Homework 5

Variant: A
Email: d.manakovskiy@innopolis.university

The given dynamics:

$$(M+m)\ddot{x} - ml\ddot{\theta}\cos(\theta) + ml\dot{\theta}^2\sin(\theta) = F \tag{1}$$

$$-\ddot{x}\cos(\theta) + l\ddot{\theta} - g\sin(\theta) = 0 \tag{2}$$

where $M = 7.5, m = 4.4, l = 1.2, g = 9.81$
The system can be written in a state space form:

$$\begin{cases} \dot{z} = f(z) + g(z)u \\ y = h(z) = \begin{bmatrix} x & \theta \end{bmatrix}^{\mathbf{T}} \end{cases}$$

where $z = \begin{bmatrix} x & \theta & \dot{x} & \dot{\theta} \end{bmatrix}^{\mathbf{T}}$
The system can also be linearized around $\bar{z} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^{\mathbf{T}}$ as was done in Homework4:

$$\begin{cases} \delta\dot{z} = A\delta z + B\delta u \\ \delta y = C\delta z \end{cases}$$

$$\begin{cases} \delta\dot{z} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{gm}{M} & 0 & 0 \\ 0 & \frac{g(m+M)}{Ml} & 0 & 0 \end{bmatrix} \delta z + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{M} \\ \frac{1}{Ml} \end{bmatrix} \delta u \\ \delta y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \delta z \end{cases}$$

$$\begin{cases} \delta\dot{z} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 5.7552 & 0 & 0 \\ 0 & 12.971 & 0 & 0 \end{bmatrix} \delta z + \begin{bmatrix} 0 \\ 0 \\ 0.133 \\ 0.111 \end{bmatrix} \delta u \\ \delta y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \delta z \end{cases}$$

Task A. Determining observability.

To verify that it is possible to design an observer for linearized system, we have to construct observability matrix $O^{np \times n}$, where $n = \dim(x) = 4, p = \dim(y) = 2$:

$$O = \begin{bmatrix} C \\ CA \\ CA^2 \\ CA^3 \end{bmatrix}$$

if $rank(O) = n = 4$, then it is possible to design a controller. Code for finding O is present at listing 1.

Listing 1: Finding observability matrix O

```
import numpy as np
from numpy.linalg import matrix_power, matrix_rank

A = np.array([
                [0,0,      1,0],
                [0,0,      0,1],
                [0,5.7552, 0,0],
                [0,12.971, 0,0],
])

B = np.array([
                [0],
                [0],
                [0.133],
                [0.111],
])

C = np.array([
               [1, 0, 0, 0],
               [0, 1, 0, 0],
])

# finding observability matrix
O = np.concatenate(
    (
        C,
        C.dot(A),
        C.dot(matrix_power(A, 2)),
        C.dot(matrix_power(A, 3)),
     )
)
```

Let us see the matrix itself and its rank:

```
>>> print(f"Observability matrix:\n{O}\n\nShape:{O.shape}\nrank: {matrix_rank(O)}")

Observability matrix:
[[ 1.      0.      0.      0.    ]
 [ 0.      1.      0.      0.    ]
 [ 0.      0.      1.      0.    ]
 [ 0.      0.      0.      1.    ]
 [ 0.      5.7552  0.      0.    ]
 [ 0.     12.971   0.      0.    ]
 [ 0.      0.      0.      5.7552]
 [ 0.      0.      0.     12.971 ]]

Shape:(8, 4)
rank: 4
```

As we see, the observability matrix has dimensions 8×4 and rank 4, so it is possible to design an observer.

Task B. Exploring openloop observer.

Openloop observer is just a simulation of a system dynamics. In case we know initial conditions, we will get 100% observation, otherwise nothing is guaranteed. Listing 2 considers system stabilized using LQR, to observe unstable system, replace line 39 to

```
return np.array([1])
```

For the code I decided to change x and θ in initial conditions since in real life we almost never know them.

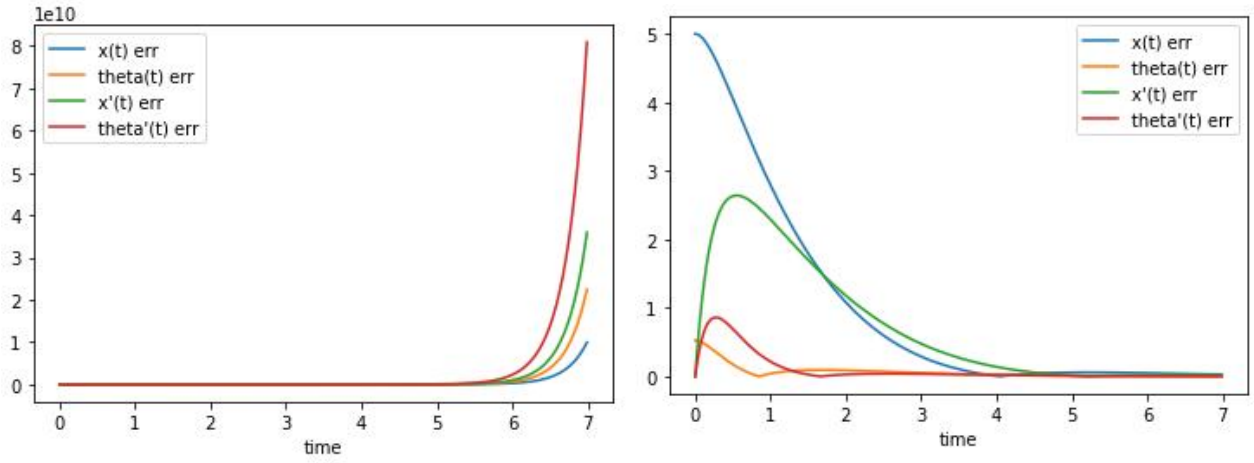
Listing 2: Open-loop observation error for stable system

```
1  import matplotlib.pyplot as plt
2  from scipy.integrate import odeint
3  from scipy.linalg import inv, solve_continuous_are
4  from math import pi
5
6  def lqr(A, B, Q, R):
7    dx/dt = Ax+ BuJ = integral of x.T * Q * x + u.T *R * u
8    # solve Algebraic Riccati equation
9    S = solve_continuous_are(A, B, Q, R)
10
11   # Compute gain
```

```
  K = inv(R) * B.T.dot(S)

  return np.asarray(K)

time_start = 0
time_finish = 7
time_step = 0.01

x_0 = 100
ang_0 = 0
diff_x_0 = 0
diff_ang_0 = 100

labels = [x(t), theta(t), x'(t), theta'(t)]

time = np.arange(time_start, time_finish, time_step)
init = [x_0, ang_0, diff_x_0, diff_ang_0]
init_obs = [x_0 + 5, ang_0 + pi/6, diff_x_0, diff_ang_0]

Q = np.eye(4) * 100
R = np.eye(1)
k = lqr(A, B, Q, R)

def u(x, t):
  return -k.dot(x)

def differential(x, t):
  Ax = A.dot(x)
  Bu = B.dot(u(x, t))
  ret = Ax + Bu
  return ret


real = odeint(differential, init, time)
observed = odeint(differential, init_obs, time)
error =  np.absolute(real - observed)

plt.plot(time, error)
plt.xlabel('time')
plt.legend(list(map(lambda x: x +  err, labels)))
plt.show()
```

Figure 1 compares error dynamics on unstable and stabilized system. As we can see at figure 1a an error dynamics in unstable, since initial conditions differ and the system itself is unstable. Figure 1b shows stable error dynamics, because LQR eventually converges on

every initial conditions, so at some moment we will see both: real state and estimation to be equal to 0, which is not really interesting to us.



(a) Unstable system

(b) Stabilized system

Figure 1: Open-loop observation error

Task C. Designing Luenberger observer.

According to MathWorks Luenberger observer has the following form:

$$\hat{x}(t+1) = A\hat{x}(t) + Bu(t) + L(y(t) - \hat{y}(t)$$

where $A - LC \prec 0$

LQR solution

We can find matrix L by solving dual problem using LQR:

$$A - LC \prec 0$$

$$(A - LC)^{\mathbf{T}} \prec 0$$

$$A^{\mathbf{T}} - C^{\mathbf{T}} L^{\mathbf{T}} \prec 0$$

$$L^{\mathbf{T}} = \text{lqr}(A^{\mathbf{T}}, C^{\mathbf{T}}, Q, R)$$

$$L = \text{lqr}(A^{\mathbf{T}}, C^{\mathbf{T}}, Q, R)^{\mathbf{T}}$$

Code equivalent is at listing 3.

Listing 3: Calculating matrix L for Luenberg observer via LQR

```
# LQR
Q =   np.array([
                    [1,0,0,0],
                    [0,1,0,0],
                    [0,0,1,0],
                    [0,0,0,1]
])

R = np.array([
               [10, 0],
               [0, 10],
])

L_lqr = lqr(A.T, C.T, Q, R).T
```

Pole Placement

Having derived dual problem:

$$A^{\mathbf{T}} - C^{\mathbf{T}} L^{\mathbf{T}} \prec 0$$

We can use pole placement to obtain L:

$$L^{\mathbf{T}} = place\_poles(A^{\mathbf{T}}, C^{\mathbf{T}}, desired\_poles).gain\_matrix$$

$$L = place\_poles(A^{\mathbf{T}}, C^{\mathbf{T}}, desired\_poles).gain\_matrix^{\mathbf{T}}$$

Code equivalent is at listing 4.

Listing 4: Calculating matrix L for Luenberg observer via LQR

```
# Pole placement
desired_poles = [−1, −2, −3, −4]
pole = place_poles(A.T, C.T, desired_poles)
L_pole = pole.gain_matrix.T
```

Implementation

In order not to implement custom ODE solver, I decided to use a trick proposed by students of 4th group: to use *odeint* on local timeline comprised of 2 neighbour timestamps.

Brief explanation of a code:

- Function **real** is responsible to calculate real value of x for given time t and input u.

- **luenberg** function calculates Luenbeg observation at time t with input u and sensor measurement y.
- Then we setup initial values. Initial values for real system and observer are different since observer almost never know real initial ones.
- finally, in a loop we calculate observation and real values step-by-step.

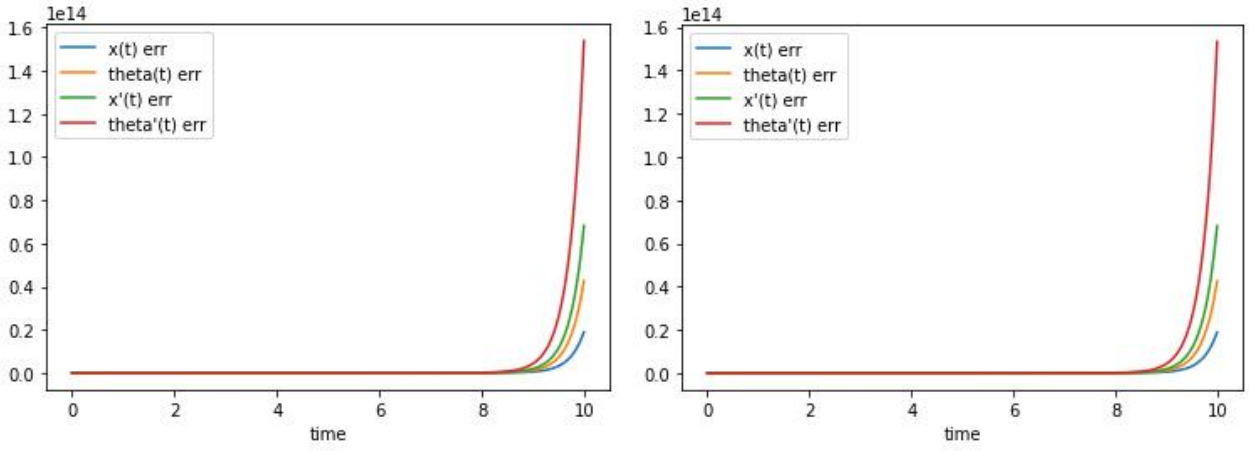
Source code is at listing 5.

Listing 5: Calculating Luenberg observer

```
L = L_lqr  # Change this var to work with calculations of L matrix
def real(x, t, u):
  Ax = A.dot(x)
  Bu = B.dot(u).reshape((4, ))
  ret = Ax + Bu

  return ret

def luenberg(x_est, t, u, y):
  y_diff = y - C.dot(x_est).reshape((2, ))

  Ax = A.dot(x_est)
  Bu = B.dot(u).reshape((4, ))
  Ly_diff = L.dot(y_diff)

  ret = Ax + Bu + Ly_diff
  return ret

time_start = 0
time_finish = 10
time_step = 0.01

x_0 = 0.5
ang_0 = pi/3
diff_x_0 = 0.01
diff_ang_0 = pi/12

x_0_est = 0.5
ang_0_est = pi/6
diff_x_0_est = 0.07
diff_ang_0_est = pi/8

labels = [x(t), theta(t), x'(t), theta'(t)]

```

```
time = np.arange(time_start, time_finish, time_step)
x = [np.array([x_0, ang_0, diff_x_0, diff_ang_0])]
x_obs = [np.array([x_0_est, ang_0_est, diff_x_0_est, diff_ang_0_est])]

k = place_poles(A, B, desired_poles).gain_matrix

for i in range(1, len(time)):
  # This trick was suggested by students of 4th group in order not to
  # implement custom numeric solver but at the same time pass to the observer
  # values from sensors
  local_time = np.linspace(time[i-1], time[i])
  u = -k.dot(x_obs[-1])
  # u = np.array([0])  # uncomment to work with uncontrolled system

  x_dot = odeint(real, x[-1], local_time, args=(u, ))
  x.append(x_dot[-1])

  y = C.dot(x[-1])
  x_obs_dot = odeint(luenberg, x_obs[-1], local_time, args=(u, y))
  x_obs.append(x_obs_dot[-1])

x = np.array(x)
x_obs = np.array(x_obs)
error = np.absolute(x - x_obs)
plt.plot(time, error)
plt.xlabel('time')

plt.legend(list(map(lambda x: x +  err, labels)))
plt.show()
```

As a result on uncotrolled system we see observation error going to infinity (figure 2). The reason for it is that the system is unstable, therefore goes to infinity, so even small approximation error also goes to infinity. The result will be better when we stabilize the system.



(a) Pole placement (b) LQR

Figure 2: Luenberg observer on uncontrolled system

Task D. Designing feedback controller.

This task was done at HomeWork4.

Stabilization is present in code at

Listing 6: Stabilizer

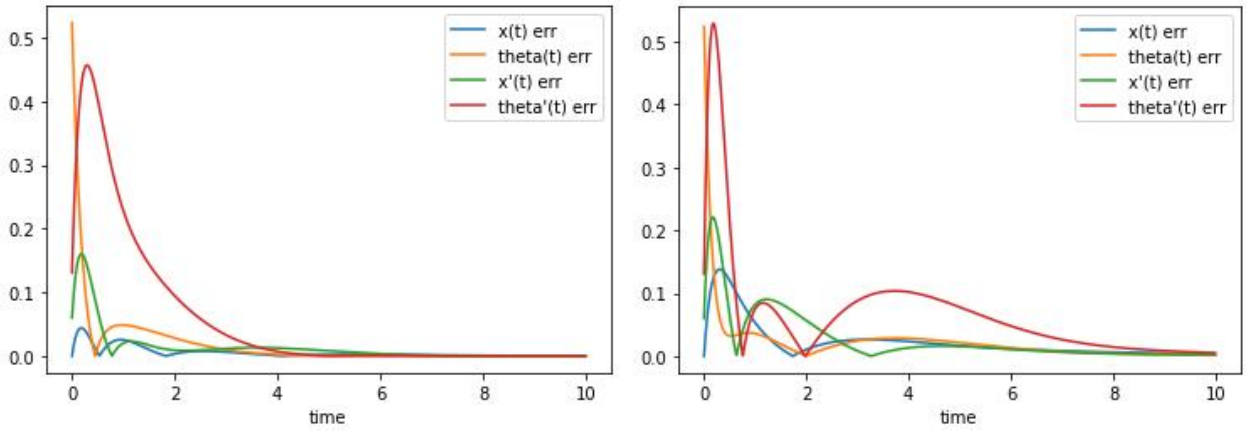
```
desired_poles = [-1, -2, -3, -4]
k = place_poles(A, B, desired_poles).gain_matrix
for i in range(1, len(time)):
  u = -k.dot(x_obs[-1])
  # u = np.array([0])  # uncomment to work with uncontrolled system
```

Task E. Simulate system with Luenberg observer and controller.

The source code is the same as shown at listing 5. The results are present at figure 3. As we can see error converges to 0 relatively fast. Convergence at LQR is not that smooth because R matrix dominates over Q.

Task F. Add white noise to the output.

After adding noise generated by *numpy.random.randn* and multiplying it by scale constant 0.5 we obtain the result presented at figure 4. Observer output becomes noise thus observation error also changes rapidly.



(a) Pole placement (b) LQR

Figure 3: Luenberg observer on controlled system

Task G. Add white noise to the dynamics.

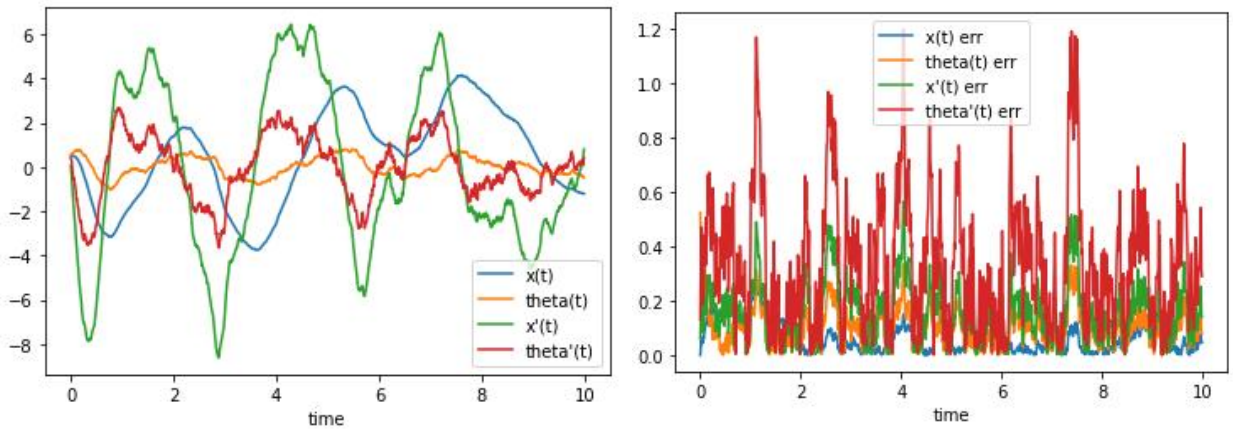
As you see, previously I have widely used odeint as DE solb a DE solver, but I experienced problems with it, because it uses 'Adams/BDF method with automatic stiffness detection and switching.' according to docs.scipy.org. I do not why, but it causes odeint to work very slow on noise data and produce some internal errors. After this I decided to switch to Runge-Kutta solver. The code is available under 'Kalman filter' section of the source code. As a result at figure 5 we can see noisy states.

Task H. Implement Kalman Filter.

Theory on Kalman filter is taken from Wikipedia and Lab 11. The source code is shown at listing 7

Listing 7: Kalman Filter implementation.

```
class KalmanFilter():
  def __init__(self, A, C, Q, R, B = None, P = None, x_init = None):

    self.n = A.shape[1]

    self.A = A
    self.B = 0 if B is None else B
    self.C = C
    self.Q = Q
    self.R = R
    self.P = np.eye(self.n) if P is None else P
    self.x = np.zeros((self.n, )) if x_init is None else x_init
```



(a) Observer output

(b) Observation error

Figure 4: System behaviour with output noise scaled by 0.5

```
13
14  def predict(self, u = np.array([0, ])):
15    self.x = np.dot(self.A, self.x) + np.dot(self.B, u)
16    self.P = np.dot(np.dot(self.A, self.P), self.A.T) + self.Q
17
18  def update(self, y_sensor):
19    y_diff = y_sensor - np.dot(self.C, self.x)
20    K_denum = self.R + np.dot(self.C, np.dot(self.P, self.C.T))
21    K = np.dot(np.dot(self.P, self.C.T), inv(K_denum))
22    self.x = self.x + np.dot(K, y_diff)
23    I = np.eye(self.n)
24    self.P = np.dot(
25        np.dot(I - np.dot(K, C), self.P),
26        (I - np.dot(K, self.C)).T
27    ) + np.dot(np.dot(K, self.R), K.T)
28    return self.x
```

Task I. Kalman Filter correctness.

Initially I had an error in the usage of the filter. Having fixed that I understood that now I have another problem: if I pass u to the KF predict function, the output becomes numerically unstable. Although if time_step is small enough, result becomes much better (see figure 6). Kalman predictions on uncontrolled system. is similar to the situation at Luenberg observer on uncontrolled system: small observation error tends to infinity when the system is unstable. As well error on stabilized system eventually converges to 0 (fig. 8).

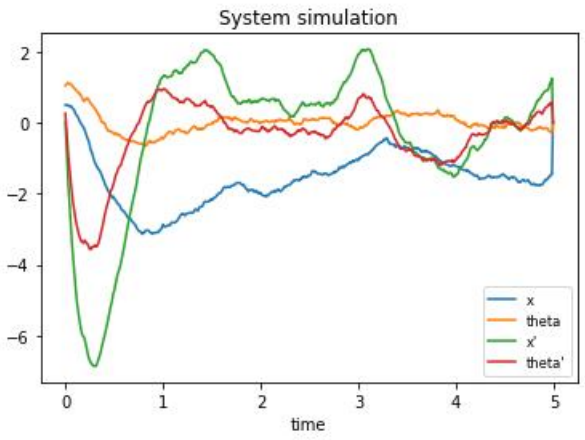


Figure 5: Noisy dynamics.

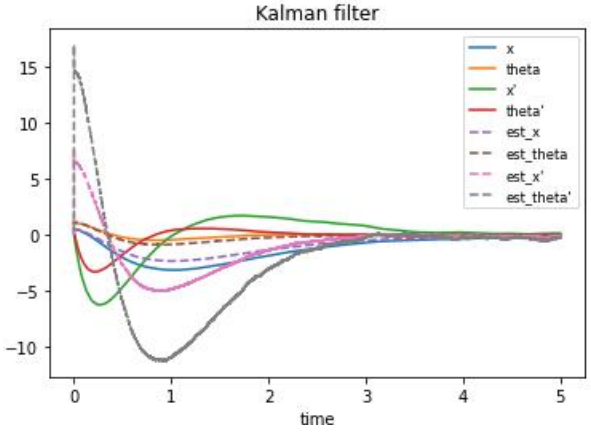


Figure 6: Kalman filter with 0 input

Task J. LQG controller.

LQG is a combination of a LQR and LQE (Kalman filter). Since both LQR and LQE - the best controller and estimator, their combination also guarantees stability, but there is a catch: sometimes controller designed using state-space tools are sensitive to errors in the knowledge of the system dynamics. My controller diverges, I think this happens due to relatively big errors at the beginning of observation (fig 8) and the controller does not recover from such mistakes. It seems to me, that the problem is in the description of matrices Q and R for Kalman Filter (matrices stating for noise covariances).

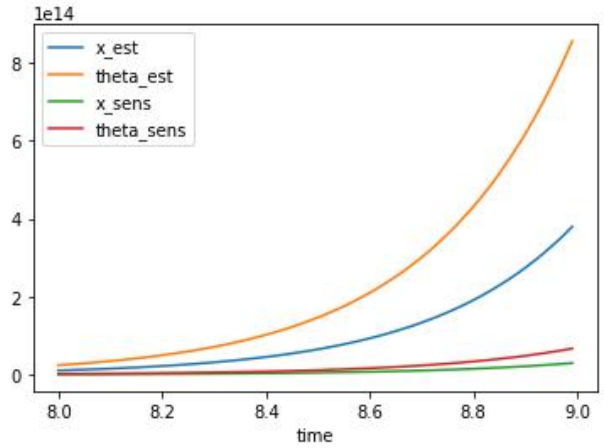


Figure 7: Kalman predictions on uncontrolled system.

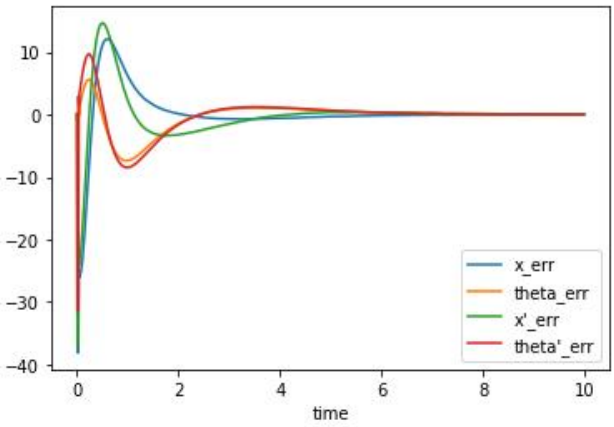


Figure 8: Kalman prediction error on controlled system.