

Variant: A

Email: d.manakovskiy@innopolis.university

The given dynamics:

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

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

where $M = 7.5, m = 4.4, l = 1.2, g = 9.81$

The system can be written in a state space form:

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

where $z = \begin{bmatrix} x & \theta & \dot{x} & \dot{\theta} \end{bmatrix}^T$

The system can also be linearized around $\bar{z} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^T$ 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 $\text{rank}(O) = n = 4$, then it is possible to design a controller. Code for finding O is present at listing 1.

Listing 1: Finding observability matrix O

```

1 import numpy as np
2 from numpy.linalg import matrix_power, matrix_rank
3
4 A = np.array([
5     [0,0,    1,0],
6     [0,0,    0,1],
7     [0,5.7552, 0,0],
8     [0,12.971, 0,0],
9 ])
10
11 B = np.array([
12     [0],
13     [0],
14     [0.133],
15     [0.111],
16 ])
17
18 C = np.array([
19     [1, 0, 0, 0],
20     [0, 1, 0, 0],
21 ])
22
23 # finding observability matrix
24 O = np.concatenate(
25     (
26         C,
27         C.dot(A),
28         C.dot(matrix_power(A, 2)),
29         C.dot(matrix_power(A, 3)),
30     )
31 )

```

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
```

```

12 K = inv(R) * B.T.dot(S)
13
14 return np.asarray(K)
15
16 time_start = 0
17 time_finish = 7
18 time_step = 0.01
19
20 x_0 = 100
21 ang_0 = 0
22 diff_x_0 = 0
23 diff_ang_0 = 100
24
25 labels = [x(t), theta(t), x'(t), theta'(t)]
26
27 time = np.arange(time_start, time_finish, time_step)
28 init = [x_0, ang_0, diff_x_0, diff_ang_0]
29 init_obs = [x_0 + 5, ang_0 + pi/6, diff_x_0, diff_ang_0]
30
31 Q = np.eye(4) * 100
32 R = np.eye(1)
33 k = lqr(A, B, Q, R)
34
35 def u(x, t):
36     return -k.dot(x)
37
38 def differential(x, t):
39     Ax = A.dot(x)
40     Bu = B.dot(u(x, t))
41     ret = Ax + Bu
42     return ret
43
44
45 real = odeint(differential, init, time)
46 observed = odeint(differential, init_obs, time)
47 error = np.absolute(real - observed)
48
49 plt.plot(time, error)
50 plt.xlabel('time')
51 plt.legend(list(map(lambda x: x + 'err', labels)))
52 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.

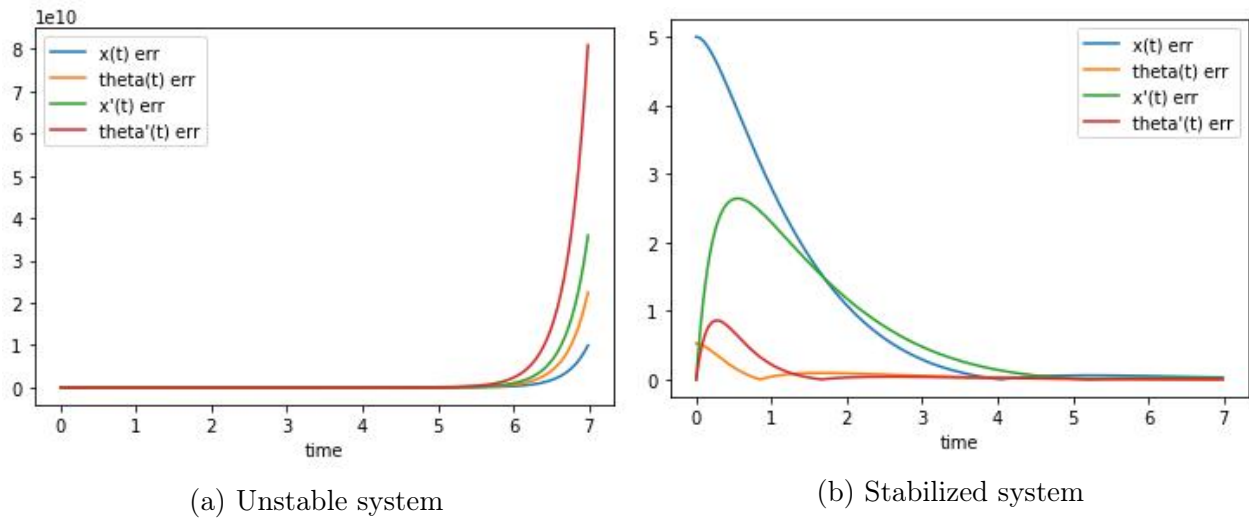


Figure 1: Open-loop observation error

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:

$$A - LC \prec 0$$

$$(A - LC)^T \prec 0$$

$$A^T - C^T L^T \prec 0$$

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

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

Code equivalent is at listing [3](#).

Listing 3: Calculating matrix L for Luenberg observer via LQR

```

1 # LQR
2 Q = np.array([
3         [1,0,0,0],
4         [0,1,0,0],
5         [0,0,1,0],
6         [0,0,0,1]
7     ])
8
9 R = np.array([
10         [10, 0],
11         [0, 10],
12     ])
13
14 L_lqr = lqr(A.T, C.T, Q, R).T

```

Pole Placement

Having derived dual problem:

$$A^T - C^T L^T \prec 0$$

We can use pole placement to obtain L:

$$L^T = \text{place_poles}(A^T, C^T, \text{desired_poles}).\text{gain_matrix}$$

$$L = \text{place_poles}(A^T, C^T, \text{desired_poles}).\text{gain_matrix}^T$$

Code equivalent is at listing 4.

Listing 4: Calculating matrix L for Luenberg observer via LQR

```

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

```
1 L = L_lqr # Change this var to work with calculations of L matrix
2 def system (x, t):
3     ret = (A - B.dot(k)).dot(x)
4     if NOISE_DYNAMICS:
5         ret += NOISE_SCALE_DYNAMICS * np.random.randn(*(ret.shape))
6
7     return ret
8
9 def luenberg(x_est, t, u, y):
10     y_diff = y - C.dot(x_est).reshape((2, ))
11
12     Ax = A.dot(x_est)
13     Bu = B.dot(u).reshape((4, ))
14     Ly_diff = L.dot(y_diff)
15
16     ret = Ax + Bu + Ly_diff
17     return ret
18
19 time_start = 0
20 time_finish = 10
21 time_step = 0.01
22
23 x_0 = 0.5
24 ang_0 = pi/3
25 diff_x_0 = 0.01
26 diff_ang_0 = pi/12
27
28 x_0_est = 0.5
29 ang_0_est = pi/6
30 diff_x_0_est = 0.07
31 diff_ang_0_est = pi/8
32
33 labels = [x(t), theta(t), x'(t), theta'(t)]
34
```

```

35 time = np.arange(time_start, time_finish, time_step)
36 x = [np.array([x_0, ang_0, diff_x_0, diff_ang_0])]
37 x_obs = [np.array([x_0_est, ang_0_est, diff_x_0_est, diff_ang_0_est])]
38
39 k = place_poles(A, B, desired_poles).gain_matrix
40
41 for real_x in np.array(list(RungeKutta(system, x[0], time))):
42     x.append(real_x)
43     # This trick with odeint was suggested by students of 4th group in order not to
44     # implement custom numeric solver but at the same time pass to the observer
45     # values from sensors
46     local_time = np.linspace(time[i-1], time[i], num=2)
47
48     y = C.dot(x[-1])
49     if NOISE_OUTPUT:
50         y += NOISE_SCALE_OUTPUT * np.random.randn(*(y.shape))
51
52     sensor_data.append(y)
53
54     u = -k.dot(x_obs[-1])
55     # u = np.array([0]) # uncomment to work with uncontrolled
56     x_obs_dot = odeint(luenberg, x_obs[-1], local_time, args=(u, y))
57     x_obs.append(x_obs_dot[-1])
58
59 x = np.array(x)
60 x_obs = np.array(x_obs)
61 error = np.absolute(x - x_obs)
62 plt.plot(time, error)
63 plt.xlabel('time')

```

As a result on uncontrolled 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.

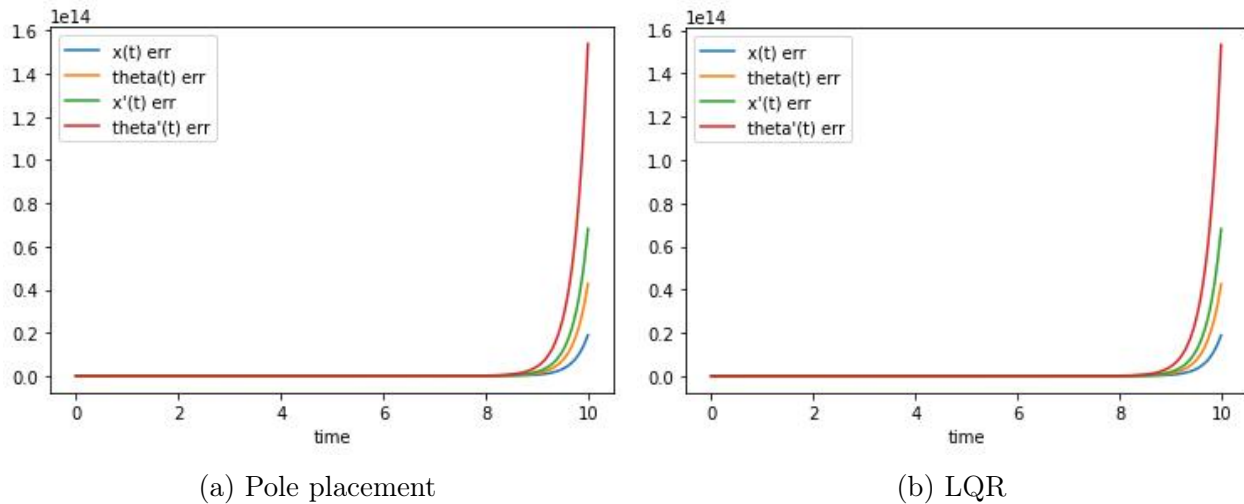


Figure 2: Luenberg observer on uncontrolled system

Task D. Designing feedback controller.

This task was done at [HomeWork4](#).

Stabilization is present in code at

Listing 6: Stabilizer

```
1 desired_poles = [-1, -2, -3, -4]
2 k = place_poles(A, B, desired_poles).gain_matrix
3 for real_x in np.array(list(RungeKutta(system, x[0], time))):
4     local_time = np.linspace(time[i-1], time[i], num=2)
```

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. At such extreme conditions observer output becomes noise thus observation error also changes extremely rapidly. Making noise more realistic results in less changes (see 5).

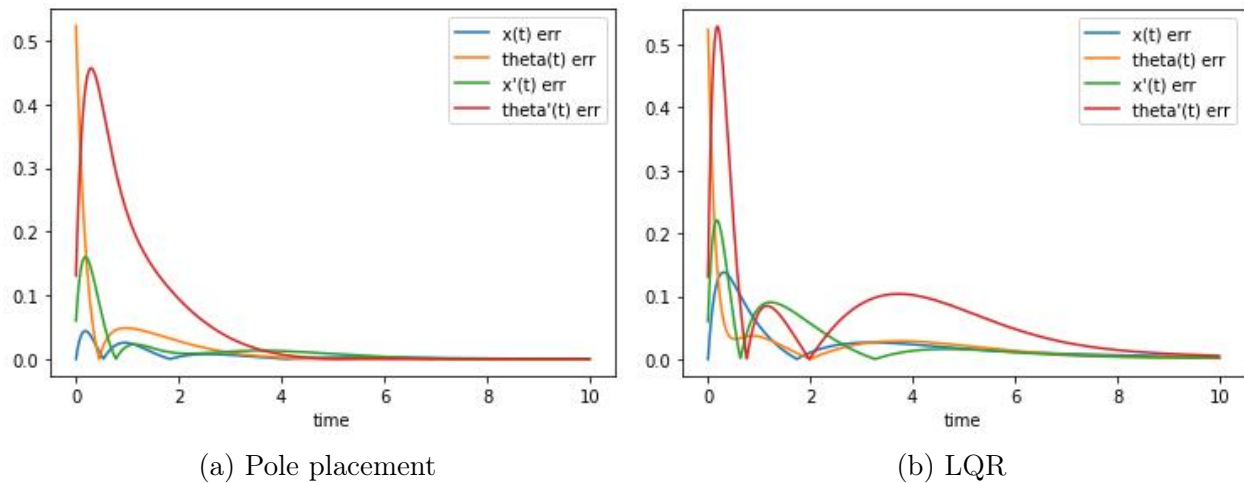


Figure 3: Luenberg observer on controlled system

Task G. Add white noise to the dynamics.

As you see, previously I have widely used `odeint` as 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](https://docs.scipy.org/doc/scipy/reference/ode.html). 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 6 we can see noisy states from the system simulation. The Luenberg observer diverges if dynamic contains a noise (fig. 7).

Task H. Implement Kalman Filter.

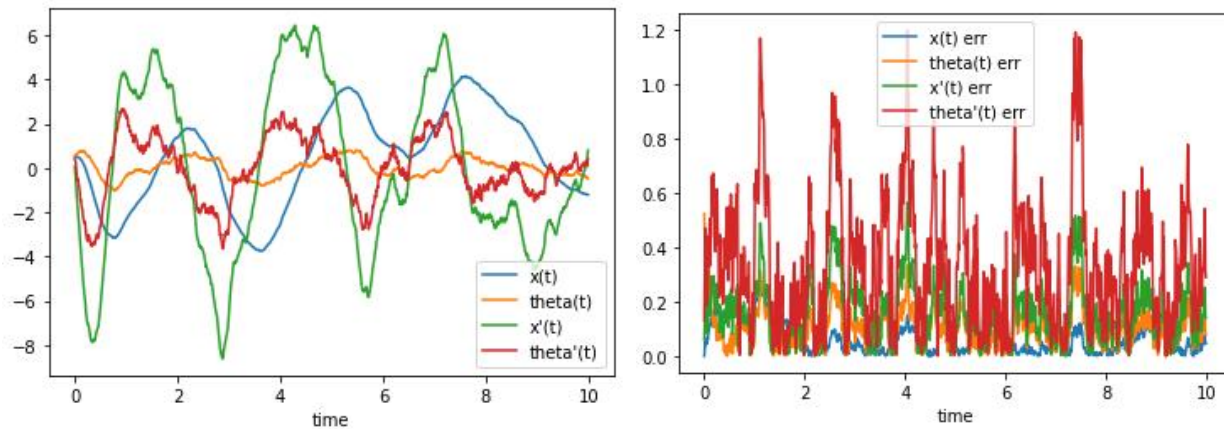
Theory on Kalman filter is taken from [Wikipedia](https://en.wikipedia.org/wiki/Kalman_filter) and Lab 11. The source code is shown at listing 7

Listing 7: Kalman Filter implementation.

```

1 class KalmanFilter():
2     def __init__(self, A, C, Q, R, B = None, P = None, x_init = None):
3
4         self.n = A.shape[1]
5
6         self.A = A
7         self.B = 0 if B is None else B
8         self.C = C
9         self.Q = Q
10        self.R = R

```



(a) Observer output

(b) Observation error

Figure 4: System behaviour with output noise scaled by 0.5

```

11 self.P = np.eye(self.n) if P is None else P
12 self.x = np.zeros((self.n, )) if x_init is None else x_init
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 8). [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.

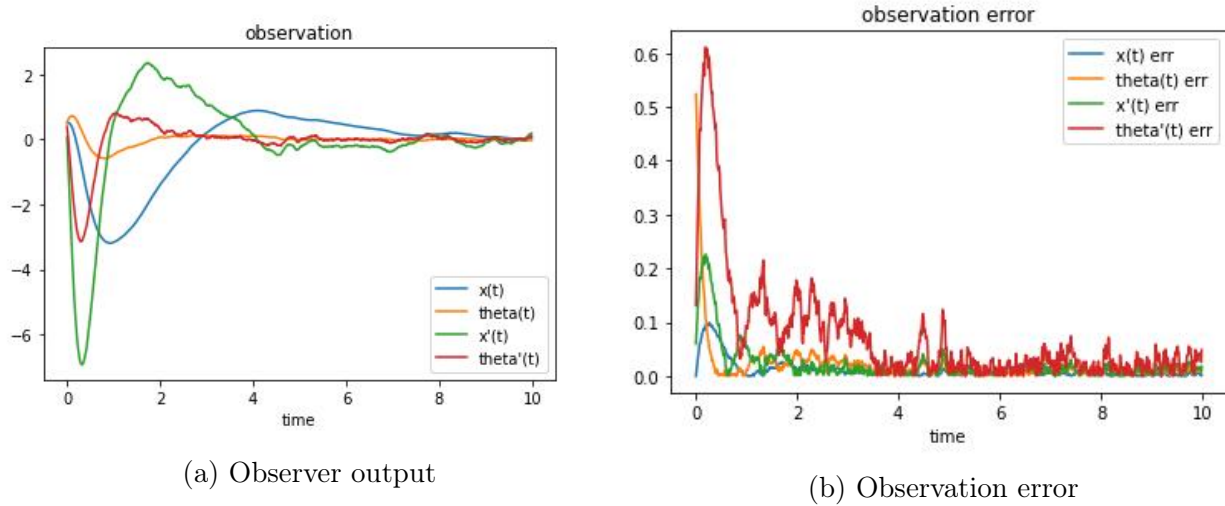


Figure 5: System behaviour with output noise scaled by 0.05

As well error on stabilized system eventually converges to 0 (fig. 10).

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 10) 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).

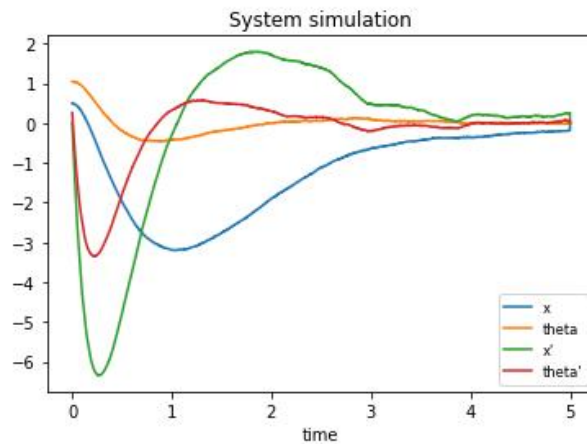
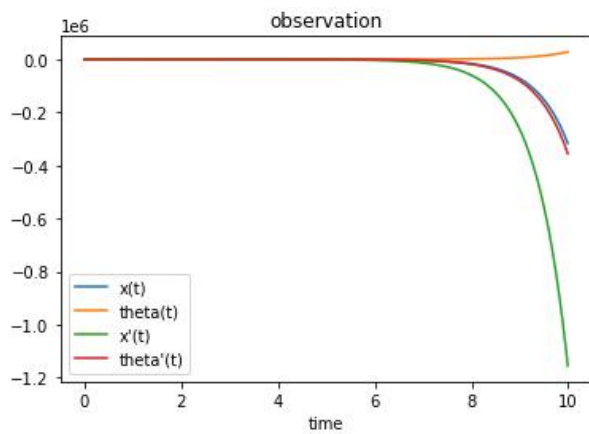
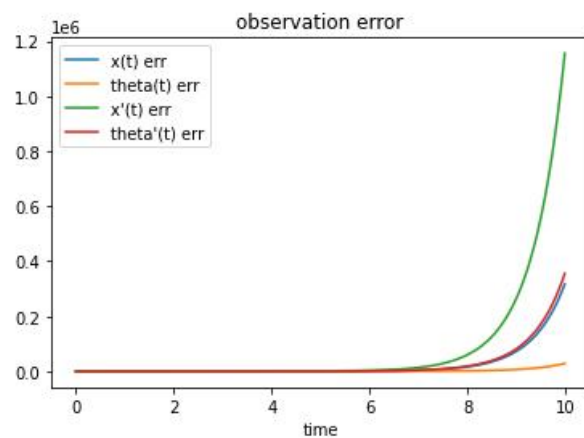


Figure 6: Noisy dynamics.



(a) Observation



(b) Observation error

Figure 7: Luenberg observer at 0.05 scale noise at the dynamics.

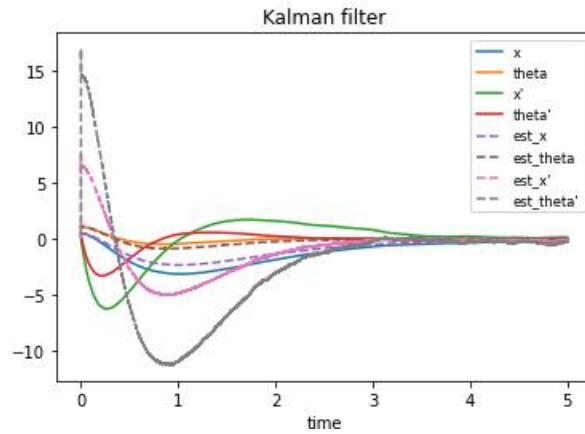


Figure 8: Kalman filter with 0 input

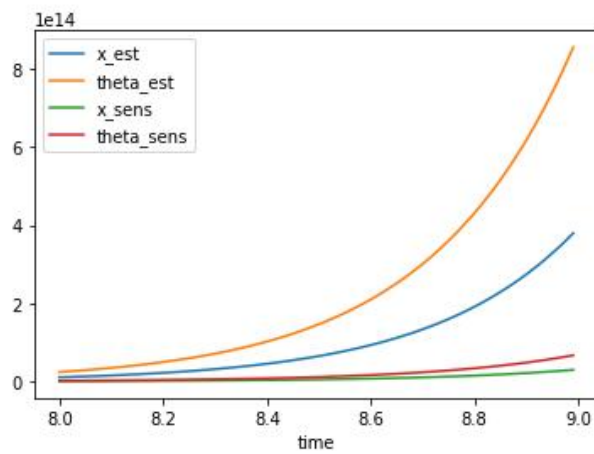


Figure 9: Kalman predictions on uncontrolled system.

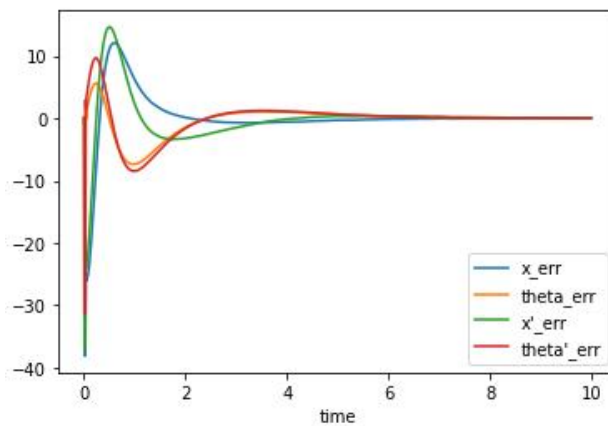


Figure 10: Kalman prediction error on controlled system.