Maxwell Cobar
Nonlinear Controls
Kalman Filter Homework

1 Introduction:

For this homework, the objective was to simulate a 1st order system with a Kalman Filter. A Kalman Filter is a linear estimation technique used in statistics and control theory. The Kalman Filter works by using a model of the system to estimate the states of the system. An initial measurement is taken from the sensor on the system which gives the model a starting point. From here, the model is integrated to estimate the states of the system. The model however can have noise that makes the model's state estimates deviate from the actual states. At some determined time interval, a measurement from the sensor on the system is taken. The sensor like the model can also have noise that can make the sensor estimate inaccurate. Using weighted linear least squares, an average is taken between the model estimate and the sensor estimate. This average is then used as the new starting point for the model and integration of the model is continued [1]. Kalman Filters are commonly used for multiple applications such as robotics motion planning and control; trajectory optimization; and guidance, navigation, and controls of vehicles. There is also a nonlinear form of the Kalman Filter known as the Extended State Kalman Filter [2].

2 Results and Discussion:

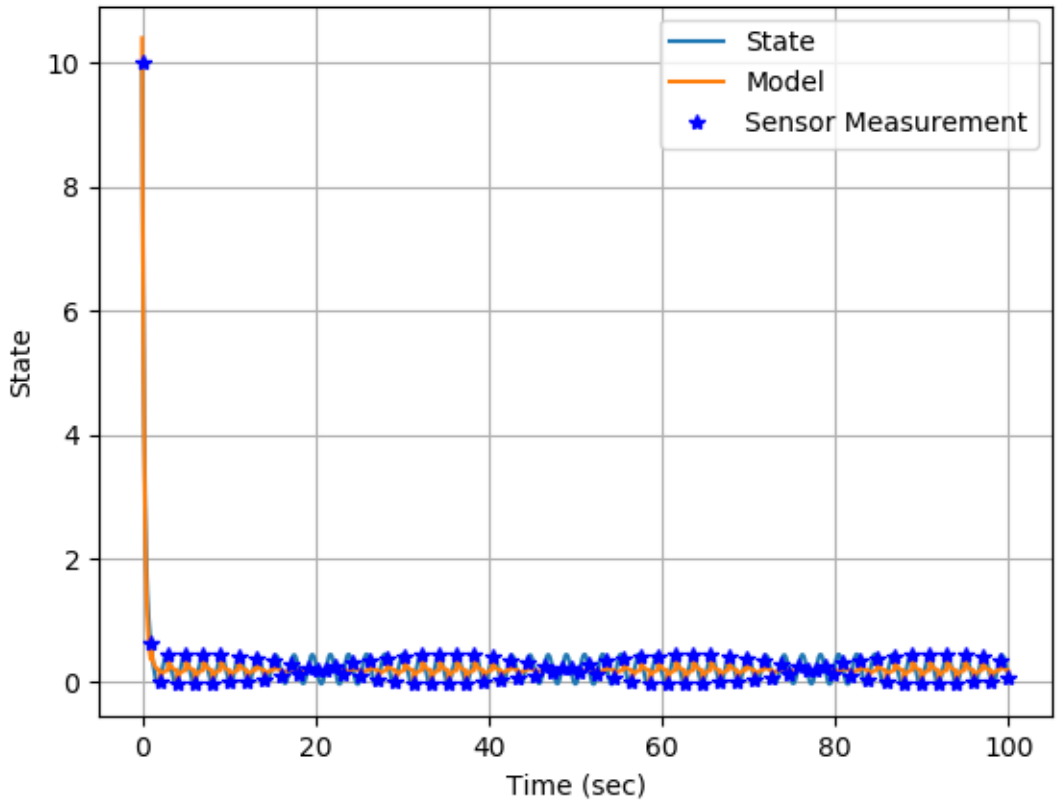

Figure 1: The model estimates, sensor estimates, and actual states plotted against time

In Fig. 1, the model estimates, sensor estimates, and actual states plotted against time. The initial condition of the system is set at a value of 10. The state decreases and oscillates around a value of 0.25.

The sensor estimates are shown in blue stars and line up with the actual states. The model is shown in orange and is integrated over time to estimate the states of the system. For this homework, the sensor was believed to be more accurate then the model, so the model is updated to the sensor estimate and integrated until the next sensor estimate is taken. Upon closer inspection, the system is experiencing aliasing which can be due to the frequency of measurements taken being too high.

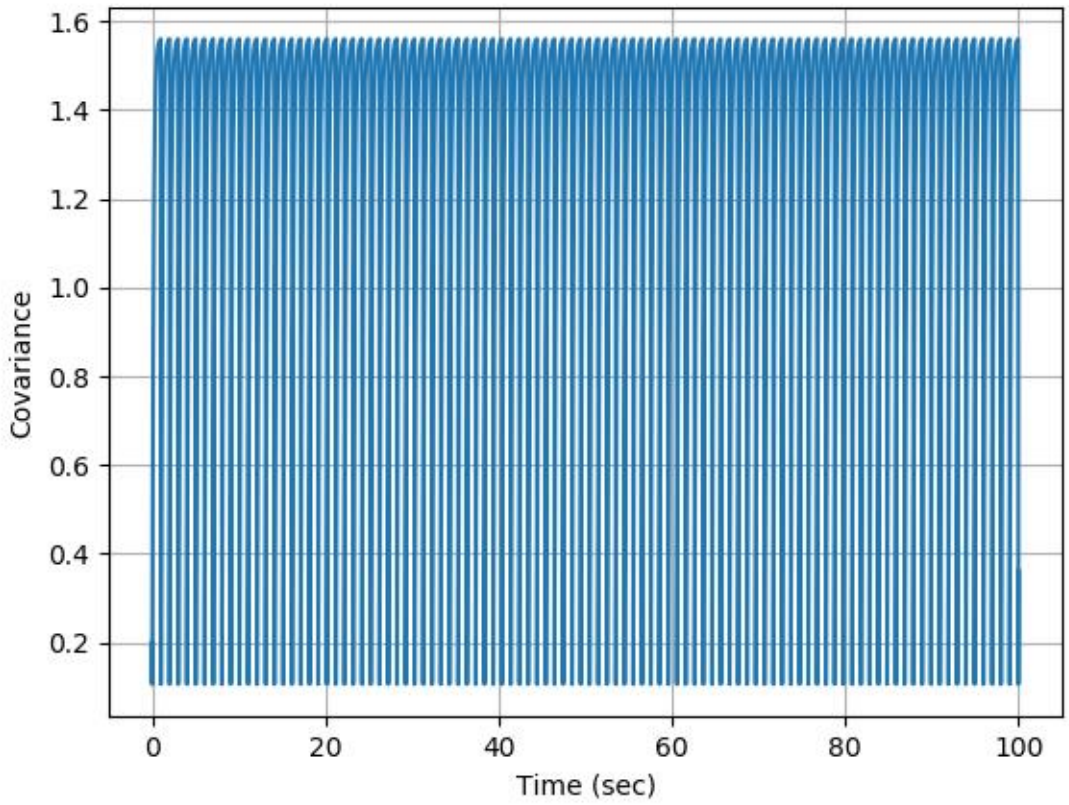

Figure 2: The covariance propagation plotted against time

In Fig. 2, the covariance propagation plotted against time. The covariance propagation is the expectation of the error from the estimated states and the actual states. As shown in Fig. 2, the covariance increases as the model deviates from the actual states. The covariance jumps back to a small number as the new model start point is found using weighted linear least squares. However, since the sensor and model estimation will never be a perfect match with the actual states, the covariance will never decrease to zero.

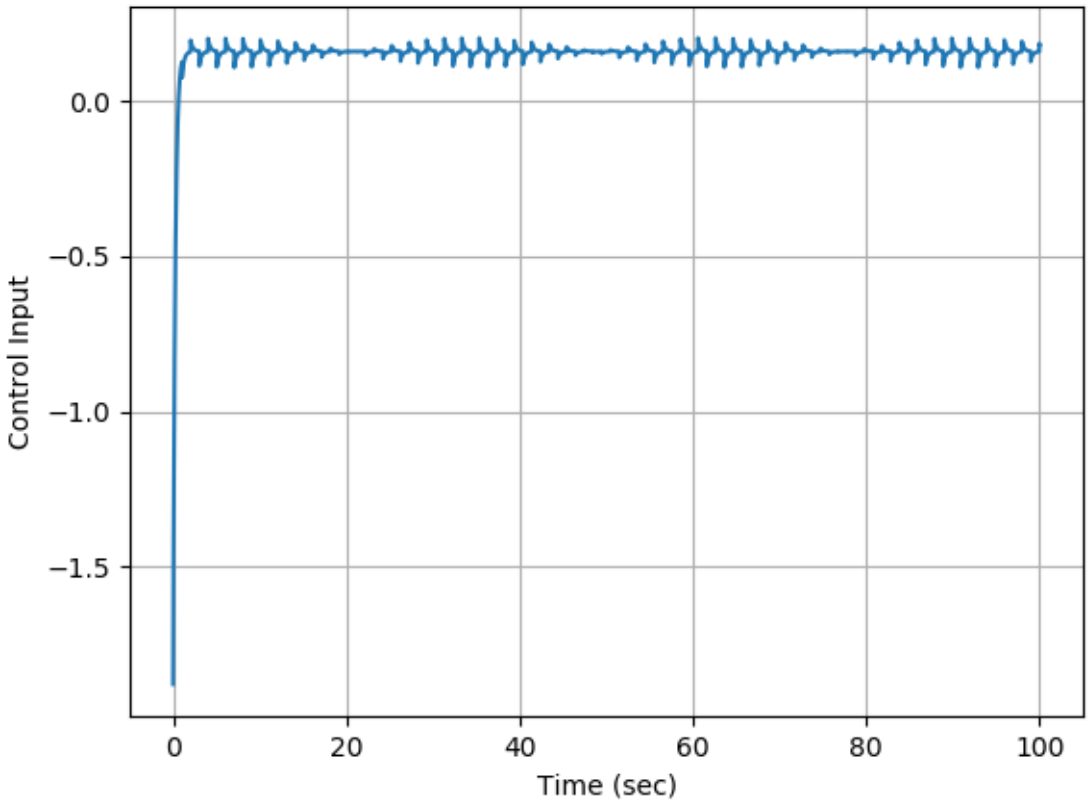

Figure 3: The control input of the system plotted against time

In Fig. 3, the control input of the system plotted against time. The gain of the controller is set to 0.2. The control input starts around -1.8 due to the initial conditions of the system. The control input increase and oscillates around 0.2. This is due to the error of the system being reset each time the Kalman Filter is updated. Upon closer inspection, the system is experiencing aliasing which can be due to the frequency of the controller taken being too high.

3 Appendix:

3.1 Kalman Filter 1st order Code:

```
import numpy as np

import matplotlib.pyplot as plt

SensorPeriod = 1.0

lastSensorTime = -2*SensorPeriod

ftilde = -3.2

def sensor(x,t):
```

```
    global lastSensorTime
    if (t-lastSensorTime) > SensorPeriod:
        lastSensorTime = t
        v = 0.01*np.sin(100*t)
        y = x + v
        return y
    else:
        return -99
def control(xm,t):
    ##Control law - make sure to use the model estimate and not the actual truth signal
    xc = 1.0
    e = xc - xm
    kp = 0.2
    u = kp*e
    return u
def Covariance_Engine(t,p):
    global ftilde
    ##Code in your Covariance Dynamics
    # q = E[ww^T]
    q = 10.0
    pdot = ftilde*p + p*ftilde + q
    return pdot
def Physics_Engine(t,x,u):
    #Disturbance
    w = np.sin(3*t)
    ##Code in your state dynamics
    xdot = -3*x + 4*u + w
    return xdot
def Model_Engine(t,xtilde,u):
```

```
    global ftilde
    ##Code in your model dynamics
    xtildedot = ftilde*xtilde + 3.9*u
    return xtildedot
##Setup time stuff
tinitial = 0
tfinal = 100
timestep = 0.01
#Then we can run the RK4 engine
t = tinitial
print('Begin RK4 Integrator')
x = 10.
xtilde = 10.4
p = 0.2
tout = np.arange(tinitial,tfinal,timestep)
xout = np.zeros((len(tout),1))
xtildeout = 0*xout
pout = 0*xout
uout = 0*tout
ctr = 0
tmeasure = []
ybarout = []
for ctr in range(0,len(tout)):
    ##Call the controller
    u = control(xtilde,t)
    #Save states for plotting
    xout[ctr] = x
    xtildeout[ctr] = xtilde
    pout[ctr] = p
```

```
uout[ctr] = u
##Extract time
t = tout[ctr]
##See if we have a new measurement?
ybar = sensor(x,t)
if (ybar != -99):
    ######Compute K
    #r = E[vv^T]
    r = 0.01
    K = p/(p+r)
    #Get a new model state
    xtilde = xtilde + K*(ybar-xtilde)
    #Get a new covariance
    p = (1-K)*p
    #Save the measurement
    ybarout.append(ybar)
    tmeasure.append(t)
##Integrate the Model Use RK4
k1 = Physics_Engine(t,x,u)
k2 = Physics_Engine(t+timestep/2.0,x+k1*timestep/2.0,u)
k3 = Physics_Engine(t+timestep/2.0,x+k2*timestep/2.0,u)
k4 = Physics_Engine(t+timestep,x+k3*timestep,u)
phi = (1.0/6.0)*(k1 + 2*k2 + 2*k3 + k4)
#Step State
x += phi*timestep
##Integrate the Model Use RK4
k1 = Model_Engine(t,xtilde,u)
k2 = Model_Engine(t+timestep/2.0,xtilde+k1*timestep/2.0,u)
k3 = Model_Engine(t+timestep/2.0,xtilde+k2*timestep/2.0,u)
```

```
    k4 = Model_Engine(t+timestep,xtilde+k3*timestep,u)
    phi = (1.0/6.0)*(k1 + 2*k2 + 2*k3 + k4)
    #Step State
    xtilde += phi*timestep
    ##Integrate the Model Use RK4
    k1 = Covariance_Engine(t,p)
    k2 = Covariance_Engine(t+timestep/2.0,p+k1*timestep/2.0)
    k3 = Covariance_Engine(t+timestep/2.0,p+k2*timestep/2.0)
    k4 = Covariance_Engine(t+timestep,p+k3*timestep)
    phi = (1.0/6.0)*(k1 + 2*k2 + 2*k3 + k4)
    #Step State
    p += phi*timestep
    print('Time =',t)
print('RK4 Integration Complete')
plt.close("all")
#plot Everything
plt.figure()
plt.plot(tout,xout,label='State')
plt.plot(tout,xtildeout,label='Model')
plt.plot(tmeasure,ybarout,'b*',label='Sensor Measurement')
plt.grid()
plt.xlabel('Time (sec)')
plt.ylabel('State')
plt.legend()
plt.figure()
plt.plot(tout,pout)
plt.grid()
plt.xlabel('Time (sec)')
plt.ylabel('Covariance')
```

```
plt.figure()

plt.plot(tout,uout)

plt.grid()

plt.xlabel('Time (sec)')

plt.ylabel('Control Input')

plt.show()
```

3.2 References:

[1] Franklin, G. F., Powell, J. D., and Emami-Naeini, A., *Feedback control of dynamic systems*, Upper Saddle River, NJ: Pearson, 2020.
[2] Slotine, J.-J. E., and Li, W., *Applied nonlinear control*, Taipei: Prentice Education Taiwan Ltd., 2005.