalman filter in structure. We then specialize to infinite-time, time-invariant systems.

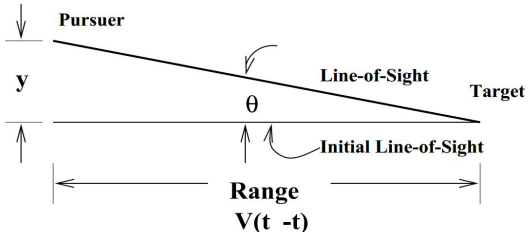

Fig.1 Missile Intercept Illustration

In the following section, we will carry out the continuous time Kalman filter algorithm. The very first step is to get the system dynamics and set up the parameters used in model as well as the measurement. Then the result of the actual position, velocity and acceleration of the truth model will be compared with the result in the continuous time Kalman filter algorithm together with the outcome of Kalman gains history and RMS as the error in the 10 second time after the missile launched. This part will based in the mathematical calculations.

Then the Monte Carlo simulation for 5000 times of realization gives the constructions to find the ensemble averages over a set of realizations, then the variance of error for position, velocity and acceleration as a result as well as the error being will plotted to verify the right implementation of continuous time Kalman filter algorithm. Moreover, we can in this process to show the independence of the residuals can to be checked, meaning residual process is uncorrelated in time.

At last, random telegraph signal for Kalman filter with parameters that are set up is implemented The objective of the above is to use the random telegraph signal in the simulation rather than the Gauss-Markov process that was used to ensure that the Kalman filter was implemented correctly.

2 THEORY AND ALGORITHM

2.1 System Dynamics

The dynamics of the problem are:

$$\dot{y} = v$$
$$\dot{v} = a_p - a_T$$

where a_p, the missile acceleration, is known and assumed here to be zero. The input, a_T, is the target acceleration and is treated as a random forcing function with an exponential correlation,

$$E[a_T] = 0$$
$$E[(a_T(t)a_T(s)] = E[a_T^2]e^{-\frac{|t-s|}{\tau}}$$

The scalar, τ, is the correlation time. The initial lateral position, $y(t_0)$, is zero by definition. The initial lateral velocity, $v(t_0)$, is random and assumed to be the result of launching error:

$$E[y(t_0)] = 0\ E[v(t_0)] = 0$$
$$E[y(t_0)^2] = 0\ E[y(t_0)v(t_0)] = 0$$
$$E[v(t_0)^2] = given.$$

2.2 Measurement

The fig.2 shows another illustration from the textbook.

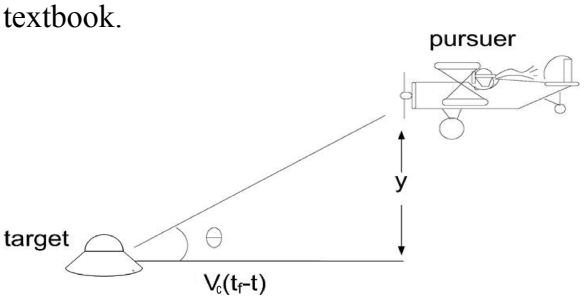

Fig.2 Missile Intercept Illustration

The measurement, z, consists of a line-of-sight angle θ in Fig. 1. It can be obtained by nonlinear relationship but can be linearized when $|\theta| \ll 1$:

$$\theta \approx \frac{y}{V_c(t_f - t)}$$

It will also be assumed that z is corrupted by fading and scintillation noise so that

$$y = \theta + n$$
$$E[n(t)] = 0$$
$$E[n(t)n(\tau)] = V\,\delta(t-\tau)$$
$$= [R_1 + \frac{R_2}{(t_f - t)^2}]\delta(t-\tau)$$

The process noise spectral density, W, is

$$W = GE[a_T^2]G^T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & E[a_T^2] \end{bmatrix}$$

The parameters of the problem of missile are listed below:

$$V_c = 300\,ft/sec, E[a_T^2] = [100\,ft \cdot \sec^{-2}]^2$$
$$t_f = 10\,sec, R_1 = 15 \times 10^{-6}\,rad^2 \cdot sec$$
$$R_2 = 1.67 \times 10^{-3} rad^2 \cdot sec, \tau = 2sec$$

The initial covariance is:

$$P(0) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & (200sec^{-1})^2 & 0 \\ 0 & 0 & (100sec^{-2})^2 \end{bmatrix}$$

2.3 Dynamic Model

The estimator can be designed according to previous discussion and the acceleration model can be discerned as Gauss-Markov process. The state-space equation for the missile intercept problem is:

$$\begin{bmatrix} \dot{y} \\ \dot{v} \\ \dot{a_T} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & -\frac{1}{\tau} \end{bmatrix}}_{F} \begin{bmatrix} y \\ v \\ a_T \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} a_p + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{G} \omega_{a_T}$$

We assume $a_p = 0$.

So, we can have:

$$\begin{bmatrix} \dot{y} \\ \dot{v} \\ \dot{a_T} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & -\frac{1}{\tau} \end{bmatrix}}_{F} \begin{bmatrix} y \\ v \\ a_T \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}}_{G} \omega_{a_T}$$

The measurement, in the meanwhile:

$$z = \underbrace{\begin{bmatrix} \frac{1}{V_c(t_f - t)} & 0 & 0 \end{bmatrix}}_{H} \begin{Bmatrix} y \\ v \\ a_T \end{Bmatrix} + n.$$

For process noise PSD and sensor noise PSD, we can have:

$$\begin{cases} W_{PSD} = E[a_T^2] = [100\,ft \cdot \sec^{-2}]^2 \\ V_{PSD} = [R_1 + \frac{R_2}{(t_f - t)^2}] rad^2 \cdot \sec \end{cases}$$

Substituting into continuous-time stochastic differential equation:

$$dx(t) = F(t)x(t)dt + G(t)d\beta_t$$
$$dz = H(t)x(t)dt + d\eta_t$$

In the meanwhile, $F(t), G(t), H(t)$ are marked in the equation above. The other parameters are listed below:

$$E[d\beta_t] = 0$$
$$E[d\eta_t] = 0$$
$$E[\omega(t)^2] = W_{PSD}\delta(t) = 100\delta(t)$$
$$E[n(t)^2] = V_{PSD}\delta(t)$$

Noted that $H(t), V_{PSD}$ are time-varying and $\omega(t), n(t)$ are stochastic Guass-Markov process with zero means and variance listed above.

2.4 Kalman Filter

Based on the stochastic differential equation derived from the system model, the Kalman filter can be applied.

$$d\hat{x}(t) = F(t)\hat{x}(t)dt + P(t)H(t)^T V(t) \cdot [dz(t) - H(t)\hat{x}(t)dt]$$

The error variance can be obtained from the Riccati equation:

$$\dot{P}(t) = F(t)P(t) + P(t)F(t)^T - P(t)H(t)^T V(t)^{-1}H(t)P(t) + G(t)W(t)G(t)^T, \; P(0) = P_0$$

By the way, Kalman gain is

$$K(t) = P(t)H(t)^T V(t)$$

To compute, we can have the Kalman filter with the form:

$$\dot{\hat{y}} = \hat{v} + K_1\left(z - \frac{\hat{y}}{V_c(t_f - t)}\right)$$

$$\dot{\hat{v}} = -\hat{a}_T + K_2\left(z - \frac{\hat{y}}{V_c(t_f - t)}\right) + a_p$$

$$\dot{\hat{a}_T} = -\frac{\hat{a}_T}{\tau} + K_3\left(z - \frac{\hat{y}}{V_c(t_f - t)}\right)$$

Where $a_p = 0$ is assumed before, and the gains are:

$$K_1 = \frac{p_{11}}{V_c R_1(t_f - t) + \frac{V_c R_2}{t_f - t}}$$

$$K_2 = \frac{p_{12}}{V_c R_1(t_f - t) + \frac{V_c R_2}{t_f - t}}$$

$$K_1 = \frac{p_{13}}{V_c R_1(t_f - t) + \frac{V_c R_2}{t_f - t}}$$

The scalars, p_{ij}, are the (i, j) elements of the error covariance matrix that is propagated by the Riccati equation:

$$\dot{P} = FP + PF^T - \frac{1}{V_c^2 R_1(t_f - t) + V_c^2 R_2} P\bar{H}^T\bar{H}P + W$$

Where $\bar{H} = [1\ 0\ 0]$. The process noise spectral density, W, is

$$W = GE[a_T^2]G^T = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & E[a_T^2] \end{bmatrix}$$

3. Simulation Results

Python 3 is used as the environment to run the simulation. Numpy is a good tool for computing, which well supports the matrix manipulations. Noting that the filter we used are continuous time, so there are no propagation and measurement updates due to setting a enough high frequency. In the coding part, the time step is set as 0.01s.

3.1 Kalman Gains and RMS

Based on the previous analysis, we can have the Kalman gains and RMS plotted. We continuous update the P matrix so as to compute gains. We can then get the Matrix covariance matrix. Knowing that the diagonal elements of the error covariance matrix represents the root-mean-square of the error when using the Kalman filter. Fig. 3 and Fig. 4 gives the plots of the Kalman Gains and RMS for each state.

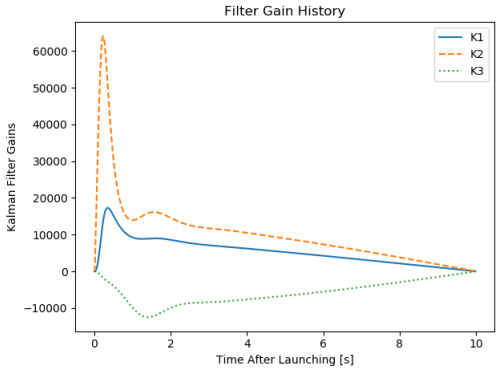

Fig. 3 Filter Gain History

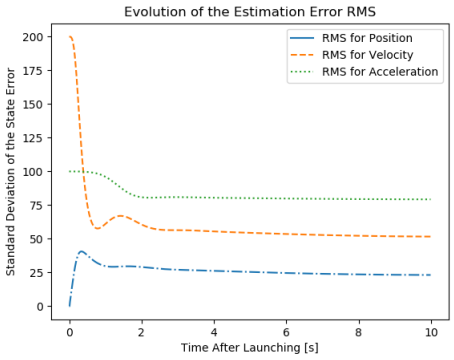

Fig. 4 Evolution of the Estimation Error RMS

It can be discerned that, Kalman filter is working

by noticing that the estimation variances get to a relatively low value. In addition, it can be seen that the position RMS is about 25ft, which is rather good. Also, the Kalman gains are well-presented, which can be store and computed off-line.

3.2 Dynamic and Measurement Simulation

Quite distinct beforehand, the dynamics are different in each realization owing to the stochastic relationship, which makes these cannot be computed offline. We will have distinct plot in each time we run the simulation. We can have the plots of actual states.

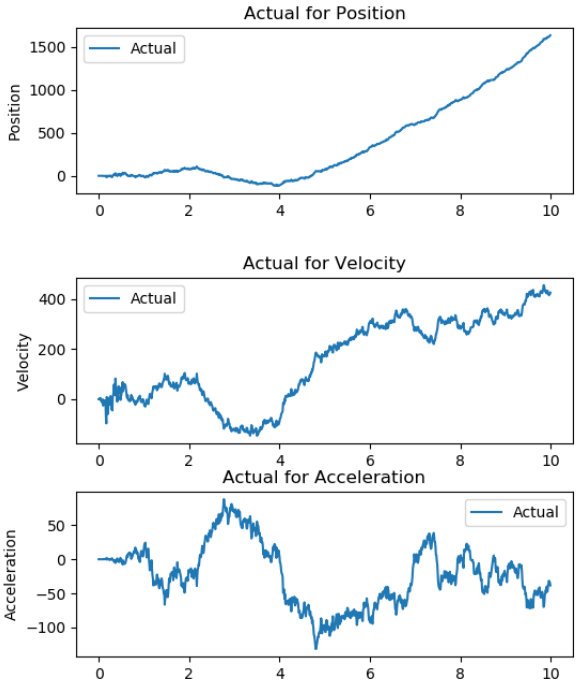

Fig 5 Actual States

Using Python to run the simulation in the time loop and we update the recurrence relationship dynamically.

$$\dot{x} = Fx + \frac{Gd\beta}{dt}$$
$$x_{t+1} = x_t + \dot{x}dt$$
$$\frac{d\beta}{dt} \sim N\left(0, \frac{W_{PSD}}{dt}\right)$$
$$x_0 = \begin{bmatrix} 0 \\ N(0,200) \\ N(0,100) \end{bmatrix}$$

Also for measurement:

$$dz = Hxdt + \frac{d\eta_t}{dt}$$
$$\frac{d\eta_t}{dt} \sim N\left(0, \frac{V_{PSD}}{dt}\right)$$
$$z_0 = 0$$

Therefore, we can get the plots showing in Fig. 5.

3.3 Estimation of states

The states estimation can be easily derived based on previous deduction. As we know, Kalman filter blends the stochastic relationship obtained as well as the model dynamic and measurement so as to estimate the state.

$$d\hat{x}(t) = F(t)\hat{x}(t)dt + P(t)H(t)^T V(t) \cdot [dz(t) - H(t)\hat{x}(t)dt]$$

Updating this will give the result of state estimation.

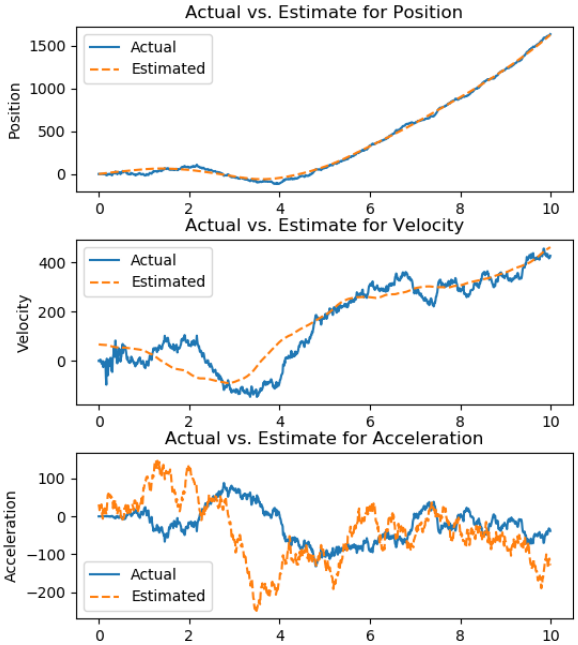

Fig. 6 Actual States VS. Estimated States

We can have figures showing the result of the simulation for one realization as well as the error bounded by the RMS of the error according to the

calculation in previous section. Generally, the error estimation is still mainly in the bound of $\pm\sigma$, indicating a good state estimation for the process.

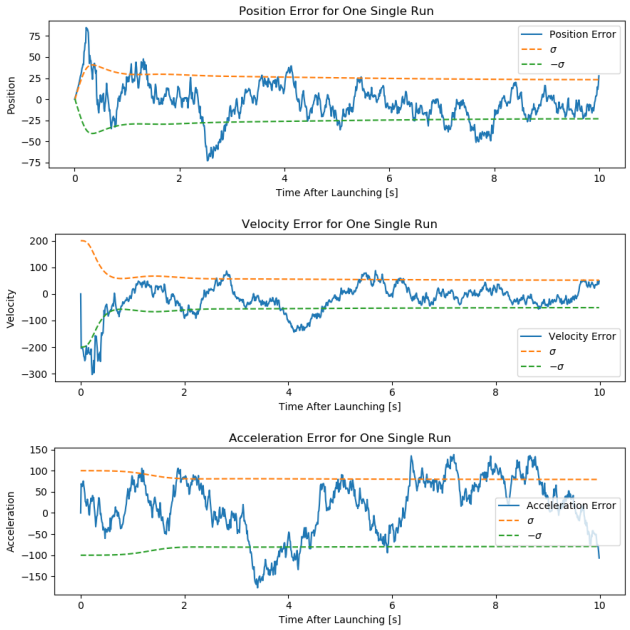

Fig. 7 Single Run for States Error VS. Bounds

3.4 Monte Carlo Simulation

A Monte Carlo simulation is to be constructed to find the ensemble averages over a set of realizations. Let $e^l(t_i)$ represent the actual error for realization l. This vector can be described as:

$$e^l(t_i) = \begin{bmatrix} e_y^l(t_i) \\ e_b^l(t_i) \\ e_{a_T}^l(t_i) \end{bmatrix}$$

To obtain each realization l, an initial condition for $v(0), a_T(0)$ are generated from a Gaussian noise generator. The state estimate is determined from measurements where the measurement noise for $\eta(t_i)$ are generated at each measurement time t_i from a Gaussian noise generator. The process noise $w(t_j)$ is generated in Gaussian noise generator.

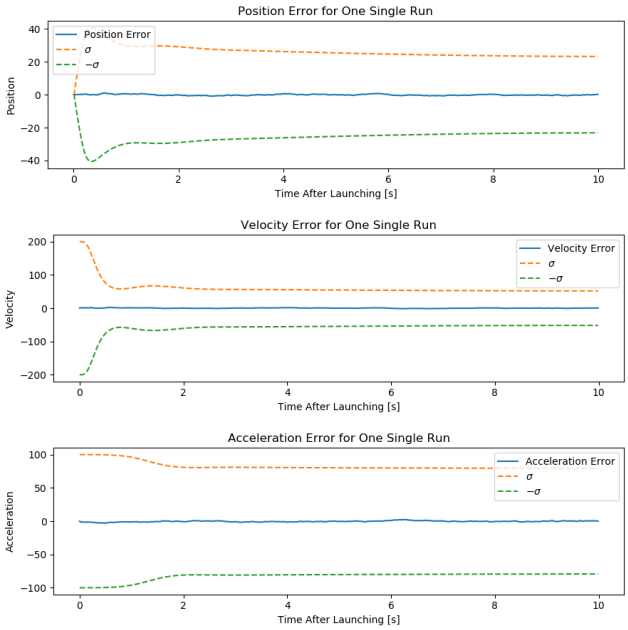

Fig. 8 States Error VS. Bounds for Monte Carlo Simulation of 5000 times

We first show that The ensemble average of $e^l(t_j)$ which produces the actual mean and is obtained directly from a Monte Carlo simulation is close to zero.

If having the number of realizations huge enough, then we can produce a good approximate statistic of model and performance so that it should be close to zero for all t in range. The mathematical formulation to produce the actual mean is

$$e^{ave}(t_i) = \frac{1}{N_{ave}} \sum_{l=1}^{N_{ave}} e^l(t_i) \approx 0,$$

The results of the Monte Carlo simulation for the expected value of the error are shown in Fig. 8. We choose

$$N_{ave} = 5000$$

Ensemble average producing the actual error variance P^{ave}:

$$P^{ave} = \frac{1}{N_{ave} - 1} \sum_{l=1}^{N_{ave}} [e^l(t_i) - e^{ave}(t_i)][e^l(t_i) - e^{ave}(t_i)]^T$$

Where $N_{ave}-1$is used for unbiased variance from small sample theory. The matrix $P^{ave}(t_i)$ should be close to $P(t_i)$ which is a posteriori error covariance matrix in the Kalman filter algorithm for a single run. This is an important check to verify that the modeling is approximately correct and the Kalman filter has been programmed and implemented correctly.

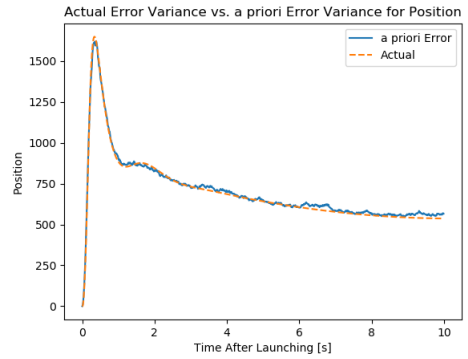

Fig. 9 Actual Error Variance VS. Priori Error Variance for Position

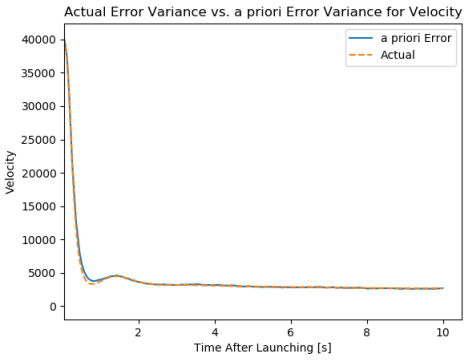

Fig. 10 Actual Error Variance VS. Priori Error Variance for Velocity

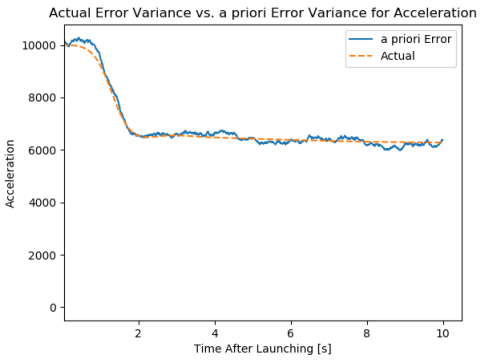

Fig. 11 Actual Error Variance VS. Priori Error Variance for Acceleration

Finally, the independence of the residuals can to be checked. The residual for a realization can be

$$r^l(t_i) = z^l(t_i) - H(t)\hat{x}^l(t)$$
$$\frac{1}{N_{ave}}\sum_{l=1}^{N_{ave}} r^l(t_i)r^l(t_m)^T \approx 0, \forall t_m < t_i$$

$r^l(t_i)$ is the residual at time at time t_i .The time chosen are following:

$$t_i = 3s, t_j = 7s$$

Yielding a result almost being -5.48057972785e-05, which is rather zero.

Autocorrelation for Residual Process for 5000 Realization is shown below in Fig. 12.

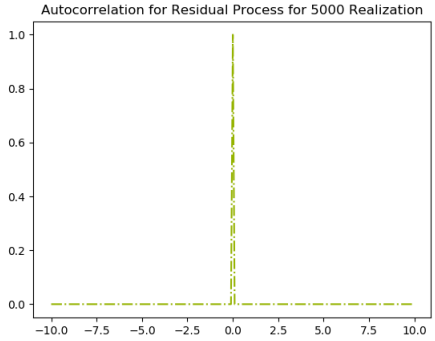

Fig. 12 Autocorrelation for Residual Process for 5000 Realization

A conclusion can be drawn here about the performance analysis is that we expect very small results for this checking under the large times of trials, which suggests that correct and nice implementation of the Kalman filter.

4. Random Telegraph Signal for Kalman Filter

The Gauss-Markov model is an approximation to the random telegraph signal model that is more realistic. In this model the value a_T changes sign at random times given by a Poisson probability. We assume $a_T(0) = \pm a_T$ with probability 0.5

and $a_T(t)$ changes polarity at Poisson times. The probability of k sign changes in a time interval of length T, $P(k(T))$, is $P\big(k(T)\big) = \frac{(\lambda T)^k e^{-\lambda T}}{k!}$, where λ is the rate. The probability of an even number of sign changes in T, $P(even\ \#\ in\ T)$, is

$$P(even\ \#\ in\ T) = \sum_{\substack{k=0 \\ k\ even}}^{\infty} \frac{(\lambda T)^k e^{-\lambda T}}{k!}$$
$$= e^{-\lambda T} \sum_{\substack{k=0 \\ k\ even}}^{\infty} \frac{(1+(-1)^k)(\lambda T)^k}{2k!}$$

Since $e^{-\lambda T} = \sum_{k=0}^{\infty} \frac{(\lambda T)^k}{2k!}$, then

$$P(even\ \#\ in\ T) = \sum_{k=0}^{\infty} \frac{(\lambda T)^k}{2k!} + \sum_{k=0}^{\infty} \frac{(-\lambda T)^k}{2k!}$$
$$= \frac{1}{2}[1 + e^{-2\lambda T}]$$

Likewise, $P(odd\ \#\ in\ T) = \frac{1}{2}[1 + e^{-2\lambda T}]$. Then the probability that $a_T(t)$ is positive:

$$P(a_T(t) = a_T)$$
$$= P(a_T(t) = a_T \,|a_T\,(0) = a_T) P(a_T(0) = a_T)$$
$$+ P(a_T(t) = a_T | a_T\,(0) = -aT\,) P(aT\,(0) = -aT\,)$$
$$= \frac{1}{2}\mathrm{P}(\text{even \# in T} = [0; \mathrm{t}])$$
$$+ \frac{1}{2}\mathrm{P}(\text{odd \# in T} = [0; \mathrm{t}])$$
$$= \frac{1}{2}\left\{[1 + e^{-2\lambda T}] + \frac{1}{2}[1 - e^{-2\lambda T}]\right\} = \frac{1}{2}$$

Then, the mean acceleration is $\bar{a}_T = 0$

The autocorrelation:

$$R_{a_T,a_T}(t_1, t_2) = a_T^2 P\big(a_T(t_2)\big) = a_T^2 \frac{1}{2}\big[1 + e^{-2\lambda|t_2 - t_1|}\big] - a_T^2 \frac{1}{2}\big[1 - e^{-2\lambda|t_2 - t_1|}\big] = a_T^2 e^{-2\lambda|t_2 - t_1|}$$

If $\frac{1}{\tau} = 2\lambda$ then the autocorrelation of the Gauss Markov process is the same as the random telegraph signal $(t_2 - t_1) = (t - s)$ and the means of both are zero.

We need a method of generating the random switching times. Let $T = t_{n+1} - t_n$ be the time between two switch times. The probability that the switch occurred after t_{n+1} is:

$$P(T' > T | t = t_n) = 1 - P(T' \le T | t = t_n)$$

Now the probability that no switch occurred in T, but occurred after t_{n+1} is:

$$P(T_0 > T | t = t_n)$$
$$= P(\#\ of\ sign\ changes\ in\ T\ is\ zero) = e^{-\lambda T}$$

Then,

$$P(T' \le T | t = t_n) = 1 - e^{-\lambda T}$$

which is the probability that at least one change occurred. To produce the random time, we set the output of [0, 1] from a uniform density function. Then,

$$1 - e^{-\lambda T} = U \rightarrow e^{-\lambda T} = 1 - U$$
$$\Rightarrow T = -\frac{1}{\lambda}\ln(1 - U)$$
$$\Rightarrow t_{n+1} - t_n = -\frac{1}{\lambda}\ln(1 - U)$$

Since $1 - U$ is also a uniform density function, then

$$t_{n+1} = t_n - \frac{1}{\lambda}\ln(1 - U)$$

The objective of the above is to use the random telegraph signal in the simulation rather than the Gauss-Markov process that was used to ensure that the Kalman filter was implemented correctly. By running a Monte Carlo analysis using the random telegraph signal with $a_T = 100 ft/s$ and $\lambda = 0.25 s^{-1}$. We determine the actual error variance shown in following part.

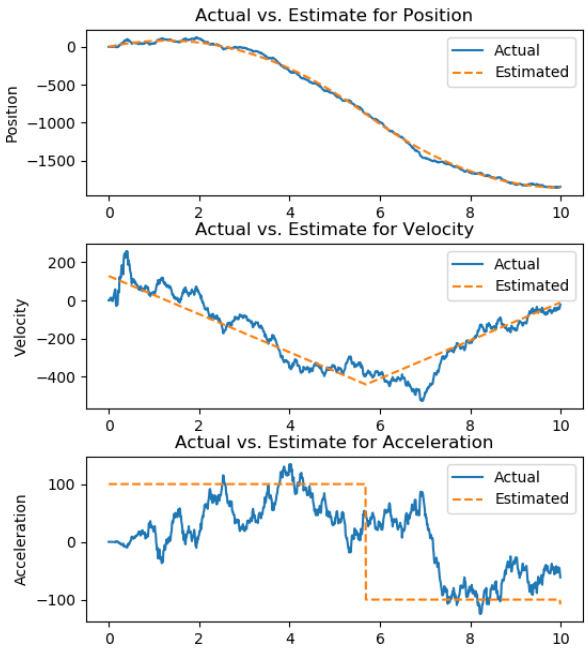

Fig. 12 Actual States VS. Estimated States with Random Telegraph Signal

Then, States Error VS. Bounds for Monte Carlo Simulation of 5000 times using with Random Telegraph Signal:

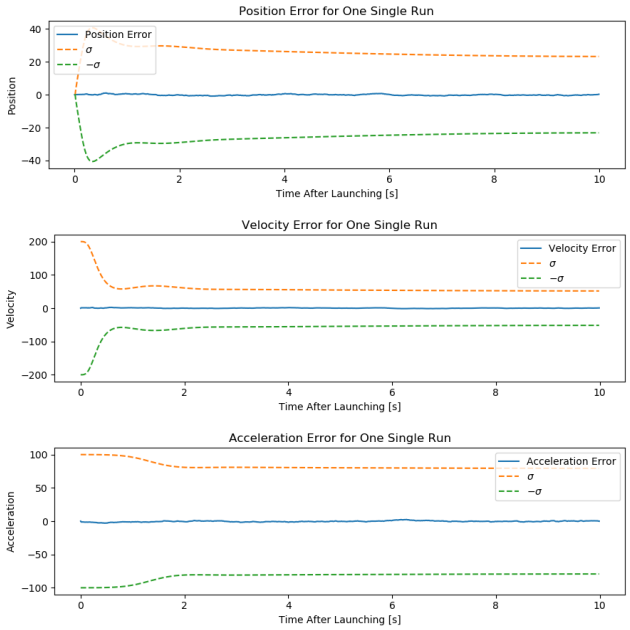

Fig. 13 States Error VS. Bounds for Monte Carlo Simulation of 5000 times with Random Telegraph Signal

Then, Actual error variance vs. Priori error variance for position, velocity and acceleration.

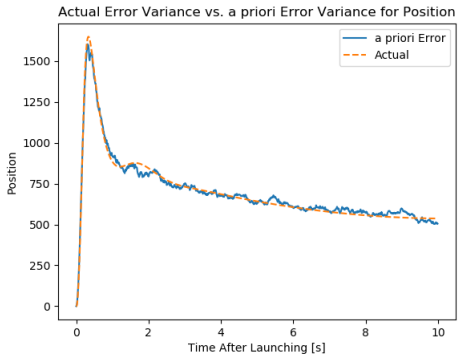

Fig. 14 Actual Error Variance VS. Priori Error Variance for Position with Random Telegraph Signal

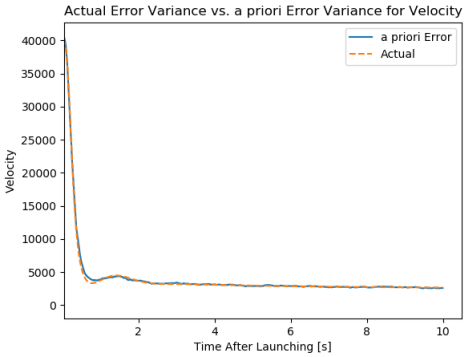

Fig. 15 Actual Error Variance VS. Priori Error Variance for Velocity with Random Telegraph Signal

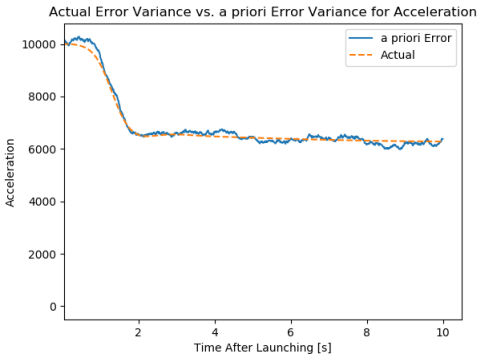

Fig. 16 Actual Error Variance VS. Priori Error Variance for Acceleration with Random Telegraph Signal

Compared with Fig. 9, 10, 11. It is clear that the tendency of each figure is almost the same together with the value. In the meanwhile, the

errors are being zero almost for each state. Therefore, the performances are rather similar for the two method. So we can say, with Random Telegraph Signal, we ensure that the Kalman filter was implemented correctly.

5. Conclusion

To sum up, in this course report, the system is deemed as a stochastic system and the continued time Kalman filter is applied to estimate the state of the missile. In the process, the assumptions are that the acceleration of the target is deemed as a stochastic differential equation and we linearize the angle of the pursuer to the target. Thereby, the based linear continuous-time state space form is set up to be utilized in the algorithm for continuous time Kalman filter.

For Kalman gains, we can discern that the three elements will converge to zero. In this process, as we can see that the variance of acceleration actually increases in the beginning showing that measurement is being paid more attention at first for the filter. In the meanwhile, for error variance the results show that in a time period of 10 seconds, the Kalman filter gives an excellent estimation in spite of the noisy measurement. In addition, a Monte Carlo simulation for 5000 times realizations is carried out to confirm that the Kalman filter's properties are satisfied by showing that the error conditional variance for each states and residuals whose process is uncorrelated in time.

To be concrete, focusing on true model and Kalman filter estimations, it can be vividly seen that the filtered and estimated position and velocity can track the truth model well and it is not too bad for the acceleration tracking, which can be tell from the Fig. 6. Moreover, the posteriori error values from the covariance matrix give the upper-bound and lower-bound. The errors lying in the bounds well present a good stochastic performance as general despite some values are out in this process which can be revisited in Fig. 7 for a single run.

Generally, the results of actual error variance vs. priori error variance for states give a rather similar performance, which indeed suggested a quite perfect implementation. The errors lying still being zero will also support the conclusion. So adding up the support from random telegraph signal, it demonstrates that despite of the fact the dynamics of system may bot be depicted as Gauss-Markov system, we can still obtain a get a good performance, which shows the good implantation of the Kalman filter one, as well.

Reference

[1] Kalman, R. E. (1960). *A New Approach to Linear Filtering and Prediction Problems*. Journal of Basic Engineering, 82(1), 35. Doi:10.1115/1.3662552

[2] Speyer, Jason L.. Chung, Walter H. (2011). *Stochastic Processes, Estimation, And Control*. S.L.: Society for Industrial And Applied Mathematics, Philadelphia

Appendix

Python Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Mar  5 17:19:18 2018

@author: carsonluuu
"""

import numpy as np
import matplotlib.pyplot as plt
import random
import math

"""

"""

def MonteCarlo(num, error_store, error_all, residual_all):

    Vc  = 300
    tf  = 10
    R1  = 15*(10**(-6))
    R2  = 1.67*(10**(-3))
    tau = 2
    W   = 100**2
    dt  = 0.01

    a_var  = 100.0**2
#    y_var  = 0
    v_var  = 200.0**2

    t = np.arange(0, 10, 0.01)

    length = len(t)
    timeList = range(0, length - 1)


    for index in range(num):
        if index%(num/250) == 0:
            print(str(100*index/num) + "% has been done.")
        x_pre     = np.zeros((3, 1, length))
```

```
    dx_pre    = np.zeros((3, 1, length))
    xhat_pre  = np.zeros((3, 1, length))
    dxhat_pre = np.zeros((3, 1, length))


    F  = np.array([[0, 1, 0],
                   [0, 0, -1],
                   [0, 0.0, -(1/tau)]])
    G  = np.array([[0], [0], [1]])
#     B  = np.array([[0], [1], [0]])
    P0 = np.array([[0,   0,    0],
                   [0, 200**2, 0],
                   [0, 0, 100**2]])

    P_pre = np.zeros((3, 3, length))
    K_pre = np.zeros((3, 1, length))
    Z_pre = np.zeros((length))


    H0 = np.array([[1.0/(Vc*tf), 0, 0]])
    V0 = R1 + (R2/(tf**2))
    y0 = 0
    v0  = np.random.normal(0, v_var**0.5)
    at0 = np.random.normal(0, a_var**0.5)
   #at0 = (1 - 2 * round(random.uniform(0, 1))) * 100
    wat = np.random.normal(0, (a_var/dt)**0.5)
    n_var = V0/dt
    n = np.random.normal(0, n_var**0.5)

    error_pre    = np.zeros((3, 1, length))
    residual     = np.zeros((1, length))

    P_pre[:,:,0] = P0
    K_pre[:,:,0] = np.dot( np.dot(P0, H0.T),
                           V0**-1 )

    x_pre[:,:,0]    = np.array([[y0], [v0], [at0]])
    xhat_pre[:,:,0] = 0

    dx_pre[:,:,0]    = np.dot(F, x_pre[:,:,0]) + np.dot(G, wat)
    dxhat_pre[:,:,0] = np.dot(F, xhat_pre[:, :, 0]) \
                           + np.dot(K_pre[:, :, 0], (Z_pre[0] \
                           - np.dot(H0, xhat_pre[:, :, 0])))

    Z_pre[0] = np.dot(H0, x_pre[:,:,0]) + n
    error_pre[:,:,0] = 0
```

```
    # t_x = -4*np.log(random.uniform(0, 1))


      for i in timeList:
         H = np.array([[1.0/(Vc*(tf-t[i])), 0.0, 0.0]])
         V = R1 + (R2/((tf-t[i])**2))
         P_dot = np.dot(F, P_pre[:,:,i]) \
               + np.dot(P_pre[:,:,i], F.T) \
               - np.dot(np.dot(np.dot(np.dot(P_pre[:,:,i], H.T), V**(-1)), H), P_pre[:,:,i]) \
               + np.dot(np.dot(G, W), G.T)

         P_pre[:,:,i+1] = P_pre[:,:,i] + np.dot(P_dot, dt)
         K_pre[:,:,i+1] = np.dot(np.dot(P_pre[:,:,i], H.T), (V**(-1)))


         n = np.random.normal(0, (V/dt)**0.5)
         wat = np.random.normal(0, (a_var/dt)**0.5)

         """
         if (t_x <= i*dt) :
            at0 = -at0
            x_pre[2,0,i] = at0
            t_x += -4*np.log(random.uniform(0, 1))
         else:
            x_pre[2,0,i] = at0
         """

         dx_pre[:,:,i+1] = np.dot(F, x_pre[:,:,i]) + np.dot(G, wat)
         x_pre[:,:,i+1]  = x_pre[:,:,i] + dx_pre[:,:,i+1]*dt
         Z_pre[i+1]      = np.dot(H, x_pre[:,:,i+1]) + n

         dxhat_pre[:,:,i+1] = np.dot(F, xhat_pre[:,:,i]) +\
                        np.dot(K_pre[:,:,i+1] , (Z_pre[i+1] - np.dot(H, xhat_pre[:,:,i])))
         xhat_pre[:,:,i+1]  = xhat_pre[:,:,i] + dxhat_pre[:,:,i+1]*dt

         residual[:,i+1]           = Z_pre[i+1] - np.dot(H, xhat_pre[:,:,i+1])
         residual_all[:,i+1, index] = residual[:,i+1]
         error_pre[:,:,i+1]        = xhat_pre[:,:,i+1] - x_pre[:,:,i+1]
         error_store[:,:,i+1,index] = error_pre[:,:,i+1]

      error_all += error_store[:,:,:,index]
   print("Monte Carlo simulation has finished with " + str(num) + " realizations")
   return K_pre, P_pre, x_pre, xhat_pre, error_all, error_store, error_pre, residual_all

def avg(P_out, error, error_store, residual_all, num):
```

```
    error_avg = error/num
    res_chk = 0

    for i in range(0, num):
        res_chk = res_chk + np.dot(residual_all[:,40,i], residual_all[:,80,i].T)
        for j in range(0, length):
            P_out[:,:,j] += np.dot((error_store[:,:,j,i] - \
                  error_avg[:,:,j]), (error_store[:,:,j,i] - error_avg[:,:,j]).T)

    _res_chk = res_chk/num
    res_chk = _res_chk
    print(res_chk)
    return P_out

def plot_Gain(K):

    plt.figure(1)
    plt.title('Filter Gain History')
    plt.plot(t, np.squeeze(K[0,:,:].T),label = "K1")
    plt.plot(t, np.squeeze(K[1,:,:].T), linestyle='--', label = "K2")
    plt.plot(t, np.squeeze(K[2,:,:].T), linestyle=':', label = "K3")
    plt.ylabel('Kalman Filter Gains')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

def plot_RMS(P):

    plt.figure(2)
    plt.title('Evolution of the Estimation Error RMS')
    plt.plot(t, np.squeeze(P[0, 0, :]**0.5), label = "RMS for Position")
    plt.plot(t, np.squeeze(P[1, 1, :]**0.5), linestyle='--', label = "RMS for Velocity")
    plt.plot(t, np.squeeze(P[2, 2, :]**0.5), linestyle=':', label = "RMS for Acceleration")
    plt.ylabel('Standard Deviation of the State Error')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")


def plot_Act_Est(xhat, x):

    plt.figure(3, figsize=(6, 2))
    plt.title('Actual vs. Estimate for Position')
    plt.plot(t, np.squeeze(xhat[0, 0, :]), label = "Actual")
    plt.plot(t, np.squeeze(x[0, 0, :]), linestyle='--', label = "Estimated")
    plt.ylabel('Position')
    plt.xlabel('Time After Launching [s]')
```

```
    plt.legend(loc="best")

    plt.figure(4, figsize=(6, 2))
    plt.title('Actual vs. Estimate for Velocity')
    plt.plot(t, np.squeeze(xhat[1, 0, :]), label = "Actual")
    plt.plot(t, np.squeeze(x[1, 0, :]), linestyle='--', label = "Estimated")
    plt.ylabel('Velocity')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

    plt.figure(5, figsize=(6, 2))
    plt.title('Actual vs. Estimate for Acceleration')
    plt.plot(t, np.squeeze(xhat[2, 0, :]), label = "Actual")
    plt.plot(t, np.squeeze(x[2, 0, :]), linestyle='--', label = "Estimated")
    plt.ylabel('Acceleration')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

def plot_Error_Variance(P_out_avg, P):

    plt.figure(6)
    plt.title('Actual Error Variance vs. a priori Error Variance for Position')
    plt.plot(t, P_out_avg[0, 0, :], label = "a priori Error")
    plt.plot(t, P[0, 0, :], linestyle='--', label = "Actual")
    plt.ylabel('Position')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

    plt.figure(7)
    plt.title('Actual Error Variance vs. a priori Error Variance for Velocity')
    plt.plot(t, P_out_avg[1, 1, :], label = "a priori Error")
    plt.plot(t, P[1, 1, :], linestyle='--', label = "Actual")
    plt.ylabel('Velocity')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

    plt.figure(8)
    plt.title('Actual Error Variance vs. a priori Error Variance for Acceleration')
    plt.plot(t, P_out_avg[2, 2, :], label = "a priori Error")
    plt.plot(t, P[2, 2, :], linestyle='--', label = "Actual")
    plt.ylabel('Acceleration')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

def plot_Error(error, P):
```

```
    plt.figure(9, figsize=(9, 3))
    plt.title('Position Error for One Single Run')
    plt.plot(t, np.squeeze(error[0,:,:].T), label = "Position Error")
    plt.plot(t, P[0, 0, :]**0.5, linestyle='--', label = r'$\sigma$')
    plt.plot(t, -P[0, 0, :]**0.5, linestyle='--', label = r'$-\sigma$')
    plt.ylabel('Position')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

    plt.figure(10, figsize=(9, 3))
    plt.title('Velocity Error for One Single Run')
    plt.plot(t, np.squeeze(error[1,:,:].T), label = "Velocity Error")
    plt.plot(t, P[1, 1, :]**0.5, linestyle='--', label = r'$\sigma$')
    plt.plot(t, -P[1, 1, :]**0.5, linestyle='--', label = r'$-\sigma$')
    plt.ylabel('Velocity')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")

    plt.figure(11, figsize=(9, 3))
    plt.title('Acceleration Error for One Single Run')
    plt.plot(t, np.squeeze(error[2,:,:].T), label = "Acceleration Error")
    plt.plot(t, P[2, 2, :]**0.5, linestyle='--', label = r'$\sigma$')
    plt.plot(t, -P[2, 2, :]**0.5, linestyle='--', label = r'$-\sigma$')
    plt.ylabel('Acceleration')
    plt.xlabel('Time After Launching [s]')
    plt.legend(loc="best")


if __name__ == "__main__":

    num = 1000
    t = np.arange(0, 10, 0.01)
    length = len(t)

    P_out       = np.zeros((3, 3, length))
    error_store  = np.zeros((3, 1, length, num))
    error_all    = np.zeros((3, 1, length))
    residual_all = np.zeros((1, length, num))

    K, P, x, xhat, error, error_store, error_, residual_all \
       = MonteCarlo(num, error_store, error_all, residual_all)
    P_out_avg = avg(P_out, error, error_store, residual_all, num)

#   plot_Gain(K)
#   plot_RMS(P)
#   plot_Act_Est(xhat, x)
```

```
#    plot_Error_Variance(P_out_avg/(num-1), P)
#    plot_Error(error_, P) #single
#    plot_Error(error/num, P) #average
```