Control and Estimation of a 2 Dimensional Quadrotor

Daksh Dhingra

***Abstract*—This paper present the mathematical design ,controller and Kalman filter implementation on a 2 dimensional quadrotor. Implementation and comparison of two different methods for closing the loop are studied and discussed in detail. Satisfactory state estimates were obtained, a satisfactory controller was deduced and implemented.**

I. Introduction

QUADROTORS are autonomous vehicles which can be programmed to operate without any pilot. Due to its small size,light weight and simple mechanical design it has become a very famous topic for research in industries and academia. It is being currently used for applications like video capturing, crop analyzing, surveillance, reconnaissance and search and rescue missions. However, this is just a tip of the iceberg, drone being most cost effective and computationally advanced unmanned aerial vehicles are finding their use in more and more innovative applications. The vertical take off, landing capability, ability to hover and their compact structures allow them to navigate to places where humans can't intervene. But as the the dynamics of the quad rotor are fast it also requires a fast state estimation which can enable it to perform control actions. The paper aims to design and implement the LQR controller and Kalman filter on a 2-dimensional quad rotor system.

A. System Model

The quad rotor model is derived using the Newton -Euler equations of motions. Solving for these equations of motion will give the state space model of the system.

$$
\begin{aligned}
m\ddot{h} &= u_1 sin\theta \\
m\ddot{v} &= -mg + u_1 \cos\theta \\
I\ddot{\theta} &= u_2
\end{aligned}
$$

here (h, v) represents the horizontal and vertical position of the quadrotor and θ represents the angular rotation of the quadrotor. The vaules (m, I) are mass and moment of inertia which are determines by the body dynamics. g is the acceleration due to gravity and $(u_1,\ u_2)$ denote the net (thrust, torque) applied to the spinning rotors. Note the Inertial measurement unit of the quad rotor will only give us the horizontal and vertical position of the quad and rest of the states will be estimated according to that. These output states given my IMU is represented by the function h which is defined as $y = h(q, \dot{q})$ here q and $\dot{q}$ are the states of the quad.

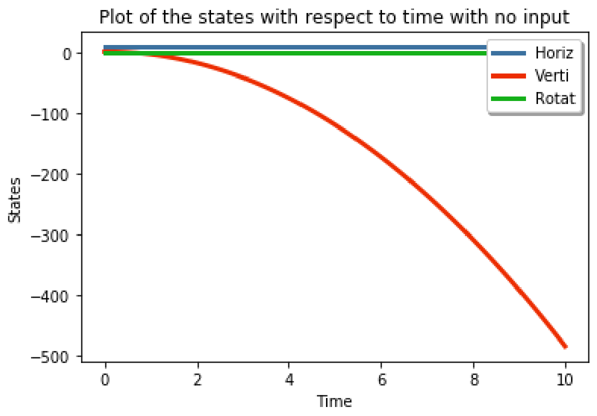


Fig. 1. Plot of state vs time for open loop system (unstable)

The states can be written in matrix form by using the function f which is defined as

$$
\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ F((q,\dot{q}),u) \end{bmatrix} = f((q,\dot{q}),u),\ \ y = h(q,\dot{q})
$$

where $q = (h, v, \theta) \in \mathbb{R}^3$, $u = (u_1, u_2) \in \mathbb{R}^2$, $F : \mathbb{R}^3 \times \mathbb{R}^2 \to \mathbb{R}^3$ is defined by

$$
F((q,\dot{q}),u) = \ddot{q} = \begin{bmatrix} \frac{u_1}{m}\sin\theta \\ -g + \frac{u_1}{m}\cos\theta \\ \frac{u_2}{I} \end{bmatrix},
$$

and $h : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$
h(q,\dot{q}) = (h, v).
$$

B. Open Loop Simulation

For this section the non linear equations were used to simulate the open loop system. The input provided to the dynamics were changed to study the effects it produces on the attitude and position of the quad.

1) Case 1: With zero input: The vertical height of the system with zero input is expected to decrease incessantly because clearly the system is unstable if the control input is not applied. Forward Euler is used to simulate the response of the ordinary differential equations of the system. The response can be seen from figure 1 which matches the intuition.

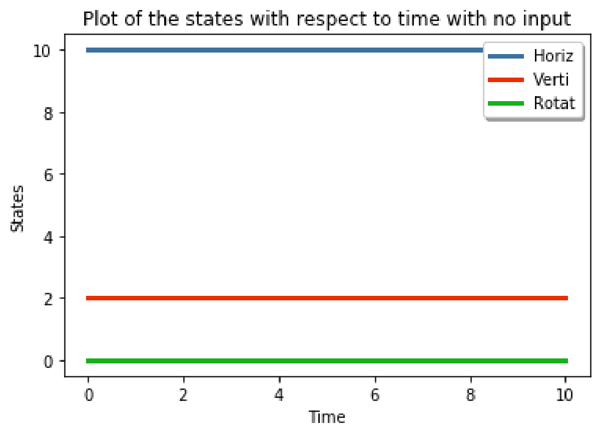


Fig. 2. Plot of state vs time for open loop system (hovering)

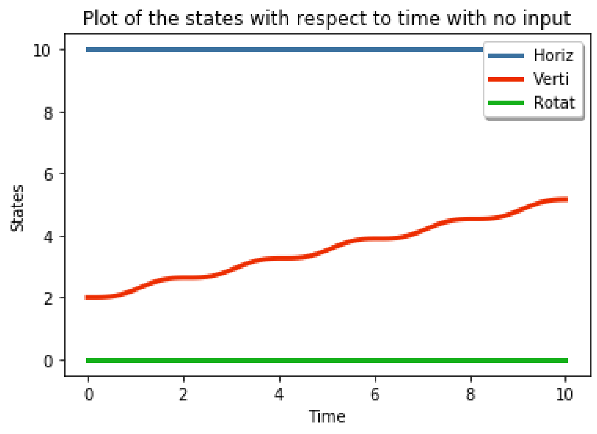


Fig. 3. Plot of state vs time for open loop system (Sine Input)

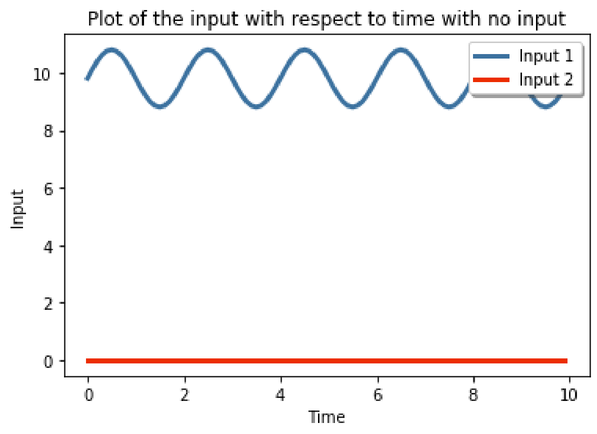


Fig. 4. Plot of Input vs time for open loop system (Sine Input)

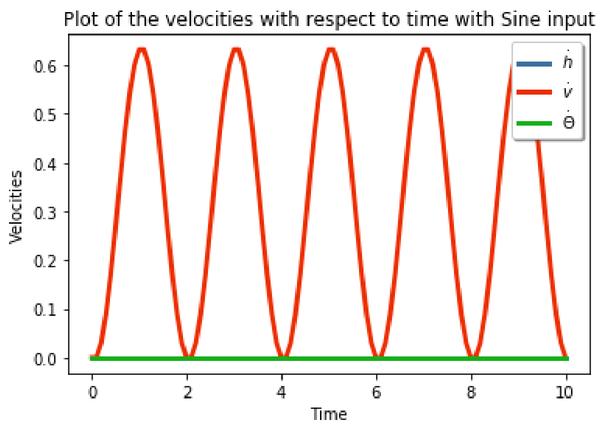


Fig. 5. Plot of Input vs time for open loop system (Sine Input)

2) Case 2: Input for hover : Now if we want the quad to hover we have to apply an input that is equal to the weight of the quadrotor. That means if we apply an input $[mg, 0]$ the quad is supposed to hover. This intuition does match with the observation in Fig2.

3) Case 3: Sinusoidal Input: In this case the quadrotor is applied with a sinusoidal input which is represented by

$$u(t) = \begin{bmatrix} mg + \sin(2t\pi\omega) \\ 0 \end{bmatrix}$$

. here ω is the frequency of the signal and initial condition of the quad is $x_0 = [10, 2, 0, 0, 0, 0]$. The plots given below are for $\omega = 0.5$ Now one interesting observation was noticed here that if we increase the value of frequency. The plot of vertical position started to smoothen out with out less oscillations and at around $\omega = 2.5$ it is almost a straight line. This is because as the frequency increases the fluctuation in input decrease which get further reduced when multiplied by the term $\frac{\sin\theta}{m}$ or $\frac{\cos\theta}{m}$. However, if we increase the time Horizon it just changes the range of plot.

C. Stabilization

Stabilization is done on the system by providing it appropriate feedback which can create an input that would drive the system towards stability. To find that appropriate feedback we need the gain matrix K. To calculate the gain matrix it is important to linearize the system about an equilibrium point. We will linearize the system about the point

$$\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} 0 \\ 0.1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and $u = (mg, 0)$ after linearizing the system we got matrices

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & mg & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1/m & 0 \\ 0 & 1/I \end{bmatrix}$$

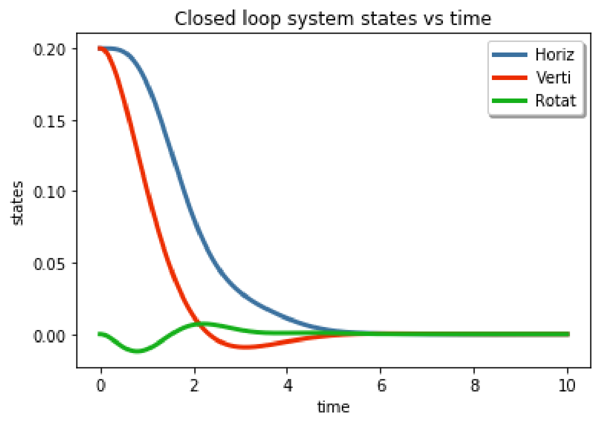


Fig. 6. Plot of state vs time for closed loop system

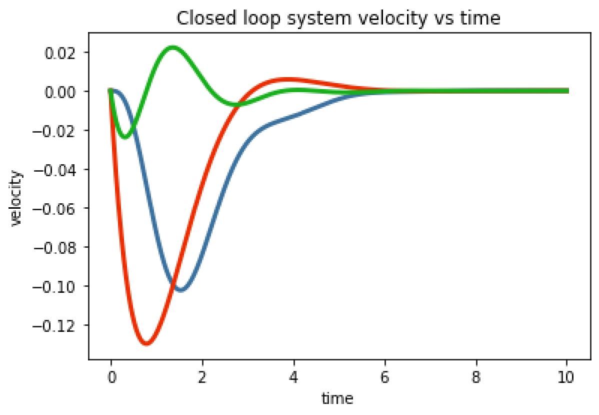


Fig. 7. Plot of velocity vs time for closed loop system

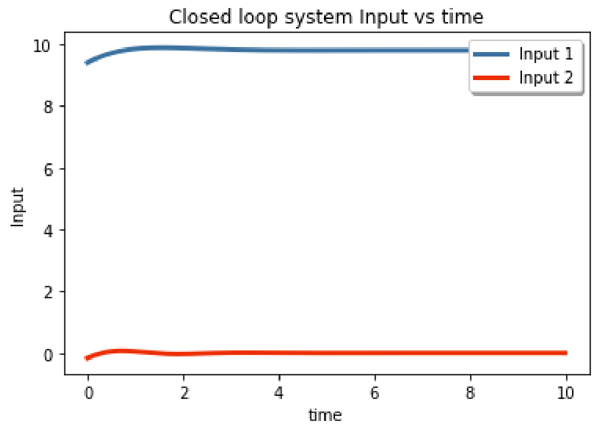


Fig. 8. Plot of Input vs time for closed loop system

The whole process of linearization with every step explained can be found in the Appendix. Now, it is a quadcopter which is designed to hover over things so it make sense to linearize it at a certain height with the thrust equal to the weight of quadrotor and everything at equilibrium. Physically this point with reasonable vertical height does make sense.

D. Controllability

Now as we observed in case 1 above if no input is given the system is clearly unstable. However, if linearized at the input $[mg, 0]$ it is marginally stable at this point i.e. if perturbed in any direction it can become unstable. To pull the system to stability we check the controllability of the system by checking the rank of the controllability matrix. As expected the system is found to be controllable. A separate function `check_ctrb` is made to check the controllability.

E. Feedback Control

As we can understand from the proof in the appendix that if (A, B) is controllable than so is $(-\lambda I - A, B)$ and the second proof shows that we can design a stabilizing K using the lyapunov equation which comes directly from W which is the inverse of steady state P obtained from the algebraic recatti equation. Using these concepts we can directly get the steady state feedback gain K using the function `find_k` from the code. So if we use $\lambda = 1$ we get the feedback gain as

$$K = \begin{bmatrix} 0 & 2 & 0 & 0 & 2 & 0 \\ 0.815 & 0 & 12 & 1.63 & 0 & 4 \end{bmatrix}$$

once this feedback is applied the system is rendered stable with eigen values at -1.

F. closed loop system

So, we perturbed the system at $x_0 = [0.2, 0.2, 0, 0, 0, 0]$ and simulated it using the forward euler technique. The input is at every iteration is updated by using $u = -K * x$. As can be seen from the graphs 6-8 all the states converges to 0. Different initial values were used to check if the system is stable for any perturbation. It is noticed that if $x0 = [10, 10, 10, 0, 0, 0]$ but if it goes beyond that perturbation is too high and system goes unstable

II. Continuous time LQR system

Now an LQR controller is defined by a cost function which is defined by acost function

$$J = x^T(t_1)F(t_1)x^T(t_1) + \int_{t_0}^{t_1} (x^T Q x + u^T R u) dt$$

So to use this cost function we need to decide Q matrix and R matrix which should be positive semi definite matrices. To initialize these positive semi definite matrix a function was written `generate_random_posdef`

These matrices are used to generate P matrix by solving algebraic recatti equation for every step. Algebraic recuatti equation is given by

$$\dot{P} = A^T P + PA + PBR^{-1}B^T P + Q$$

This equation is solved using forward euler iteratively. The value of P obtained is then used to calculate K by using equation

$$K = R^{-1}B^T P$$

Now it is observed that the value of P and K converges to a steady value. These values were compared to the inbuilt solver using same Q and R and found to be the same. These value of optimal K are then used to calculate input at every time $u_t = -K_t x_{t-1}$ and fed to the non linear system.

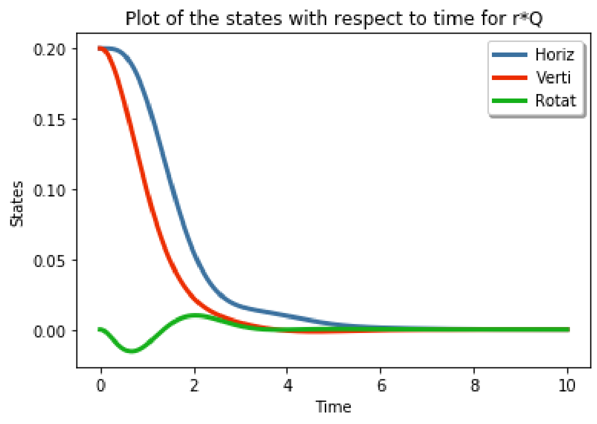


Fig. 9. Plot of state vs time for LQR closed loop system

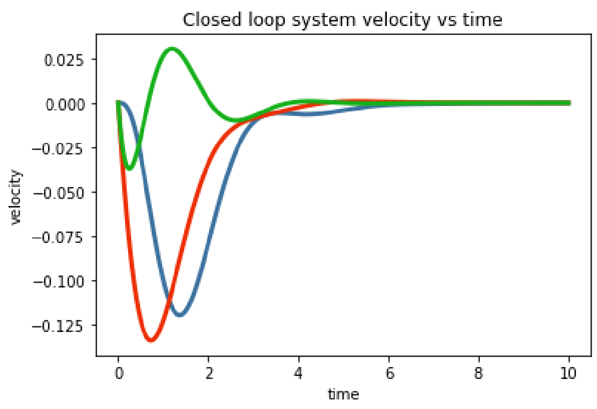


Fig. 10. Plot of velocity vs time for LQR closed loop system

Forward euler is then applied to the system to simulate its dynamics with the optimal u. The graphs obtained can be seen in fig 9-11.

To get an idea of how the states would vary if we keep on increasing Q. A multiplier r is used which is increased in value to show how the convergence of states is effected when Q is increased it turns out the controller become more aggressive as we increase Q. However, as we increase R the controller convergence become slower. It also makes intuitive sense because R penalizes the amount of input we are providing to system and Q penalizes the states.

Yes it is possible to generate a Q and R which would make the system converge like the feedback controller. For this purpose R is needed to be small enough and Q is needed to be big enough so system response can be fast.

III. Kalman Filter

The Kalman filter is an estimation technique which will help us to estimate the value of all those states which are not being measured by inertial measurement unit. We assume here that vertical height and horizontal distance are only quantities that we can measure but that too with a lot of noise. Now our discrete time lman filter will estimate the rest of the outputs. This is done three steps. Firstly, discretization of the closed loop system is done secondly initial states are defined and noise is added to the system and lastly Kalman filter is applied to the system to estimate the states.

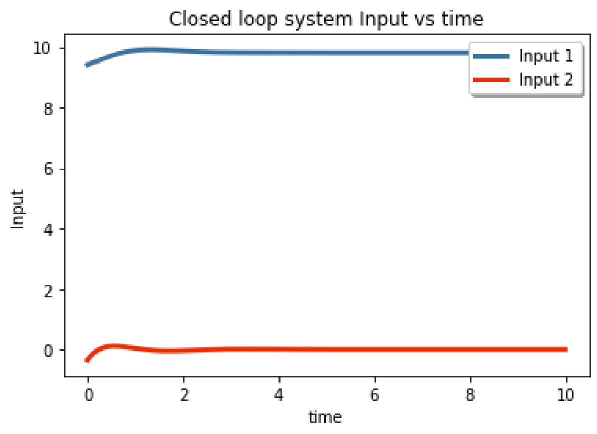


Fig. 11. Plot of Input vs time for LQR closed loop system

A. Discretization

Discretization of the model is done to get discrete A, discrete B discrete C matrices using the equations below.

$$A_d = e^{A-BK}dt$$

and

$$B_d = (A - BK)^{-1}(A_d - I)B$$

$$C_d = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Here the K used is the steady state K we obtained from solving the lyabpunov equation. So if we use the equation

$$x_{t+1} = A_d x_t$$

the system is stable.

B. System Setup

To setup the system before employing Kalman filter to it we defined two initial values of the system

$$x_{0p} = [1, 0, 0, 0, 0, 0]$$

and

$$x_0 \sim \mathcal{N}([0, 0, 0, 0, 0, 0], \Sigma_0)$$

where σ_0 is defines as $\Sigma_0 = \text{diag}(0.1, 0.1, 0.1, 0.1, 0.1, 0.1)$ which is the covariance of the initial condition. The simulation is done for 5 seconds with time step $dt = 0.01$. Now there is a process noise defined as $w_t \sim \mathcal{N}(0, 0.1 I_{2\times 2})$ and measurement noise defined as $v_t \sim \mathcal{N}(0, 0.2 I_{2\times 2})$. So the equations of the system looks like

$$\begin{aligned} x_{t+1} &= A_{\text{cl},d} x_t + F_t w_t \\ y_t &= C_d x_t + H_t v_t \end{aligned}$$

where F_t be the discretized control dynamics B_d and $H_t = I$.

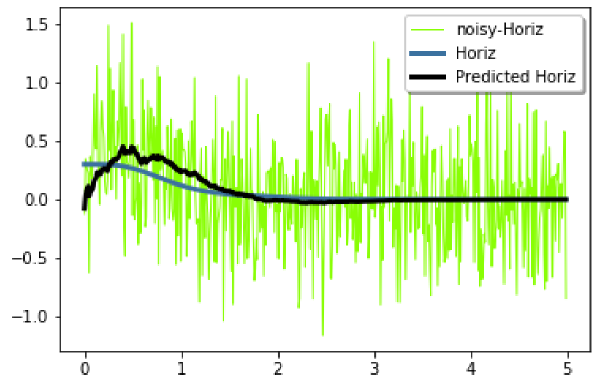


Fig. 12. Plot of actual, estimated and noisy horizontal state vs time

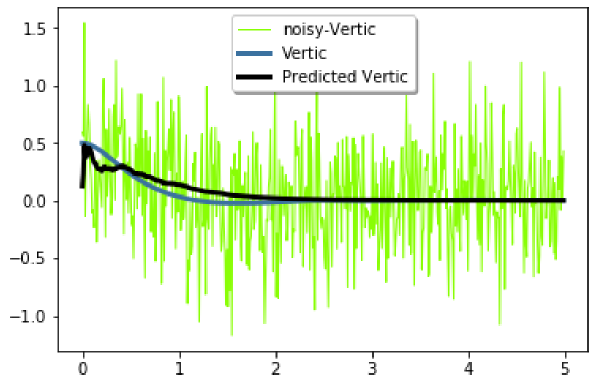


Fig. 13. Plot of actual, estimated and noisy vertical state vs time

C. Kalman Filter

To Implement the Kalman filter following equations are used

$$x_{t+1} = x_t + dt(F_{lin}(t, x_t, u)) + F_t w_t$$

In the above equation x_{t+1} are the actual states with noise forward in time. F_{lin} is the leniarized function which is rand in Forward euler. While using this function we also give it $u = -kx_t$ where k is the steady state k.

$$y_{t+1} = Cx_{t+1} + Hv$$

here y_{t+1} defines the output with noise forward in time

$$x_{\bar{t}+1} = A_d + \bar{x}_t$$

This is initializing the estimated states $x_{\bar{t}+1}$ using the decritized A matrix .

$$\Sigma_{t+1} = A_d \Sigma_t A^T + FQF^T$$

In the above equation we are defining the covariance forward in time using the Ricatti equation.Here Q is covariance of w_t

$$L_t = A_d \Sigma_{t+1} C^T (C\Sigma_{t+1} C^T + HRH^T)^{-1}$$

In the above equatio we calculated the gain of the Kalman filter using the updated covariance. This gain will now be used to update the estimated states. Here R is covariance of v_t

$$x_{\bar{t}+1} = x_{\bar{t}+1} + L_t(y_{t+1} - Cx_{\bar{t}+1})$$

$$\Sigma_{t+1} = (I - L_t C)\Sigma_{t+1}$$

The above equations are updating the value of estimates and covariance for next iteration. The above algorithm is ran forward in time to estimate all the states. The output from the system and the kalman filter are plotted in graph 12-14.

As can be seen from the graph the values of the estimator closely follows the graph of actual y. Even when the signal is very noisy. Th last graph shows that all the states from the estimator starts differently in the beginning but slowly it converges to the actual states.

Now the kalman filter is checked for randomly generated positive semi positive definite values of state and noise covariance. The results are given in graphs 15-17

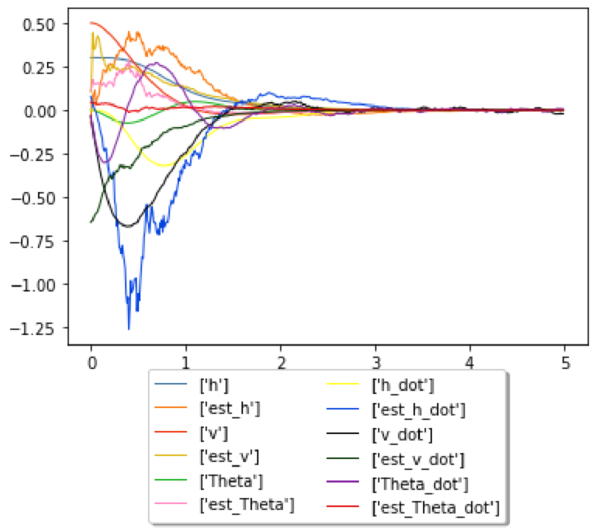


Fig. 14. Plot of actual, estimated states vs time

IV. Conclusion

The model of quadrotor is made and simulated as open loop and closed loop system. Two methods for simulating closed loop system are used first is feed back controller and second is LQR controller. It was observed we can actually control the system response by changing Q and R of the LQR controller. Then an estimator was applied to the system after introducing two noises, measurement noise and process noise into the system. The output of kalman filter was satisfactory as it was able to estimate the states even with all the noise present.

V. Appendices

A. Appendix 1a: Linearization of equations

Given:
$\ddot{h} = \frac{u_1}{m} sin(\theta)$
$\ddot{v} = -g + \frac{u_1}{m} cos(\theta)$
$\ddot{\theta} = \frac{u_2}{I}$
where $q = (h, v, \theta), \dot{q} = (\dot{h}, \dot{v}, \dot{\theta})$

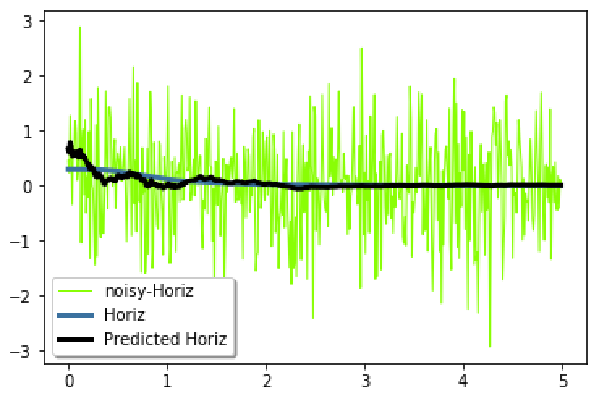


Fig. 15. Plot of actual, estimated and noisy horizontal state vs time

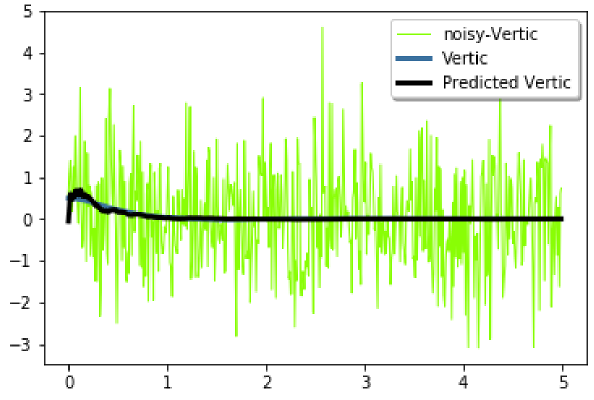


Fig. 16. Plot of actual, estimated and noisy vertical state vs time

Linearize about: $\begin{bmatrix}\tilde{q}\\ \dot{\tilde{q}}\end{bmatrix} = \begin{bmatrix}0\\0.1\\0\\0\\0\\0\end{bmatrix}$; $\tilde{u} = \begin{bmatrix}mg\\0\end{bmatrix}$

Order reduction:

$$f = \frac{d}{dt}\begin{bmatrix}h\\v\\\theta\\\dot{h}\\\dot{v}\\\dot{\theta}\end{bmatrix} = \begin{bmatrix}\dot{h}\\\dot{v}\\\dot{\theta}\\\frac{u_1}{m}sin(\theta)\\-g+u_1cos(\theta)\\\frac{u_2}{I}\end{bmatrix}$$

$$\frac{\partial f(q,\dot{q},u)}{\partial(q,\dot{q})} = \begin{bmatrix}0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\\0&0&\frac{u_1}{m}cos(\theta)&0&0&0\\0&0&-\frac{u_1}{m}sin(\theta)&0&0&0\\0&0&0&0&0&0\end{bmatrix}_{\tilde{q},\dot{\tilde{q}},\tilde{u}}$$

$$\frac{\partial f(q,\dot{q},u)}{\partial u} = \begin{bmatrix}0&0\\0&0\\0&0\\\frac{sin(\theta)}{m}&0\\\frac{cos(\theta)}{m}&0\\0&\frac{u_2}{I}\end{bmatrix}_{\tilde{q},\dot{\tilde{q}},\tilde{u}}$$

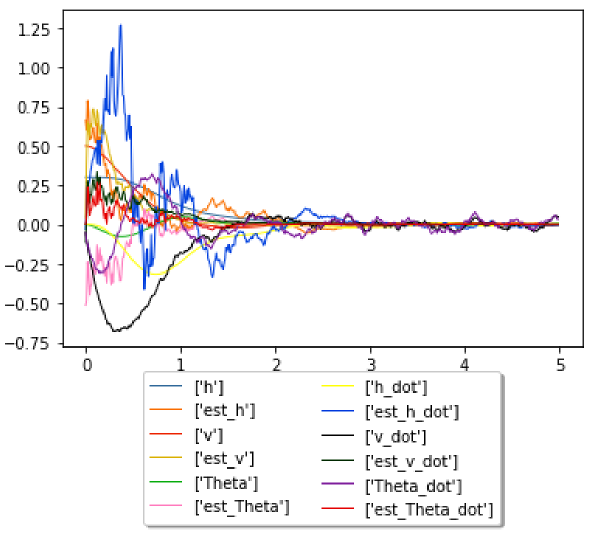


Fig. 17. Plot of actual, estimated states vs time

Apply small angle approximation: $cos(\theta) = 1; sin(\theta) = \theta$, and evaluate about equilibrium points $\tilde{q}, \dot{\tilde{q}}, \tilde{u}$ results in the following linearized state space system.

$$\frac{d}{dt}\begin{bmatrix}h\\v\\\theta\\\dot{h}\\\dot{v}\\\dot{\theta}\end{bmatrix} = \begin{bmatrix}0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\\0&0&g&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\end{bmatrix}\begin{bmatrix}h\\v\\\theta\\\dot{h}\\\dot{v}\\\dot{\theta}\end{bmatrix} + \begin{bmatrix}0&0\\0&0\\0&0\\0&0\\\frac{1}{m}&0\\0&\frac{1}{I}\end{bmatrix}\begin{bmatrix}u_1\\u_2\end{bmatrix}$$

B. Appendix 1b: Proof of Controlability

1) Given :(A,B) is controlable.
2) We can say that
$$A'\lambda = \lambda x$$
where λ is the eigen values of A and x are the eigen vectors.
3) Now we subtract the whole equation with $-\mu Ix$
$$-\mu Ix - Ax = -\mu Ix - \lambda x$$
4) $(-\mu I - A)x = (-\mu I - \lambda)x$
5) By the eigen vector test, which says that if (A,B) is only controllable if every eigen vector of A' is not in kernel of B'.Now, we know that $(-\mu I - A')x$ holds all eigen vectors of A. therefore all eigen vectors of of $(-\mu I - A')$ are NOT in the kernel of B therefore $(-\mu I - A')$ is controllable.

Prove eig vec test:

1) If system is (AB-LTI) is controlable, then all eigenvectors of A' are NOT in the kernel of B'.
2) Assume: There is an eigenvalue A'x = λx with x $/not$ 0, for which B'x = 0
3) Then: $C'x = [B', B'A', ..., B'(A')^{n-1}]^T x = [B'x, \lambda B'x, ..., \lambda^{n-1}B'x] = 0$
4) This means that null space of C has at least one nonzero vector. Therefore, from the fundamental theorem of linear equations, we can say:

5) $dim[ker(C')] > 1$ which directly implies rank C = rank C' = n - rank C'
6) This contradict the uocntrollablity of AB-LTI. Similarly, converse of this will also not hold.
7) therefore all eigenvectors of A' are NOT in the kernel of B'.

Now prove controlability:

C. Appendix 1c: Lambda Place Proof

1) $P(A - BK) + (A - BK)^T P + 2\lambda P = 0$
2) $PA - PBK + A^T P - K^T B^T P + 2\lambda P = 0$
3) $PA - PB(0.5B^T P) + A^T P - PB(0.5B^T P) + 2\lambda P = 0$
4) $P^{-1}PAP^{-1} - P^{-1}PBB^T PP^{-1}0.5 + P^{-1}A^T PP^{-1} - P^{-1}PBB^T PP^{-1}0.5 + P^{-1}2\lambda PP^{-1} = 0$
5) $AP^{-1} - BB^T + P^{-1}\lambda + P^{-1}\lambda = 0$
6) $(A + \lambda I)P^{-1} + P(A + \lambda I) - BB^T = 0$
7) Assume: $W = P^{-1}$
8) $(-\lambda I - A)W + W(-\lambda - A)^T + BB^T = 0$

Project 548

You will

1. Simulate the 2-d quadrotor system
2. Linearize the system
3. Design stabilizing feedback control
4. Design an observer You are not allowed to use any off-the-shelf tools. You must write your own ODE solvers, LQR implementation, KF implementation, etc. The only exception is that you may use lyap. That is the only exception

In [1]:
```
%matplotlib inline
#%pdb on
%run _547
np.set_printoptions(precision=2)
```

control system model

Consider the simplified quadrotor model

$$m\ddot{h} = u_1 \sin\theta,$$
$$m\ddot{v} = -mg + u_1 \cos\theta,$$
$$I\ddot{\theta} = u_2$$

where (h, v) denote the quadrotor (horizontal, vertical) position and θ denotes the quadrotor's rotation, (m, I) denote quadrotor (mass, inertia), g is acceleration due to gravity, and (u_1, u_2) denote the net (thrust, torque) applied by the spinning rotors.

If we measure or observe positions (h, v), e.g. with GPS, then the control system model is

$$\frac{d}{dt}\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \dot{q} \\ F((q,\dot{q}),u) \end{bmatrix} = f((q,\dot{q}),u),\ y = h(q,\dot{q})$$

where $q = (h, v, \theta) \in \mathbb{R}^3$, $u = (u_1, u_2) \in \mathbb{R}^2$, $F : \mathbb{R}^3 \times \mathbb{R}^2 \to \mathbb{R}^3$ is defined by

$$F((q,\dot{q}),u) = \ddot{q} = \begin{bmatrix} \frac{u_1}{m}\sin\theta \\ -g + \frac{u_1}{m}\cos\theta \\ \frac{u_2}{I} \end{bmatrix},$$

and $h : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$h(q,\dot{q}) = (h, v).$$

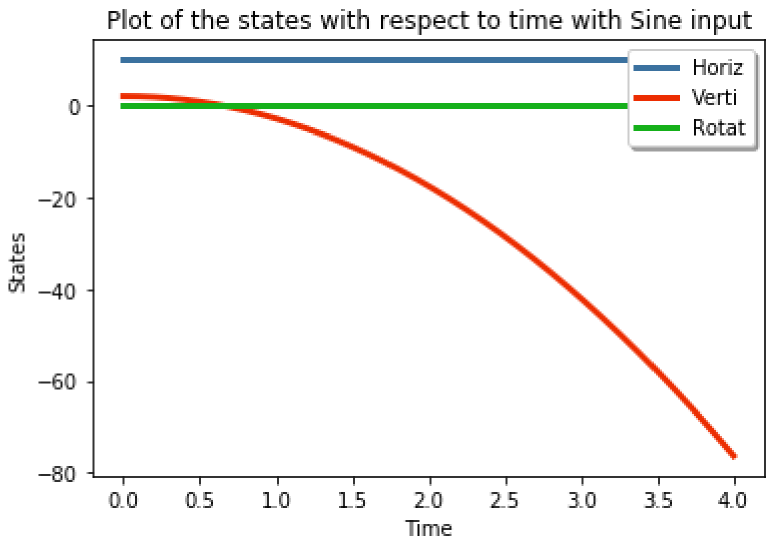
```
[1]: http://dx.doi.org/10.1109/ROBOT.2010.5509452
```

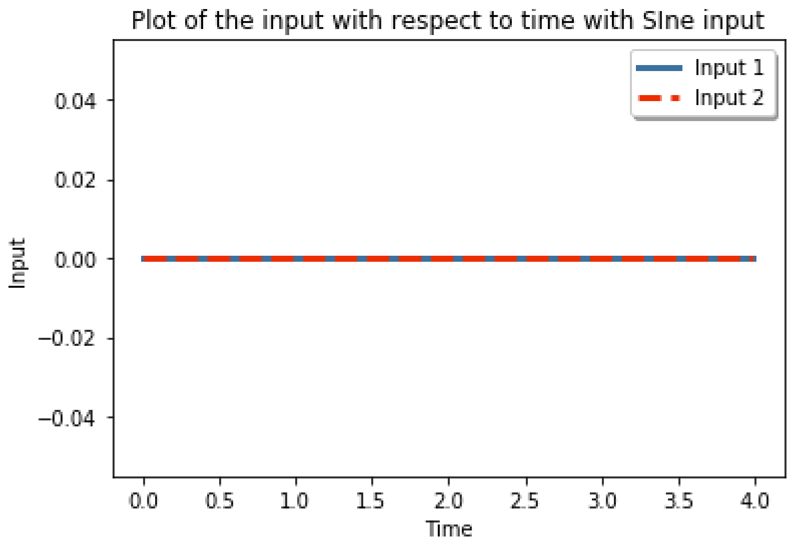
Part 1. Simulation

(a)

To simulate the quadrotor, first implement Python versions of the ODE control system model; that is create two functions, one for f and one for h, that take t, u, x as arguments and return the dynamics $f(t, x, u)$ and the observation $h(t, x, u)$ respectively

In [2]:
```
from _547 import forward_euler
import matplotlib.pyplot as plt
import math
import numpy as np
from seaborn import xkcd_rgb as xkcd
from matplotlib import animation, rc
from numpy import linalg as LA
import control
import scipy
import copy
g,m,I = 9.81,1.,1. # m/sec^2, kg, kg m^2

def f(t,x,u):
    # positions, velocities
    # horiz., vert., rotation
    # thrust, torque
    vector= np.asarray([x[3],x[4],x[5],(u[0]/m)*math.sin(x[2]),(-g+(u[0]
/m)*math.cos(x[2])),u[1]/I])
    return vector# returning the vecotr of the functions given in the qu
estion
def u(t):# function of initial inputs
    return [0,0]

def h(t,x,u):
    # positions, velocities
    # horiz., vert., rotation
    return x[0],x[1]# horizontal, vertical position

def simulate(t):
    t0,x0,t,dt=0,np.array([10,2,0,0,0,0]),t,0.01# defining initial condi
tions
    T,x,uout= forward_euler(f,t,x0,t0,dt,u,return_u=True)# using forward
 euler
    cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]# intersti
ng colors
    # plotting the states
    fig, ax = plt.subplots()
    ax.plot(T,x[:,0],c=cols[0],label='Horiz',linewidth=3)
    ax.plot(T,x[:,1],c=cols[1],label='Verti',linewidth=3)
    ax.plot(T,x[:,2],c=cols[2],label='Rotat',linewidth=3)
    legend = ax.legend(loc='upper right', shadow=True)
    plt.title('Plot of the states with respect to time with Sine input')
    plt.xlabel('Time')
    plt.ylabel('States')
    plt.show()
    #Plotting the inputs
    cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]# intersti
ng colors
    fig, ax = plt.subplots()
    ax.plot(T[0:int(t/dt)],uout[:,0],c=cols[0],label='Input 1',linewidth
=3)
    ax.plot(T[0:int(t/dt)],uout[:,1],'--',c=cols[1],label='Input 2',line
width=3)
    plt.title('Plot of the input with respect to time with SIne input')
    plt.xlabel('Time')
    plt.ylabel('Input')
    legend = ax.legend(loc='upper right', shadow=True)
    plt.show()


    return T,x

T,x=simulate(4)
```

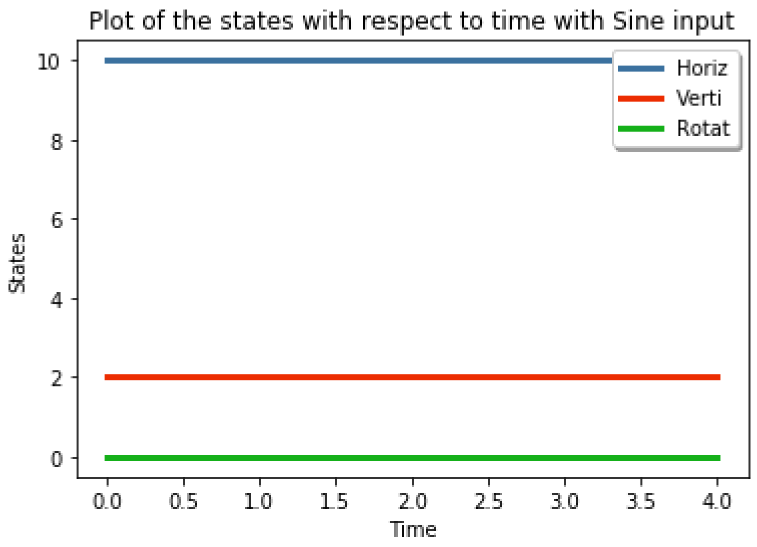


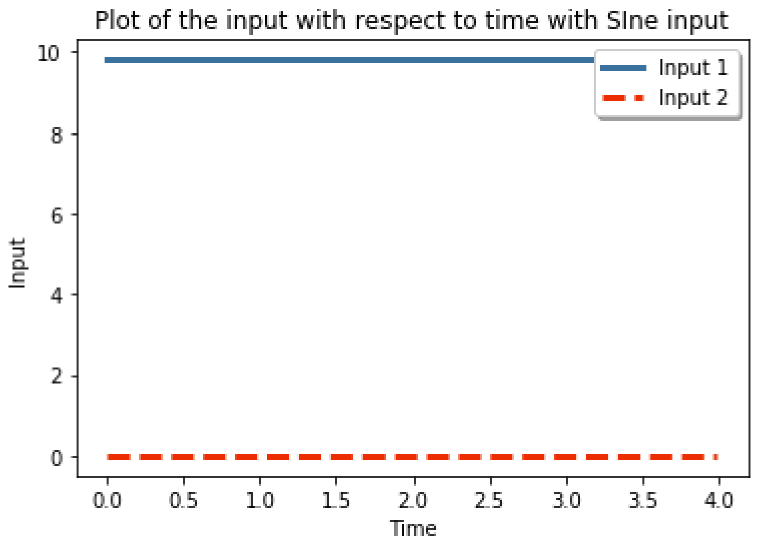


(b)

What is the equilibrium when $u = (mg, 0)$ (i.e. zero torque and thrust is mg)?

In [3]:
```
def u(t):
    return [m*g,0]
T,x=simulate(4)
```





(c)

Simulate the result of applying a sinusoidal thrust of the form

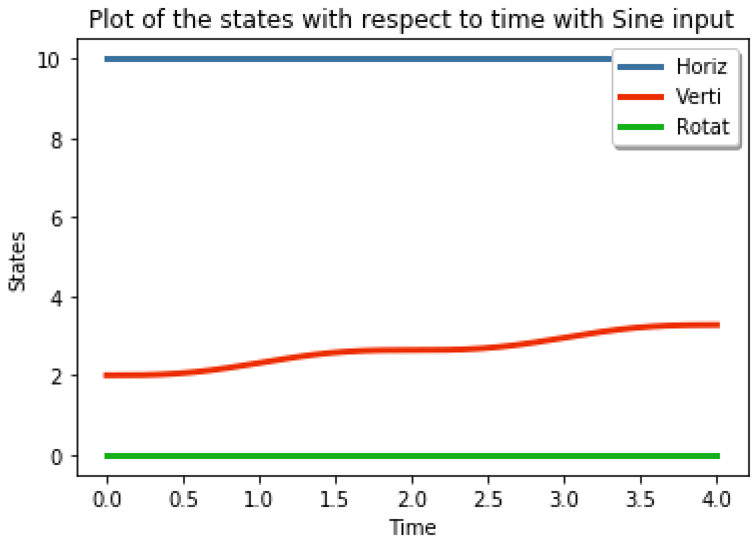
$$u(t) = \begin{bmatrix} mg + \sin(2t\pi\omega) \\ 0 \end{bmatrix}$$

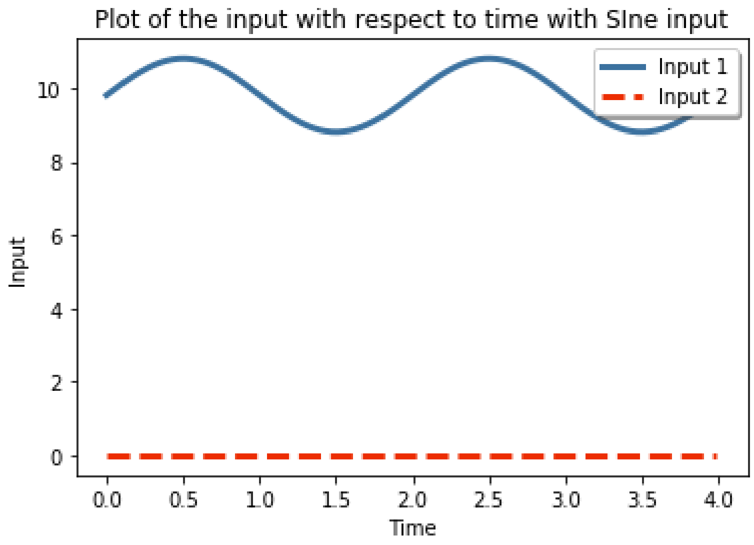
and plot the position (h,v,θ), velocity ($\dot{h}$, $\dot{v}$, $\dot{\theta}$) and input (u_1, u_2) in three seperate plots. I strongly suggest you animate the quadrotor as it will help you visualize the results and better understand what is happening.

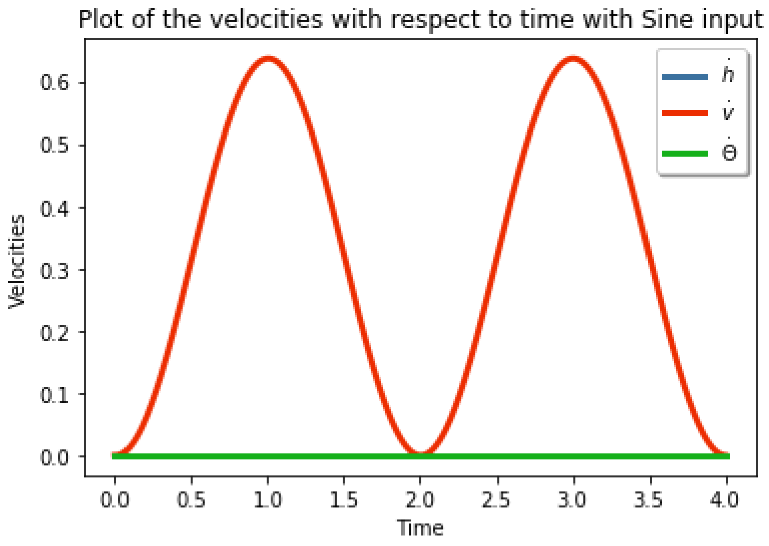
Explore different sinudoidal thrusts and comment on what you observe. That is, choose different frequencies ω and final simulation times. To start, try a frequency of one cycle every two seconds and a final time of two periods. What do you notice about the quadrotors behavior as you vary these aspects?

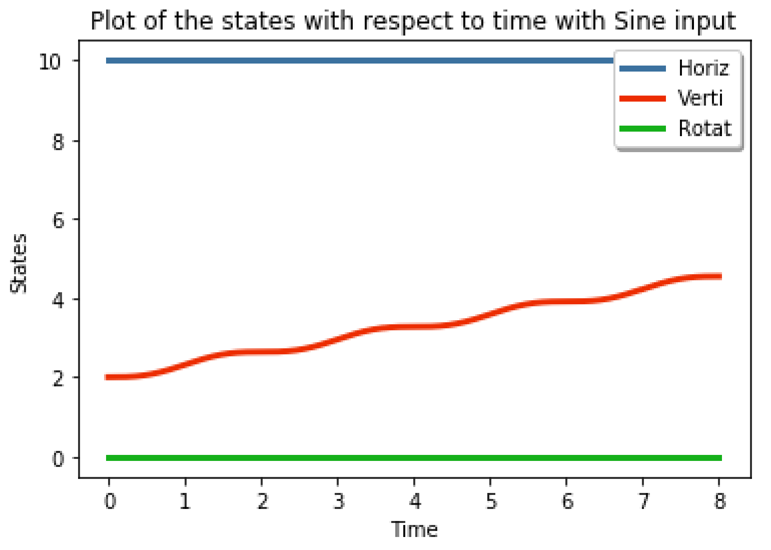
Note you need to choose an appropriate initial conditions $(q(0), \dot{q}(0))$, step size α for your trajectory simulation, time horizon T, etc.

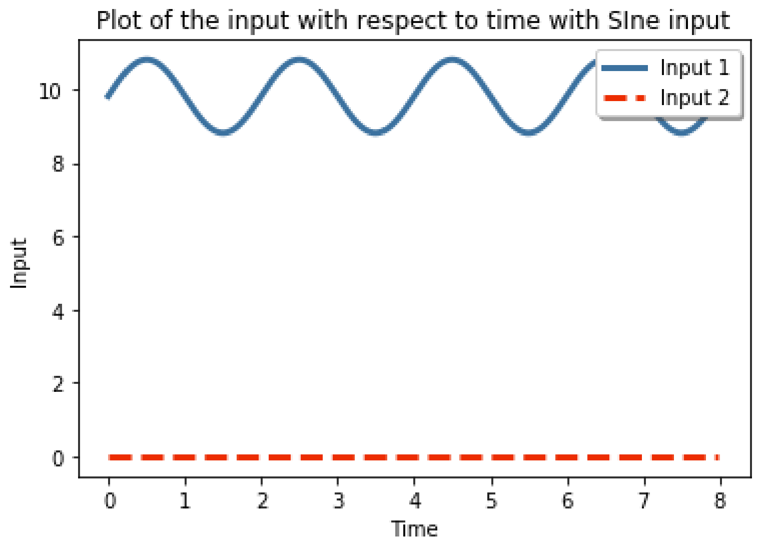
In [4]:
```
freq= [0.5,1]
time= [4,8]
for w in freq:
    for t in time:
        def u(t):
            return [m*g+ math.sin(2*t*math.pi*w),0]
        T,x= simulate(t)
        fig, ax = plt.subplots()
        cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]# inte
rsting colors
        ax.plot(T,x[:,3],c=cols[0],label='$\dot{h}$',linewidth=3)
        ax.plot(T,x[:,4],c=cols[1],label='$\dot{v}$',linewidth=3)
        ax.plot(T,x[:,5],c=cols[2],label='$\dot{\Theta}$',linewidth=3)
        legend = ax.legend(loc='upper right', shadow=True)
        plt.title('Plot of the velocities with respect to time with Sine
 input')
        plt.xlabel('Time')
        plt.ylabel('Velocities')
        plt.show()
```

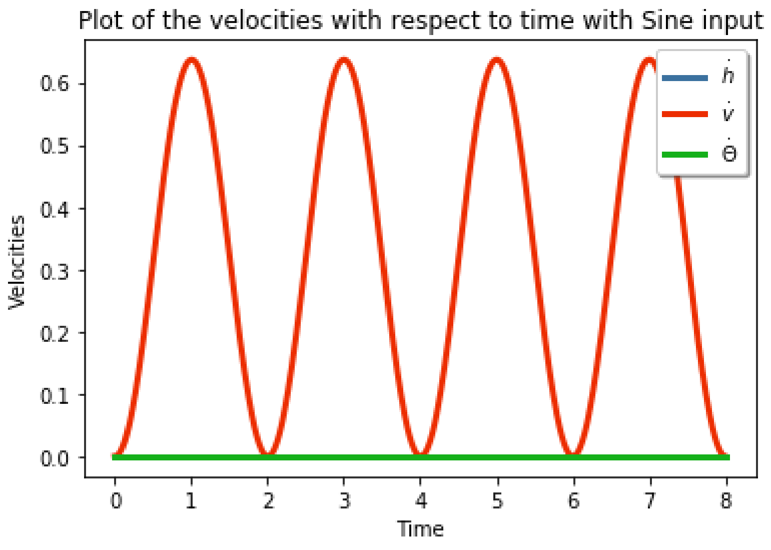

Plot of the states with respect to time with Sine input
States
Time
Horiz
Verti
Rotat
0
2
4
6
8
10
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

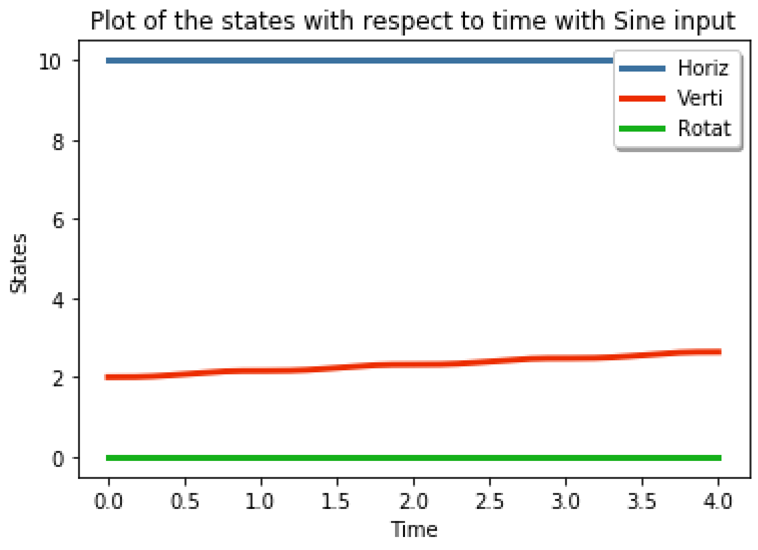

Plot of the input with respect to time with SIne input
Input
Time
Input 1
Input 2
0
2
4
6
8
10
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

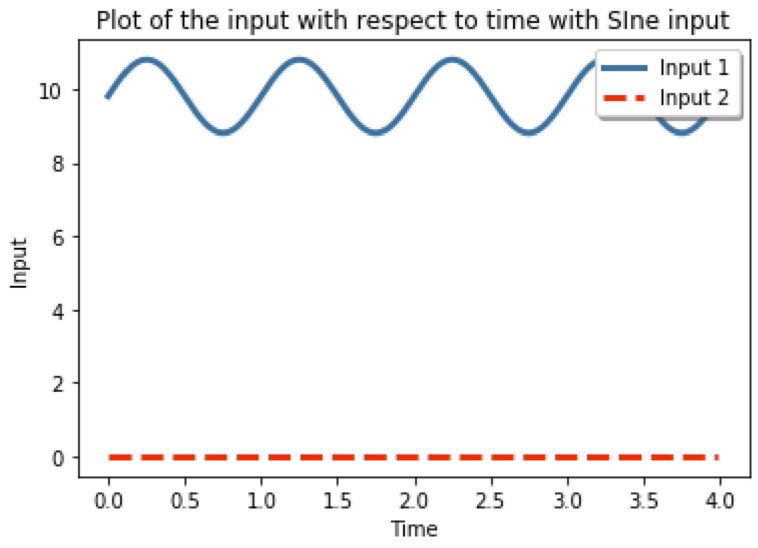

Plot of the velocities with respect to time with Sine input
Velocities
Time
$\dot{h}$
$\dot{v}$
$\dot{\Theta}$
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

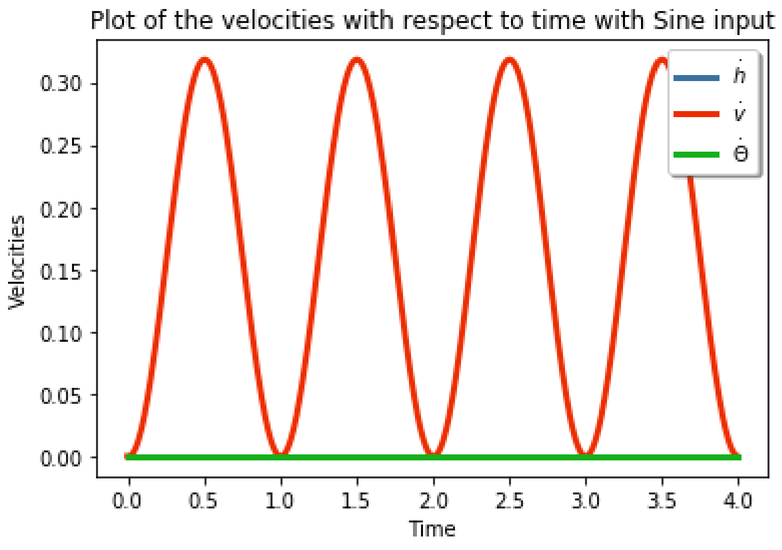

Plot of the states with respect to time with Sine input
States
Time
Horiz
Verti
Rotat
0
2
4
6
8
10
0
1
2
3
4
5
6
7
8

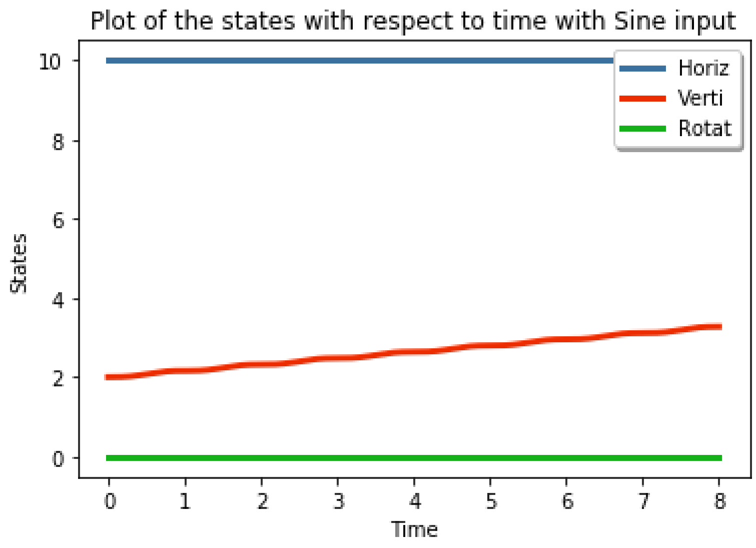

Plot of the input with respect to time with SIne input
Input
Time
Input 1
Input 2
0
2
4
6
8
10
0
1
2
3
4
5
6
7
8

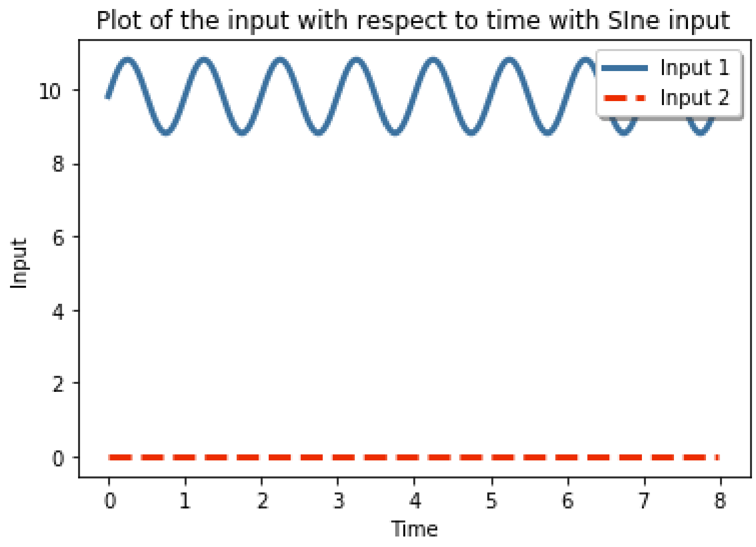

Plot of the velocities with respect to time with Sine input
Velocities
Time
$\dot{h}$
$\dot{v}$
$\dot{\Theta}$
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
1
2
3
4
5
6
7
8

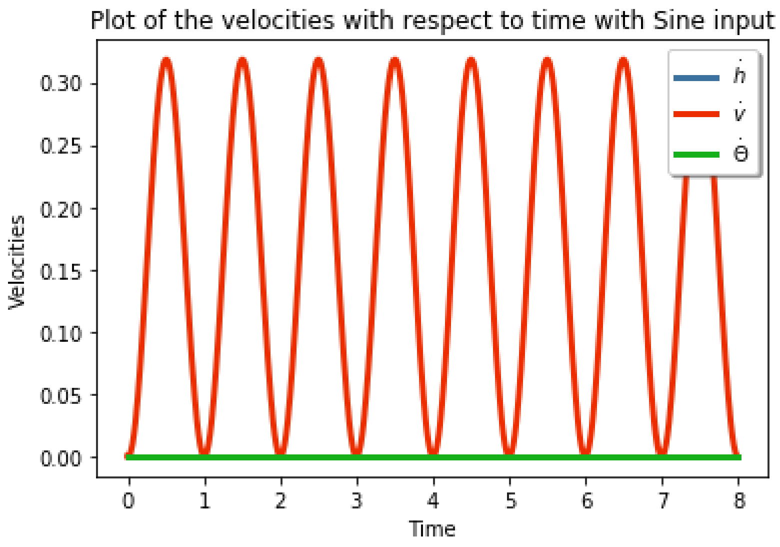

Plot of the states with respect to time with Sine input
States
Time
Horiz
Verti
Rotat
10
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Plot of the input with respect to time with SIne input
Input
Time
Input 1
Input 2
10
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Plot of the velocities with respect to time with Sine input
Velocities
Time
$\dot{h}$
$\dot{v}$
$\dot{\Theta}$
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0







If you want to implement the animation I provided, use this code below.

In [5]:
```
# %run _anim
# fig, ax = plt.subplots(figsize=(4,4)); ax.axis('equal'); ax.grid('o
n');
# line= ax.plot(T, x[:,3], 'b', lw=2);

# plt.close(fig)

# # call the animator
# animation.FuncAnimation(fig, animate, init_func=init, repeat=True,
#                         frames=np.arange(0.,t_[-1],.1), interval=20, b
lit=True)
```

Part 2. Stabilization

In order to stabilize the system, first linearize it.

(a)

Linearize about the point

$$\begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} 0 \\ 0.1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and $u = (mg, 0)$ Provide a full mathematical derivation of your linearization. Thinking back to part 1.a, why is it reasonable to linearize about this point?

In [6]:
```
def f_lin(t,x,u): #function if linearized dynamics
    A=np.asarray([[0,0,0,1,0,0],[0,0,0,0,1,0],[0,0,0,0,0,1],[0,0,m*g,0,0
,0],[0,0,0,0,0,0],[0,0,0,0,0,0]])
    B=np.asarray([[0,0],[0,0],[0,0],[0,0],[1/m,0],[0,1/I]])
    return np.dot(A,x)+np.dot(B,u)
```

(b)

Is the linearized system controllable? Provide justification. That is, use a controllability test.

In [7]:
```
def check_ctrb(A,B):#controllability check.
    n,m=A.shape
    l,q=B.shape
    if n!=m or l!=n:
        print('wrong shape!')
        return 0
    ctrb=B
    for i in range(0,n-1):
        temp=np.dot(LA.matrix_power(A,i+1),B)
        ctrb= np.concatenate((ctrb,temp),axis=1)
    #checking with the inbuilt function
    #ctrbchk=control.ctrb(A,B)
    #if (ctrbchk==ctrb).all:
        #return 0
    #return ctrbchk,ctrb
    #checking the rank
    rank=LA.matrix_rank(ctrb)
    r,c=ctrb.shape
    if rank==r:
        print("Full row rank matrix-> Controllable")
        return 1
    return 0

x1,u=np.array([0,0.1,0,0,0,0]),[m*g,0]
A=np.asarray([[0,0,0,1,0,0],[0,0,0,0,1,0],[0,0,0,0,0,1],[0,0,u[0]/m,0,0,
0],[0,0,0,-u[0]/m*x1[2],0,0],[0,0,0,0,0,0]])
B=np.asarray([[0,0],[0,0],[0,0],[x1[2]/m,0],[1/m,0],[0,1/I]])
status=check_ctrb(A,B)
print(LA.eigvals(A))
```

```
Full row rank matrix-> Controllable
[ 0.  0.  0.  0.  0.  0.]
```

(c)

Design a stabilizing state feedback control law using Lyapunov.

1. First, show that if (A, B) is controllable so is $(-\lambda I - A, B)$ for every $\lambda \in \mathbb{R}$.
2. Choose λ sufficiently large so that $-\lambda I - A$ is stable. Starting with
$$P(A - BK) + (A - BK)^T P + 2\lambda P = 0, \;\; K = \frac{1}{2}B^T P$$
use the Lyapunov stability theorem to show that you can solve
$$(-\lambda I - A)W + W(-\lambda I - A)^T + BB^T = 0$$
to find a K that stabilizes the system.
3. Numerically design K using the above analysis. And provide a check that the closed loop system is stable (that is, check the eigenvalues).

```
In [8]: def find_k(A,B):
            la=1
            n,m= A.shape
            newA= -la*np.identity(n)-A
            # check if stable

            status= check_ctrb(newA,B)
            if status==1:
                #print(newA)
                #print(B)
                W= np.array(scipy.linalg.solve_lyapunov(newA,np.dot(B,np.transpo
        se(B))))
                #print(W)
                P=LA.inv(W)
                #print(P)
                K= -1/2*np.dot(np.transpose(B),P)
                p,r=K.shape
                Kvalue=np.zeros((p,r))
                for i in range(p):
                    for j in range(r):
                        Kvalue[i,j]= float(K[i,j])
                return Kvalue,newA

        if status == 1: #if its the controllable
            K,newA= find_k(A,B)
            print(K)
            NewA= A-np.dot(B,K)
            E=np.real(LA.eigvals(NewA))

            if np.all(E<0):
                print('system is stable')
```

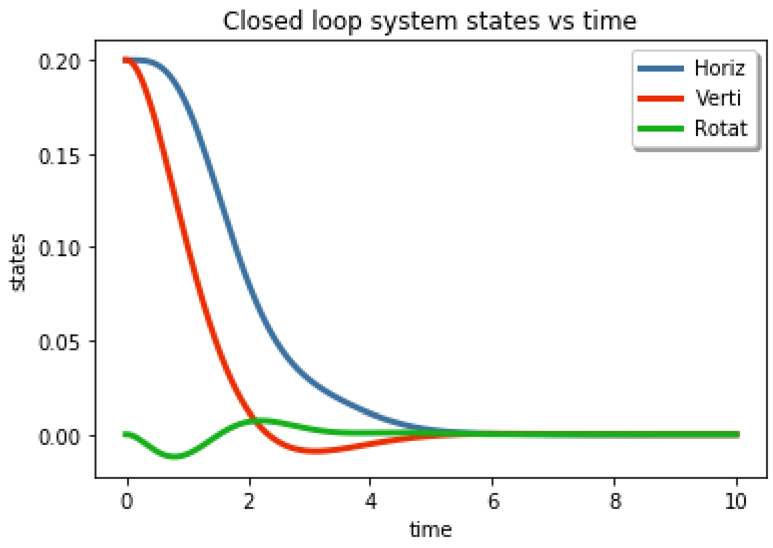
```
Full row rank matrix-> Controllable
[[ -0.     2.    -0.    -0.     2.    -0.  ]
 [  0.82  -0.    12.     1.63  -0.     4.  ]]
system is stable
```

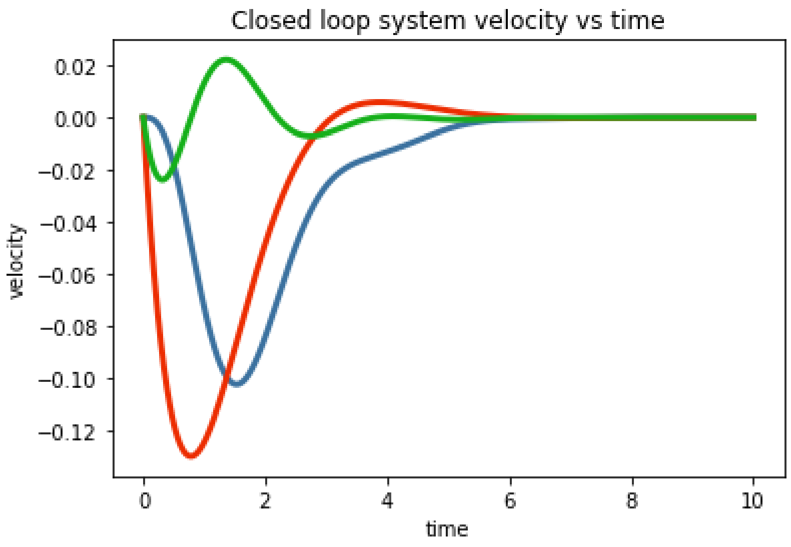
(d)

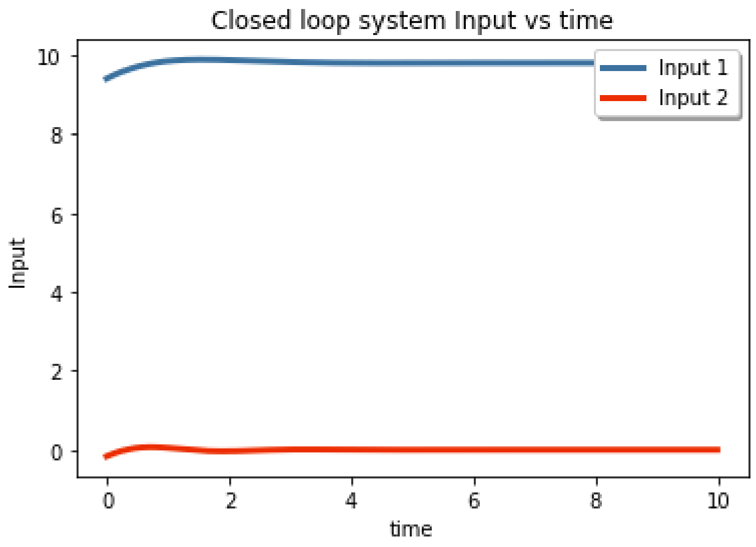
Simulate the nonlinear closed loop system using the designed stabilizing feedback control for a random initializaton in the neighborhood of the point (x_0, u_0) around which you linearized the system.

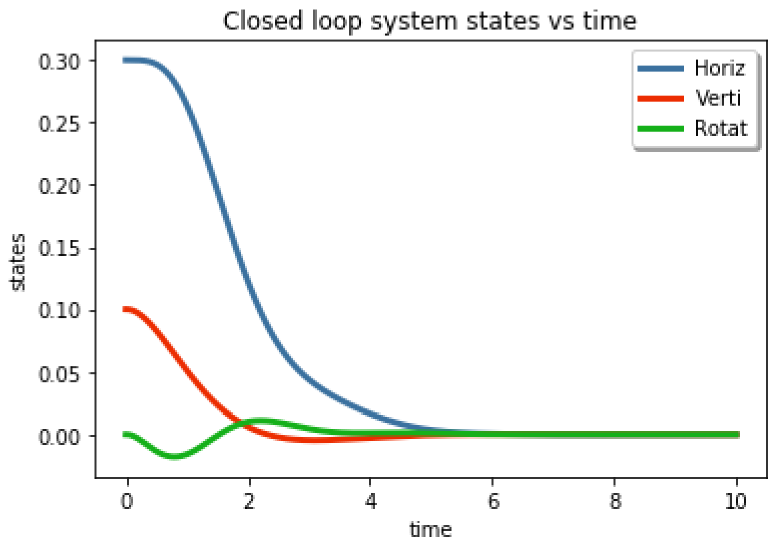
1. That is, you should perturb x_0 and use that as your initilization and apply the designed control input to the nonlinear system in a neighborhood of the linearization point.
2. Try a few different initilizations and describe what you see.
3. As before, plot the position (h,v,θ), velocity $(\dot{h}, \dot{v}, \dot{\theta})$ and input (u_1, u_2) in three seperate plots. Animation will help you visualize what is happening.

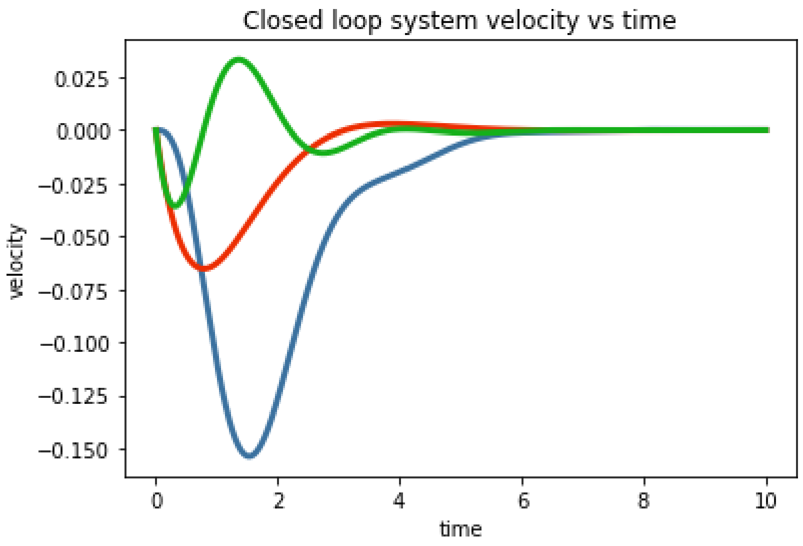
In [9]:
```
def u(x):
    s=np.dot(-K,x)
    return s+[m*g,0]
def simulatex(x0):
    t0,t,dt=0,10,0.01
    T,x,uout= forward_euler(f,t,x0,t0,dt,ux=u,return_u=True)
    cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]

    fig, ax = plt.subplots()
    ax.plot(T,x[:,0],c=cols[0],label='Horiz',linewidth=3)
    ax.plot(T,x[:,1],c=cols[1],label='Verti',linewidth=3)
    ax.plot(T,x[:,2],c=cols[2],label='Rotat',linewidth=3)
    plt.title('Closed loop system states vs time')
    plt.xlabel('time')
    plt.ylabel('states')
    legend = ax.legend(loc='upper right', shadow=True)
    plt.show()

    fig, ax2 = plt.subplots()
    ax2.plot(T,x[:,3],c=cols[0],label='Horiz',linewidth=3)
    ax2.plot(T,x[:,4],c=cols[1],label='Verti',linewidth=3)
    ax2.plot(T,x[:,5],c=cols[2],label='Rotat',linewidth=3)
    plt.title('Closed loop system velocity vs time')
    plt.xlabel('time')
    plt.ylabel('velocity')
    plt.show()

    fig, ax3 = plt.subplots()
    ax3.plot(T[0:1000],uout[:,0],c=cols[0],label='Input 1',linewidth=3)
    ax3.plot(T[0:1000],uout[:,1],c=cols[1],label='Input 2',linewidth=3)
    plt.title('Closed loop system Input vs time')
    plt.xlabel('time')
    plt.ylabel('Input')

    legend = ax3.legend(loc='upper right', shadow=True)
    plt.show()
    return T,x
# def forwardeuler(t0,t,f,x0,dt,u):
#     steps= np.arange(t0,t,dt)
#     n=x0.shape
#     xout=[]
#     #xout= np.zeros((int(n[0]),int(t/dt)))
#     for i in steps:
#         #print(x0,u)]
#         #print(K,'u',u)
#         u=np.dot(-K,x0)
#         out= x0 + dt*f(i,x0,u)
#         xout.append(out)
#         x0=out
#         #print(out)
#     return steps, xout
# t0,x0,t,dt,u=0,np.array([0.1,0.1,0.1,0.1,0.1,0.2]),5,0.01,[0,0]
# tout,xout= forwardeuler(t0,t,f,x0,dt,u)
# cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]
# fig, ax = plt.subplots()
# arrayx=np.array(xout)
# ax.plot(tout,arrayx[:,0],c=cols[0],label='Horiz',linewidth=1)
# ax.plot(tout,arrayx[:,1],c=cols[1],label='Verti',linewidth=3)
# ax.plot(tout,arrayx[:,2],c=cols[2],label='Rotat',linewidth=3)
# legend = ax.legend(loc='upper right', shadow=True)
# plt.show()
x0=np.array([0.2,0.2,0,0,0,0])
T,x=simulatex(x0)
x0=np.array([0.3,0.1,0,0,0,0])
T,x=simulatex(x0)
x0=np.array([10,10,10,0,0,0])
T,x=simulatex(x0)
```

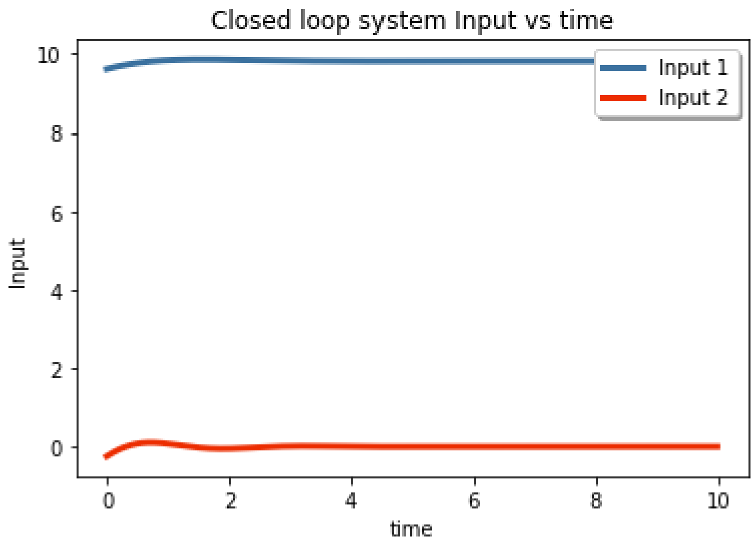

Closed loop system states vs time
states
time
Horiz
Verti
Rotat
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10

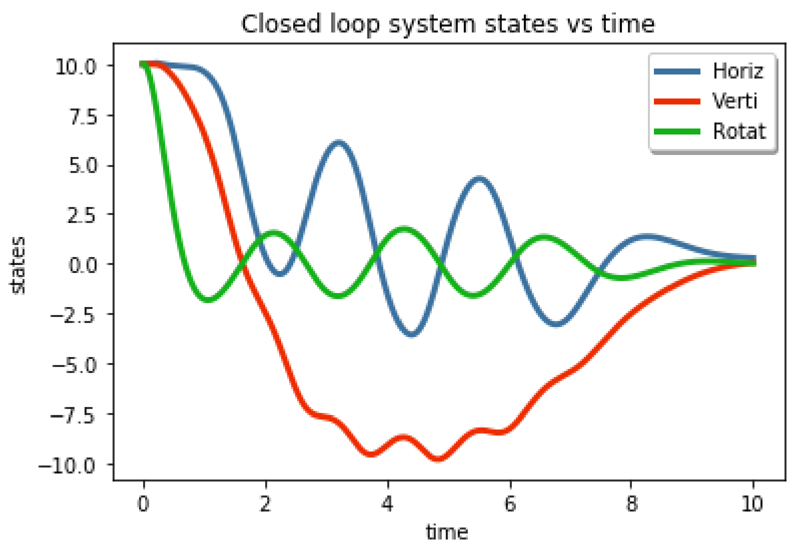

Closed loop system velocity vs time
velocity
time
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
0
2
4
6
8
10

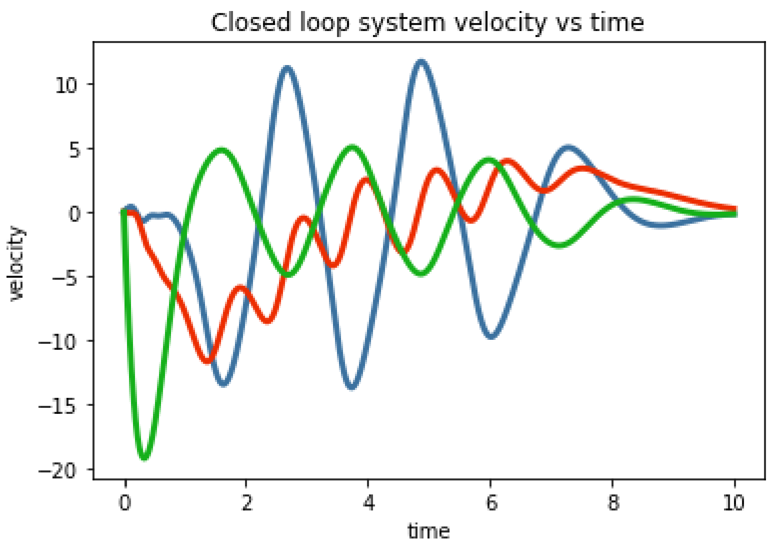

Closed loop system Input vs time
Input
time
Input 1
Input 2
10
8
6
4
2
0
0
2
4
6
8
10

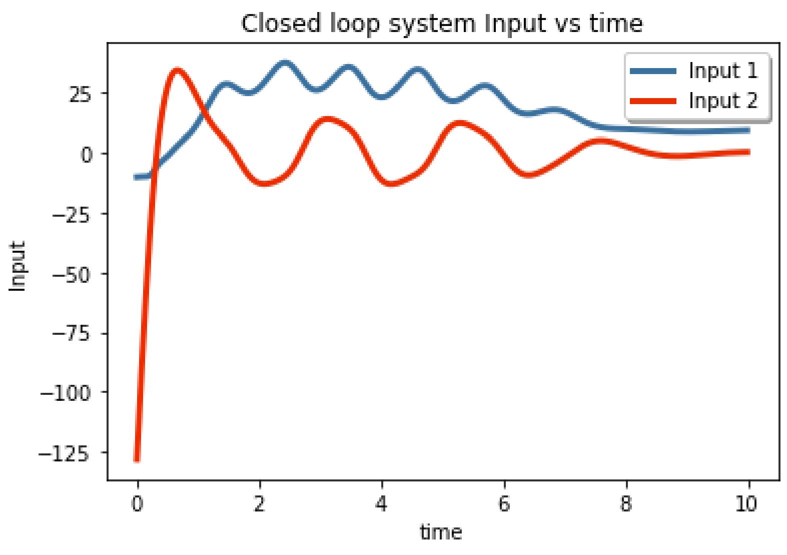

Closed loop system states vs time
states
time
Horiz
Verti
Rotat
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10


Closed loop system velocity vs time
velocity
time
0.025
0.000
−0.025
−0.050
−0.075
−0.100
−0.125
−0.150
0
2
4
6
8
10


Closed loop system Input vs time
Input
time
Input 1
Input 2
10
8
6
4
2
0
0
2
4
6
8
10







Part 3. Continuous Time LQR Design

In this part you will design an LQR controller.

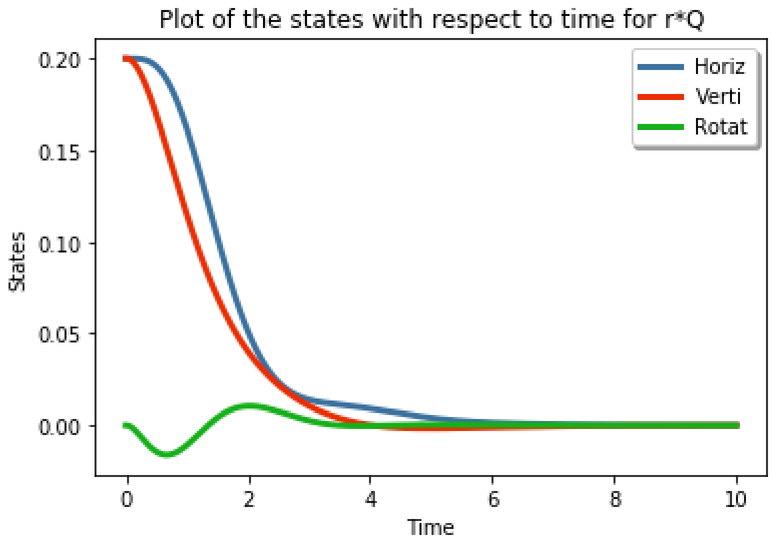
(a)

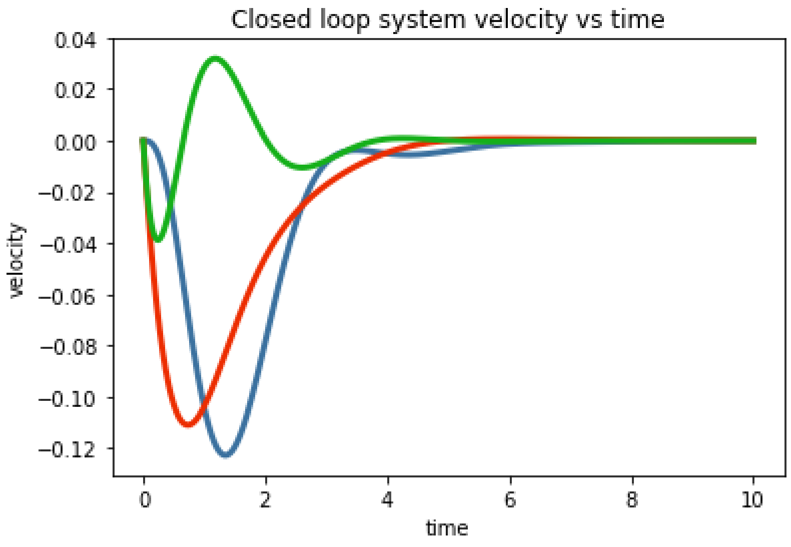
1. Write a function to randomly generate positive semidefinite matrices. (hint: look at _547.py it should be in there already)
2. Write your own numerical scheme to solve LQR. That is, you will need to write your own script to numerically solve the Riccati recursion.

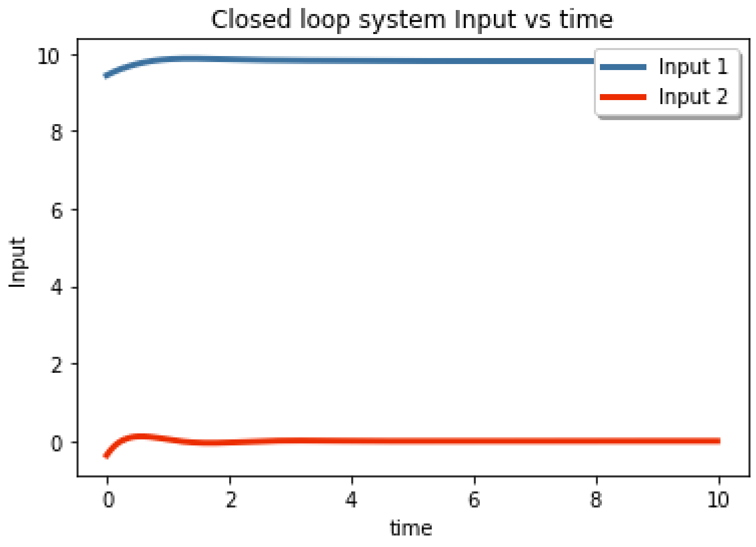
Hint: write a function to vectorize your matrix differential equation and use the forward_euler implementation you already are using for simulating systems. You can check that you have the correct answer by comparing to an lqr solver, but you have to implement your own method for solving lqr. You CANNOT just use an off-the-shelf tool.

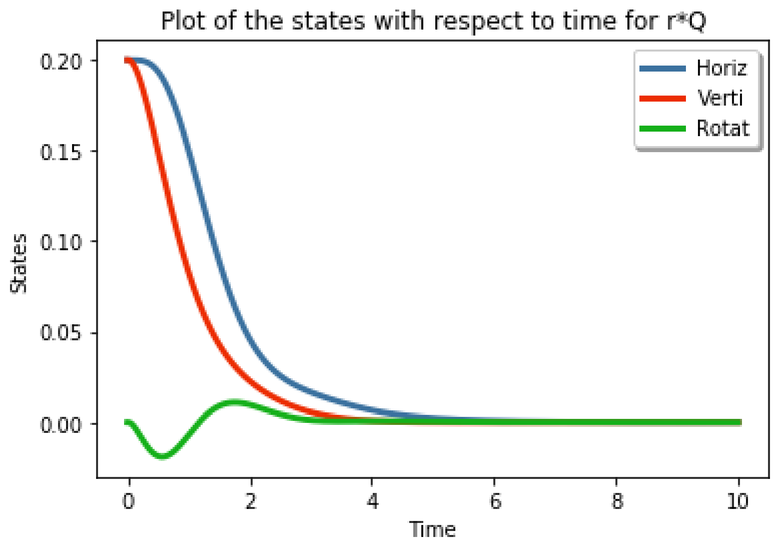
In [10]:
```
# generate random positive matrix
def generate_random_posdef(size):
    temp = scipy.random.rand(size,size)
    B = np.dot(temp,temp.transpose())
    return B+np.identity(size)

def LQRscheme(A,Q,R,B,t0,t,dt):
    steps= np.arange(t0,t,dt)
    P = Q
    finalP=[]
    finalK=[]
    count1=0
    for i in steps:

        dp= P +dt*(np.dot(A.transpose(),P)+np.dot(P,A)- np.dot(P,np.dot(
B,np.dot(LA.inv(R),np.dot(B.transpose(),P))))+Q)
        P=dp
        finalP.append(dp)
        count1+=1
    count=0

    for i in steps:
        finalK.append(np.dot(LA.inv(R),np.dot(B.transpose(),np.array(fin
alP[count1-count-1]))))
        count+=1
    return np.asarray(finalK), np.asarray(finalP)
Q1= generate_random_posdef(A.shape[0])
R= generate_random_posdef(2)
for r in np.arange(1,10,2.5):
    Q=r*Q1
    #R=np.identity(2)
    t0,t,dt=0,10,0.01
    lqrk,lqrP= LQRscheme(A,Q,R,B,t0,t,dt)

    # for compare
    def tvCTLQR(A,B,Q,R,Pt,tf,dt=dt):
        t_ = 0.0
        K = []
        P = []
        while t_ < tf:
            P_ = scipy.linalg.solve_continuous_are(A,B,Q,R)
            K_ = np.dot(LA.inv(R), np.dot(B.T,P_))
            K.append(K_)
            P.append(P_)
            t_ += dt
        K_ = np.dot(LA.inv(R), np.dot(B.T,Pt))
        K.append(K_)
        P.append(Pt)
        return np.asarray(K),np.asarray(P)
    lqrKac,lqrPac=tvCTLQR(A,B,Q,R,t0,t,dt)
    #print("lqrKac",lqrKac[502,:,:])
    #print("lqrk",lqrk[500,:,:])
    #fig,ax=plt.subplots()
    #ax.plot(np.arange(t0,t,dt),lqrk[:,0,6])
    #plt.show()
    #print('lqrk',lqrk[490:500,:,:],'lqrKac',lqrKac[490:502,:,:])
```

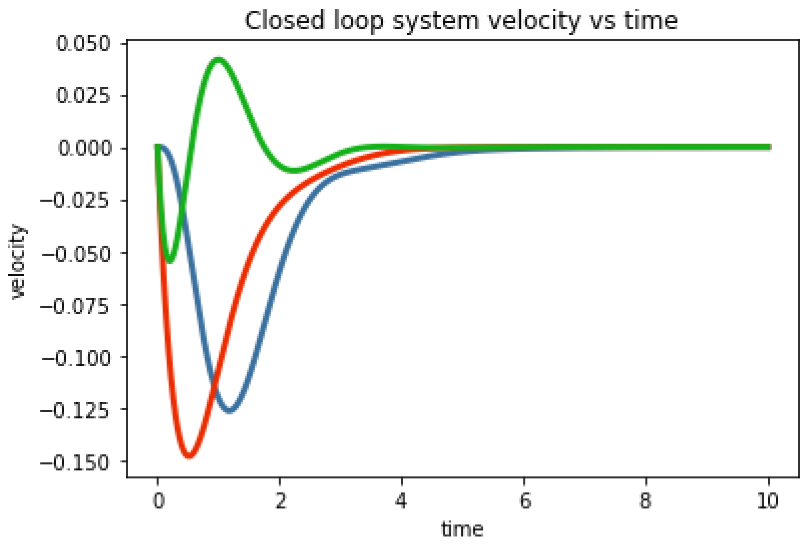
```
def u(t,x):
    dt=0.01
    s=np.dot(-lqrKac[int(t/dt),:,:],x)
    return s+[m*g,0]
def simulatex(r):
    t0,x0,t,dt=0,np.array([0.2,0.2,0,0,0,0]),10,0.01
    T,x,uout= forward_euler(f,t,x0,t0,dt,utx=u,return_u= True)
    cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]
    fig, ax = plt.subplots()
    ax.plot(T,x[:,0],c=cols[0],label='Horiz',linewidth=3)
    ax.plot(T,x[:,1],c=cols[1],label='Verti',linewidth=3)
    ax.plot(T,x[:,2],c=cols[2],label='Rotat',linewidth=3)
    legend = ax.legend(loc='upper right', shadow=True)
    plt.title('Plot of the states with respect to time for r*Q')
    plt.xlabel('Time')
    plt.ylabel('States')
    plt.show()

    fig, ax2 = plt.subplots()
    ax2.plot(T,x[:,3],c=cols[0],label='Horiz',linewidth=3)
    ax2.plot(T,x[:,4],c=cols[1],label='Verti',linewidth=3)
    ax2.plot(T,x[:,5],c=cols[2],label='Rotat',linewidth=3)
    plt.title('Closed loop system velocity vs time')
    plt.xlabel('time')
    plt.ylabel('velocity')
    plt.show()

    fig, ax3 = plt.subplots()
    ax3.plot(T[0:1000],uout[:,0],c=cols[0],label='Input 1',linewidth
=3)
    ax3.plot(T[0:1000],uout[:,1],c=cols[1],label='Input 2',linewidth
=3)
    plt.title('Closed loop system Input vs time')
    plt.xlabel('time')
    plt.ylabel('Input')

    legend = ax3.legend(loc='upper right', shadow=True)
    plt.show()
    return 1
a=simulatex(r)
```

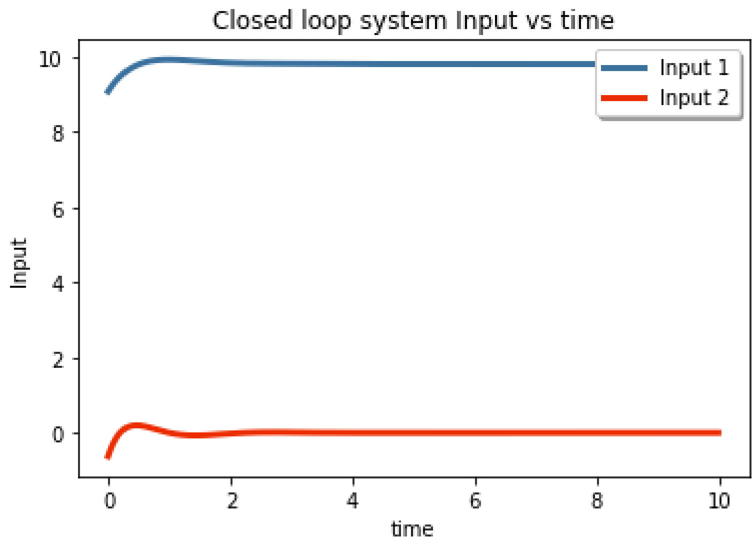

Plot of the states with respect to time for r*Q
States
Time
Horiz
Verti
Rotat
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10

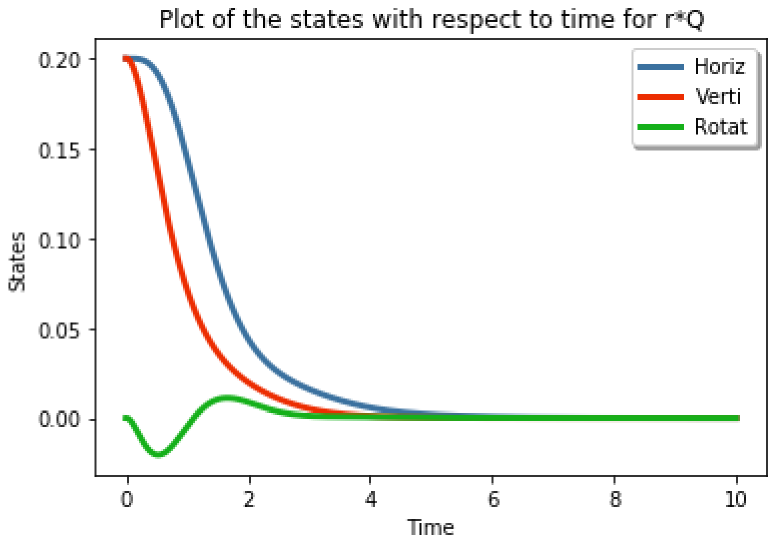

Closed loop system velocity vs time
velocity
time
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
0
2
4
6
8
10

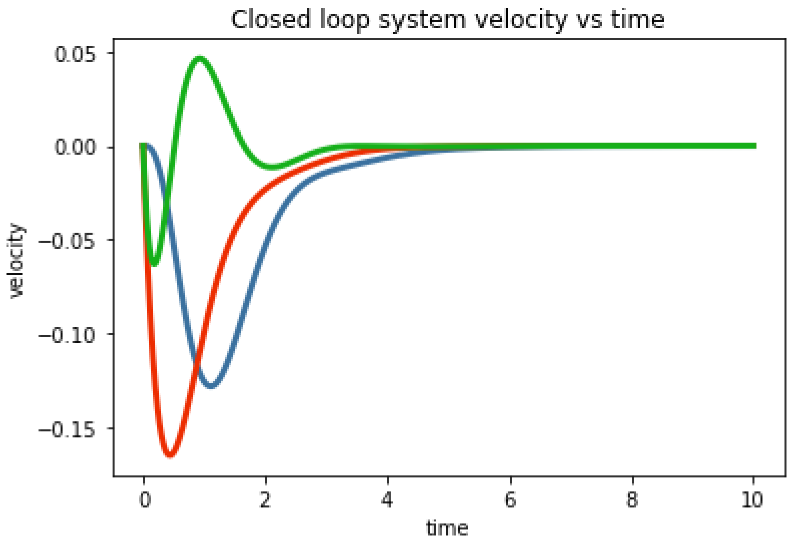

Closed loop system Input vs time
Input
time
Input 1
Input 2
10
8
6
4
2
0
0
2
4
6
8
10

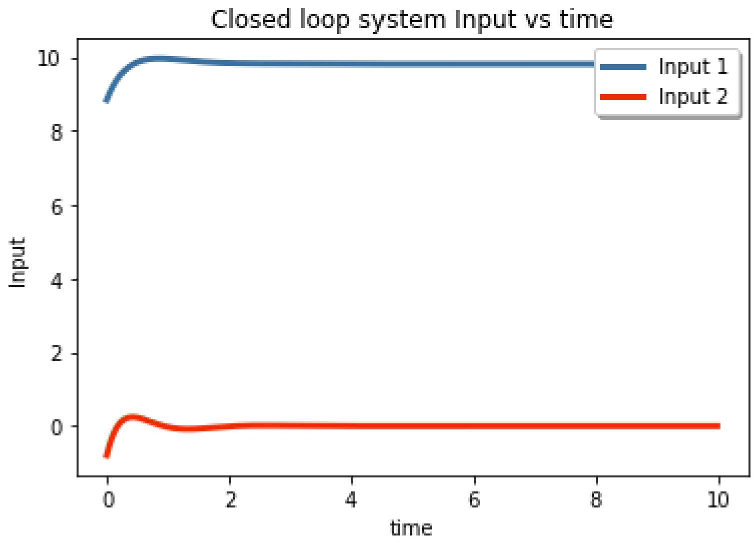

Plot of the states with respect to time for r*Q
Horiz
Verti
Rotat
States
Time
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10

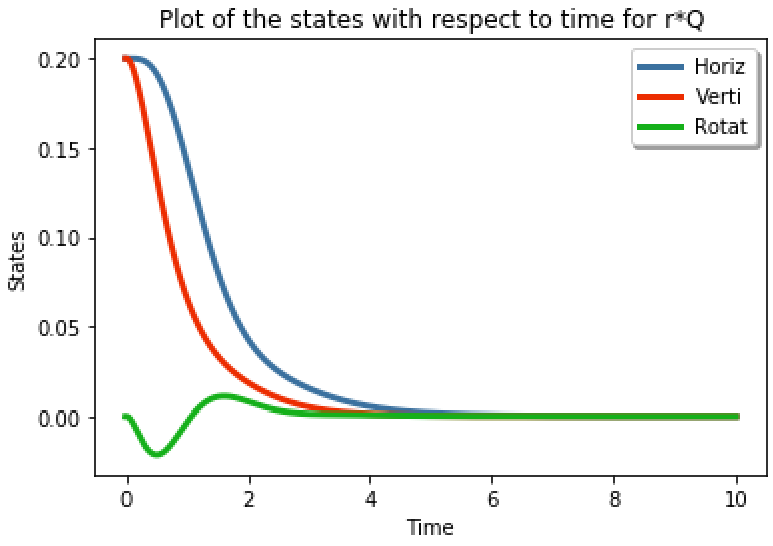

Closed loop system velocity vs time
velocity
time
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100
−0.125
−0.150
0
2
4
6
8
10

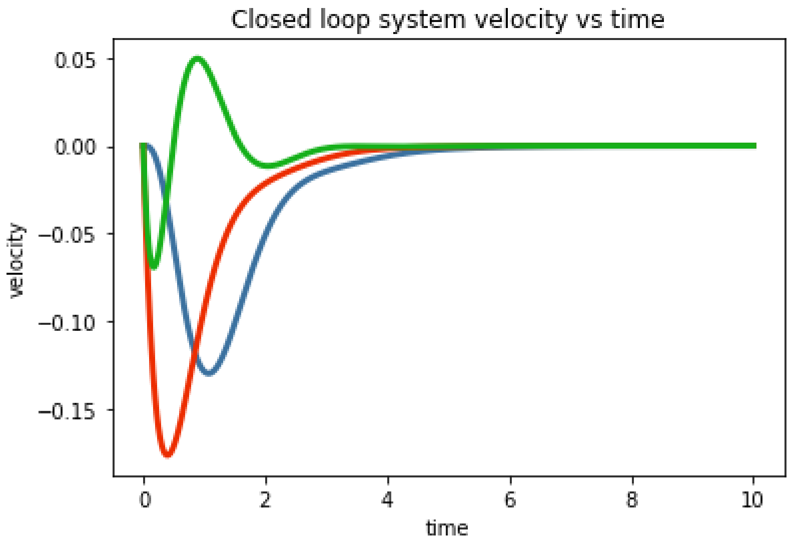

Closed loop system Input vs time
Input 1
Input 2
Input
time
10
8
6
4
2
0
0
2
4
6
8
10

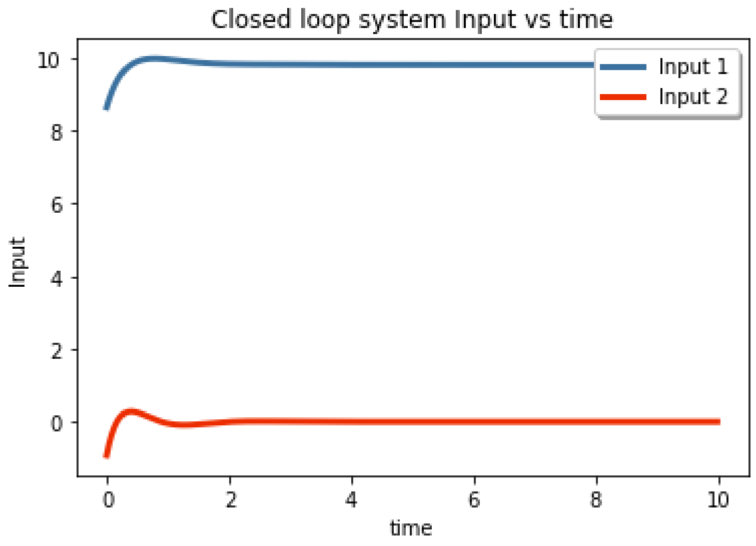

Plot of the states with respect to time for r*Q
Horiz
Verti
Rotat
States
Time
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10


Closed loop system velocity vs time
velocity
time
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10


Closed loop system Input vs time
Input 1
Input 2
Input
time
10
8
6
4
2
0
0
2
4
6
8
10


Plot of the states with respect to time for r*Q
Horiz
Verti
Rotat
States
Time
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10


Closed loop system velocity vs time
velocity
time
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10


Closed loop system Input vs time
Input 1
Input 2
Input
time
10
8
6
4
2
0
0
2
4
6
8
10

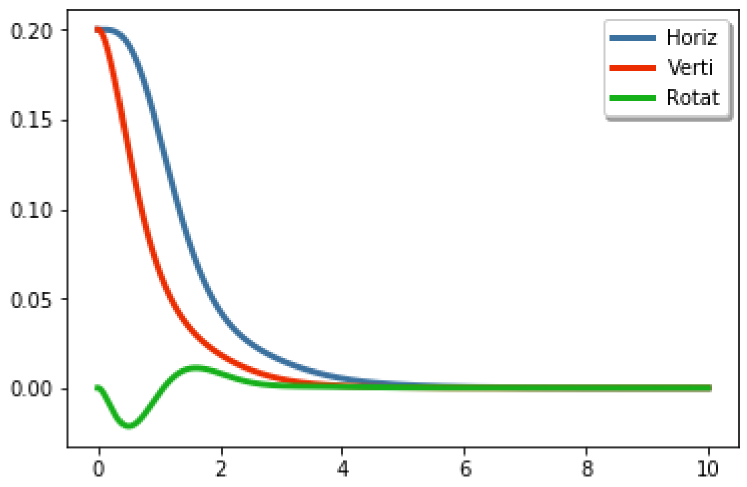
(b) 3

Use that function from (a.1) to randomly generate different Q, R matrices and use the implementation in (a.2) to find an optimal controller for the quadrotor. Implement the optimal control on the nonlinear quadrotor and provide plots of the position, velocity and input as before.

Again, animation will help you visualize what is happening.

Compare the optimal controller performance to the stabilizing feedback performance. What happens as you vary Q and R? Can you generate a Q and R that give you similar performance as the stabilizing feedback controller?

In [12]:
```
def u(t,x):
    dt=0.01
    s=np.dot(-lqrk[int(t/dt),:,:],x)
    return s+[m*g,0]
def simulatex():
    t0,x0,t,dt=0,np.array([0.2,0.2,0,0,0,0]),10,0.01
    T,x= forward_euler(f,t,x0,t0,dt,utx=u)
    cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]
    fig, ax = plt.subplots()
    ax.plot(T,x[:,0],c=cols[0],label='Horiz',linewidth=3)
    ax.plot(T,x[:,1],c=cols[1],label='Verti',linewidth=3)
    ax.plot(T,x[:,2],c=cols[2],label='Rotat',linewidth=3)
    legend = ax.legend(loc='upper right', shadow=True)
    plt.show()
    return 1
# def forwardeuler(t0,t,f,x0,dt,u):
#     steps= np.arange(t0,t,dt)
#     n=x0.shape
#     xout=[]
#     #xout= np.zeros((int(n[0]),int(t/dt)))
#     for i in steps:
#         #print(x0,u)]
#         #print(K,'u',u)
#         u=np.dot(-lqrk[int(i/dt),:,:],x0)+[m*g,0]
#         out= x0 + dt*f(i,x0,u)
#         xout.append(out)
#         x0=out
#         #print(out)
#     return steps, xout
a=simulatex()
# other method
# t0,x0,t,dt,u=0,np.array([0.2,0.2,0,0,0,0]),20,0.01,[0,0]
# tout,xout= forwardeuler(t0,t,f,x0,dt,u)
# cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green']]
# fig, ax = plt.subplots()
# arrayx=np.array(xout)
# ax.plot(tout,arrayx[:,0],c=cols[0],label='Horiz',linewidth=3)
# ax.plot(tout,arrayx[:,1],c=cols[1],label='Verti',linewidth=3)
# ax.plot(tout,arrayx[:,2],c=cols[2],label='Rotat',linewidth=3)
# legend = ax.legend(loc='upper right', shadow=True)
# plt.show()
```



The controller her is faster in response if the value of Q increases moreover if the value of R decresases then the controller is slower in response.

Part 4 KF on Closed Loop Linearized System

In this part you will implement a discrete time KF on the closed loop linearized system. Linearize your dynamics as in part 1. Take a stabilizing controller and create the closed loop linerized system which is an approximation of the non-linear dynamics in a neighborhood of x_0, u_0 around which we linearized.

In each of the steps where you implement the KF on your system, provide commentary on what you observe. What can you say about the performance as you vary the noise?

(a)

Discretize the stable closed loop system to create a DT difference equation

$$x_{t+1} = A_{\text{cl},d} x_t$$

(b) Consider a noisy version of this system

$$\begin{aligned} x_{t+1} &= A_{\text{cl},d} x_t + F_t w_t \\ y_t &= C_d x_t + H_t v_t \end{aligned}$$

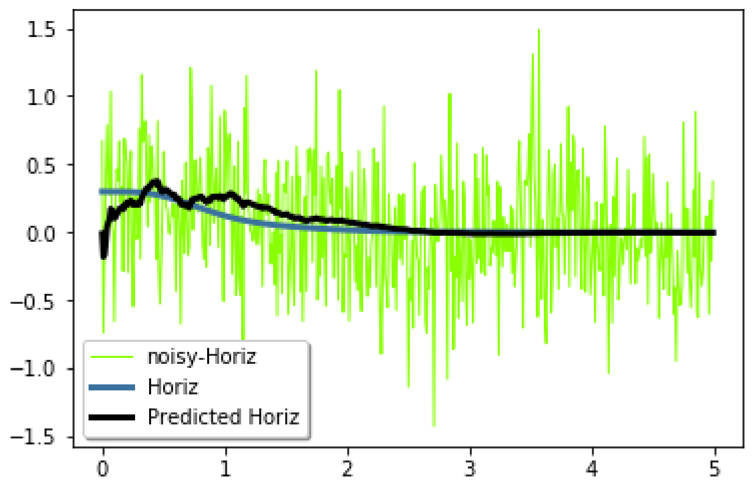
Generate noisy samples of the discrete time system where $\mathbb{E}[w_t w_t^T] = W$, $\mathbb{E}[v_t v_t^T] = V$, $\mathbb{E}[x_0] = \bar{x}_0$ and $\mathbb{E}[(x_0 - \bar{x}_0)(x_0 - \bar{x}_0)^T] = \Sigma_0$. Start with $v_t \sim \mathcal{N}(0, 0.2 I_{2\times 2})$ and $w_t \sim \mathcal{N}(0, 0.1 I_{2\times 2})$, $\Sigma_0 = \text{diag}(0.1, 0.1, 0.1, 0.1, 0.1, 0.1)$. Let F_t be the discretized control dynamics B_d and $H_t = I$.

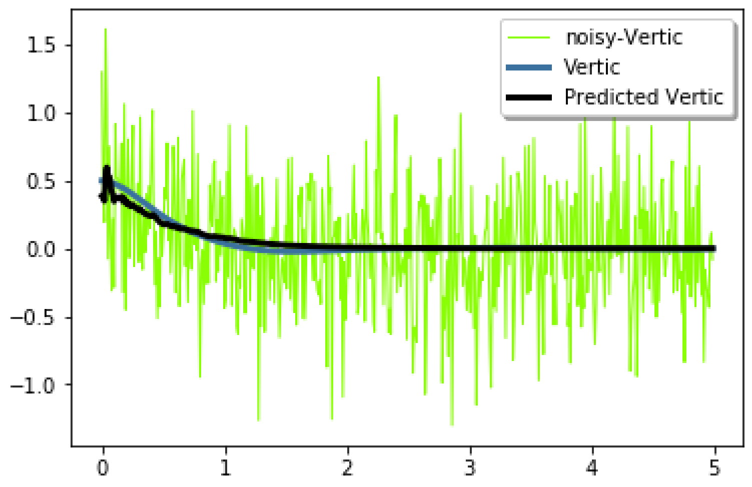
(c)

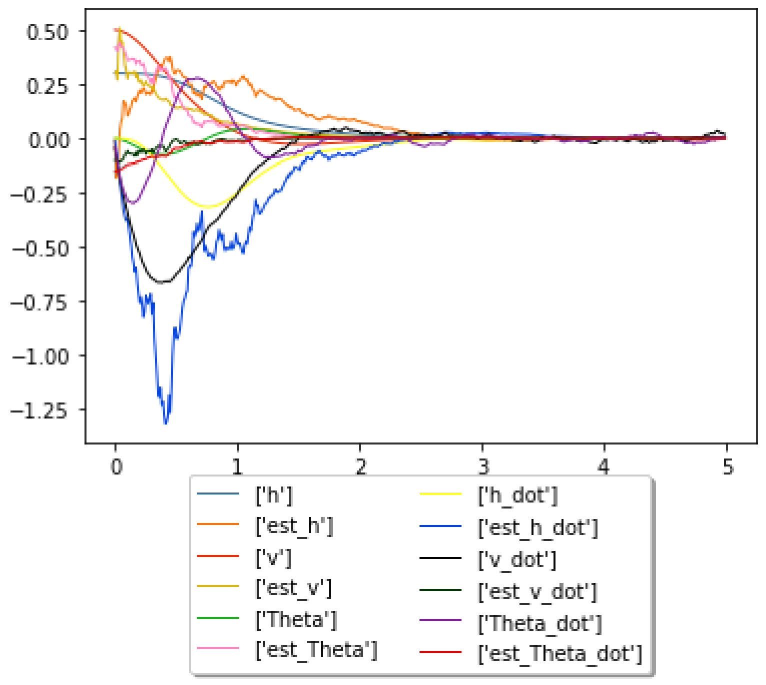
Implement the discrete time Kalman filter. You must do this your self, not using any off-the shelf tools.

In [1]:
```
import sys
sys.modules[__name__].__dict__.clear()
from _547 import forward_euler
import matplotlib.pyplot as plt
import math
import numpy as np
from seaborn import xkcd_rgb as xkcd
from matplotlib import animation, rc
from numpy import linalg as LA
import control
import scipy
import copy
from matplotlib.font_manager import FontProperties



m,g,I=1,9.81,1
A = np.array([[0,0,0,1,0,0],[0,0,0,0,1,0],[0,0,0,0,0,1],[0,0,g,0,0,0],[0
,0,0,0,0,0],[0,0,0,0,0,0]])
B = np.array([[0,0],[0,0],[0,0],[0,0],[1,0],[0,1]])
def f(t,x,u):
    # positions, velocities
    # horiz., vert., rotation
    # thrust, torque
    vector= np.asarray([x[3],x[4],x[5],(u[0]/m)*math.sin(x[2]),(-g+(u[0]
/m)*math.cos(x[2])),u[1]/I])
    return vector# returning the vecotr of the functions given in the qu
estion
def f_lin(t,x,u): #function if linearized dynamics
    A=np.asarray([[0,0,0,1,0,0],[0,0,0,0,1,0],[0,0,0,0,0,1],[0,0,m*g/m,0
,0,0],[0,0,-m*g/m*x[2],0,0,0],[0,0,0,0,0,0]])
    B=np.asarray([[0,0],[0,0],[0,0],[x[2]/m,0],[1/m,0],[0,1/I]])
    return np.dot(A,x)+np.dot(B,u)
def find_k(A,B):
    la=2
    n,m= A.shape
    newA= -la*np.identity(n)-A
    # check if stable

    W= np.array(scipy.linalg.solve_lyapunov(newA,np.dot(B,B.transpose
())))
    #print(W)
    P=LA.inv(W)
    #print(P)
    K= -1/2*np.dot(np.transpose(B),P)
    p,r=K.shape
    Kvalue=np.zeros((p,r))
    for i in range(p):
        for j in range(r):
            Kvalue[i,j]= float(K[i,j])
    return Kvalue,newA

K,newA= find_k(A,B)
t0,t,dt=0,5,0.01
Kss=K
t_=np.arange(t0,t,dt)
```

```
#print(Kss)
NewA=A-np.dot(B,Kss)
Ad = scipy.linalg.expm(newA*dt)
Bd = np.dot(LA.inv(newA),np.dot((Ad-np.identity(Ad.shape[0])),B))
Q,R,P0,H,u1 =0.1*np.identity(2),0.2*np.identity(2),0.1*np.identity(A.sha
pe[0]),np.identity(2),np.array([0,0])
x0=np.random.multivariate_normal(np.array([0,0,0,0,0,0]),P0)
# starting point of filter
def kalman_filter(x0,P0,Adis,Bdis,Ks,Q,H,R,t_,dt,f,u,A,B):

    F,C,x0p =Bdis,np.array([[1,0,0,0,0,0],[0,1,0.2,0,0,0]]),np.array([0.
3,0.5,0,0,0,0]) #startig point of equation
    savey,savex,saveybar,savexbar,say=[],[],[],[],[]
    for i in t_:
        u1=-np.dot(Ks,x0p)#+[m*g,0]
        xt= x0p + dt*(f_lin(i,x0p,u1))+np.dot(F,np.random.multivariate_n
ormal([0,0],Q))# the main equation 1
        yt1= np.dot(C,xt)
        yt= np.dot(C,xt) +np.dot(H,np.random.multivariate_normal([0,0],R
))# the main equauation 2

        barx= np.dot(Adis,x0)
        #print('barx',barx)
        Pbar= np.dot(Adis,np.dot(P0,Adis.transpose())) + np.dot(F,np.dot
(Q,F.transpose()))
        #rint('Pbar',Pbar)
        L1= LA.inv(np.dot(C,np.dot(Pbar,C.transpose()))+np.dot(H,np.dot(
R,H.transpose())))
        L2= np.dot(Adis,np.dot(Pbar,C.transpose()))
        L= np.dot(L2,L1)
        barx1= barx + np.dot(L,(yt-np.dot(C,barx)))
        Pbar1=  Pbar-np.dot(np.dot(L,C),Pbar)
        x0= copy.deepcopy(barx1)
        P0= copy.deepcopy(Pbar1)
        #print(L)
        #print('xt',xt)
        x0p= copy.deepcopy(xt)
        say.append(yt1)
        savey.append(yt)
        savex.append(xt)
        saveybar.append(np.dot(C,barx1))
        savexbar.append(barx1)
    cols=[xkcd['muted blue'],xkcd['tomato red'],xkcd['green'],xkcd['yell
ow'],xkcd['black'],xkcd['purple']]
    est_cols=[xkcd['orange'],xkcd['gold'],xkcd['pink'],xkcd['blue'],xkcd
['dark green'],xkcd['red']]
    lab = [['h'],['v'],['Theta'],['h_dot'],['v_dot'],['Theta_dot']]
    est_lab = [['est_h'],['est_v'],['est_Theta'],['est_h_dot'],['est_v_d
ot'],['est_Theta_dot']]
    fig, ax = plt.subplots()
    fig, ax1 = plt.subplots()
    fig, ax2 = plt.subplots()



    say=np.array(say)
    savex=np.array(savex)
    savey= np.array(savey)
    savexbar= np.array(savexbar)
    saveybar= np.array(saveybar)


    ax.plot(t_,savey[:,0],c=xkcd['lime green'],label='noisy-Horiz',linew
idth=1)
    ax.plot(t_,say[:,0],c=cols[0],label='Horiz',linewidth=3)
    ax.plot(t_,saveybar[:,0],c=xkcd['black'],label='Predicted Horiz',lin
ewidth=3)

    ax1.plot(t_,savey[:,1],c=xkcd['lime green'],label='noisy-Vertic',lin
ewidth=1)
    ax1.plot(t_,say[:,1],c=cols[0],label='Vertic',linewidth=3)
    ax1.plot(t_,saveybar[:,1],c=xkcd['black'],label='Predicted Vertic',l
inewidth=3)

    for i in range(6):
        ax2.plot(t_,savex[:,i],c=cols[i],label=(str(lab[i])),linewidth=1
)
        ax2.plot(t_,savexbar[:,i],c=est_cols[i],label=(str(est_lab[i])),
linewidth=1)
    #ax.plot(tout,arrayx[:,1],c=cols[1],label='Verti',linewidth=3)
    #ax.plot(tout,arrayx[:,2],c=cols[2],label='Rotat',linewidth=3)
    legend = ax.legend(loc='best', shadow=True)
    legend = ax1.legend(loc='best', shadow=True)
    #legend = ax2.legend(loc='best', shadow=True)
    legend = ax2.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
  shadow=True, ncol=2)
    plt.show()


kalman_filter(x0,P0,Ad,Bd,Kss,Q,H,R,t_,dt,f,u1,A,B)
```


noisy-Horiz
Horiz
Predicted Horiz
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
1
2
3
4
5


noisy-Vertic
Vertic
Predicted Vertic
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5

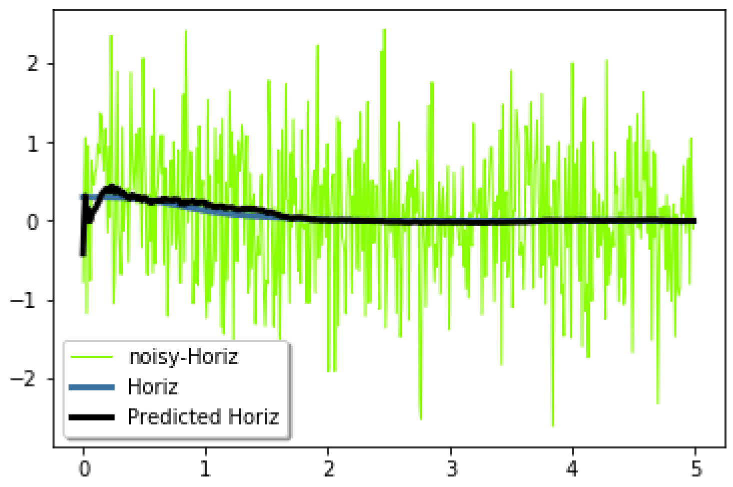

0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
0
1
2
3
4
5
['h']
['est_h']
['v']
['est_v']
['Theta']
['est_Theta']
['h_dot']
['est_h_dot']
['v_dot']
['est_v_dot']
['Theta_dot']
['est_Theta_dot']

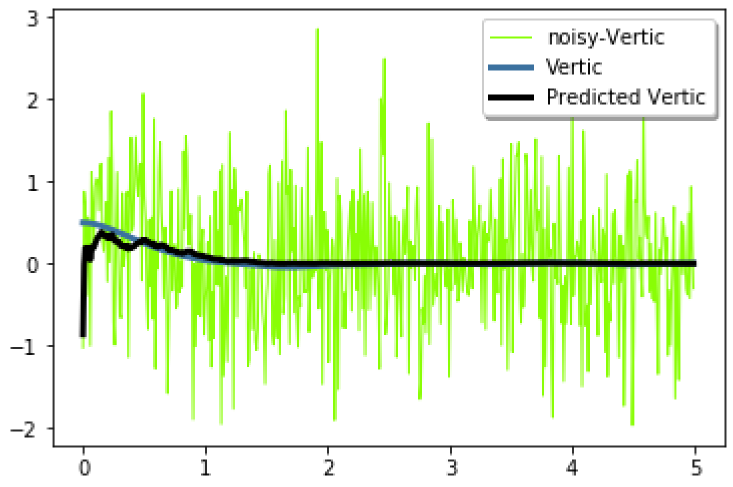
(d) Apply the kalman filter to your system.

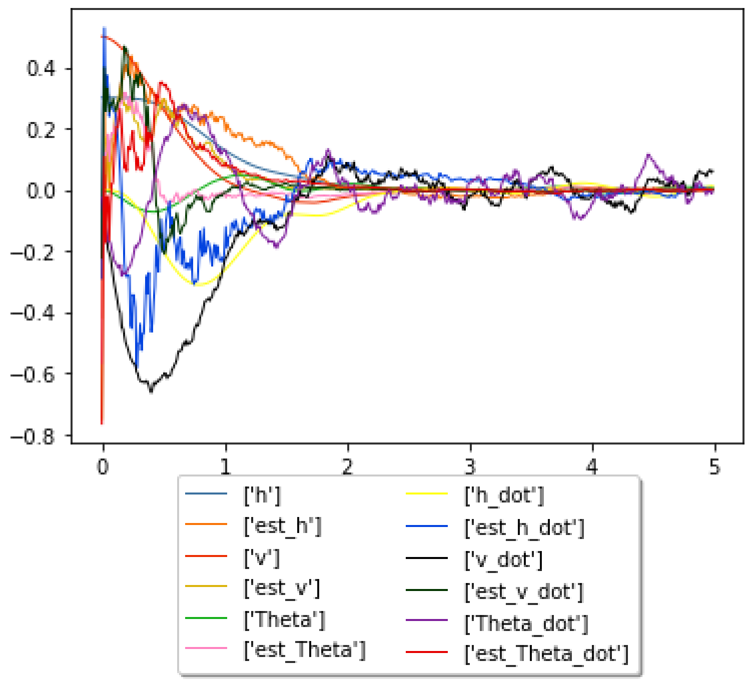
Test first with the above noise statistics. Plot the y's you generated (i.e. $y_t = Cx_t$ where $x_t = A_{\mathrm{cl}}x_{t-1} + F_t w_t$), the noisy y's (i.e. $y_t = Cx_t + H_t v_t$) and the KF estimates of y_t. On another figure, plot also the KF estimated x's and the noisy samples of x.

Now use the PSD function to test other randomly generate Σ_0's and test other distributions for v_t and w_t. Compare the performance.

In [2]:
```
def generate_random_posdef(size):
    temp = scipy.random.rand(size,size)
    B = np.dot(temp,temp.transpose())
    return B
Q,R,P0,H,u1 =generate_random_posdef(2),generate_random_posdef(2),generat
e_random_posdef(A.shape[0]),np.identity(2),np.array([0,0])
x0=np.random.multivariate_normal(np.array([0,0,0,0,0,0]),P0)
kalman_filter(x0,P0,Ad,Bd,Kss,Q,H,R,t_,dt,f,u1,A,B)
```


noisy-Horiz
Horiz
Predicted Horiz
2
1
0
−1
−2
0
1
2
3
4
5


noisy-Vertic
Vertic
Predicted Vertic
3
2
1
0
−1
−2
0
1
2
3
4
5


0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
1
2
3
4
5
['h']
['est_h']
['v']
['est_v']
['Theta']
['est_Theta']
['h_dot']
['est_h_dot']
['v_dot']
['est_v_dot']
['Theta_dot']
['est_Theta_dot']