ME424 Project 3 Report

Name: 江铁豪 SID: 11912404

1. solution:

$$u + w = IR + L\dot{I} + V_c$$

$$I = C\dot{V}_c$$

Knowing that $x = \begin{bmatrix} I \\ V_c \end{bmatrix}$

then $\begin{bmatrix} \dot{I} \\ \dot{V}_c \end{bmatrix} = \begin{bmatrix} \frac{1}{L}(w + u - IR - V_c) \\ \frac{I}{C} \end{bmatrix} = \begin{bmatrix} -\frac{R}{L} & -\frac{1}{L} \\ \frac{1}{C} & 0 \end{bmatrix} \begin{bmatrix} I \\ V_c \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} u + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} w$

$$= A_c x(t) + B_c u(t) + G_c w(t)$$

$$A_c = \begin{bmatrix} -\frac{R}{L} & -\frac{1}{L} \\ \frac{1}{C} & 0 \end{bmatrix} = \begin{bmatrix} -3 & -1 \\ 2 & 0 \end{bmatrix}$$

where $B_c = \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$$G_c = \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

2. solution:

$$dt = 0.1$$

$$x_{k+1} = x_k + \dot{x}dt = x_k + (A_c x_k + B_c u_k + G_c w_k)dt$$

$$= (0.1A_c + I)x_k + 0.1B_c u_k + 0.1G_c w_k$$

$$= A_d x_k + B_d u_k + \tilde{w}_k$$

$$A_d = \begin{bmatrix} -\frac{0.1R}{L} + 1 & -\frac{0.1}{L} \\ \frac{0.1}{C} & 1 \end{bmatrix} = \begin{bmatrix} -0.7 & -0.1 \\ 0.2 & 1 \end{bmatrix}$$

where $B_d = \begin{bmatrix} \frac{0.1}{L} \\ 0 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0 \end{bmatrix}$

$$\tilde{w}_k = \begin{bmatrix} \frac{0.1}{L} \\ 0 \end{bmatrix} w_k = \begin{bmatrix} 0.1 \\ 0 \end{bmatrix} w_k$$

$$Cov(\tilde{w}_k) = \begin{bmatrix} 0.1 \\ 0 \end{bmatrix} Cov(w_k) \begin{bmatrix} 0.1 & 0 \end{bmatrix} = 0.25 \begin{bmatrix} 0.01 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.0025 & 0 \\ 0 & 0 \end{bmatrix}$$

3. solution:

(1) First define the system

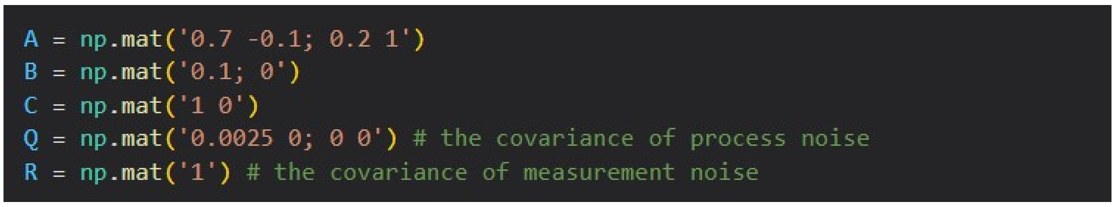
$$x_{k+1} = \begin{bmatrix} 0.7 & -0.1 \\ 0.2 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0.1 \\ 0 \end{bmatrix} u_k + \tilde{w}_k$$

$$y_k = \begin{bmatrix} 0 & 1 \end{bmatrix} x_k + v_k$$

$$\tilde{w}_k \sim N(0, \begin{bmatrix} 0.0025 & 0 \\ 0 & 0 \end{bmatrix})$$

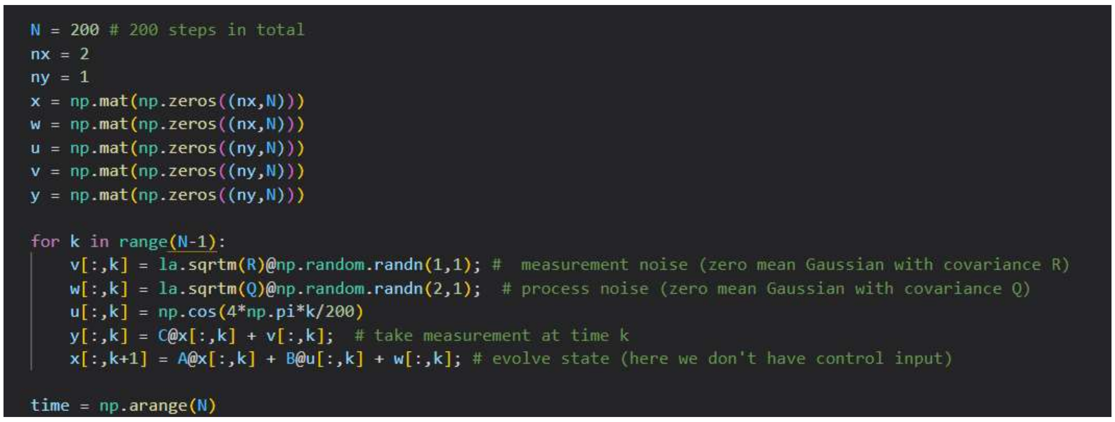
where $v_k \sim N(0,1)$

$$u_k = \cos(\frac{4\pi k}{200})$$



```python
A = np.mat('0.7 -0.1; 0.2 1')
B = np.mat('0.1; 0')
C = np.mat('1 0')
Q = np.mat('0.0025 0; 0 0') # the covariance of process noise
R = np.mat('1') # the covariance of measurement noise
```

(2) Second, start the simulation process in N = 200 steps in total

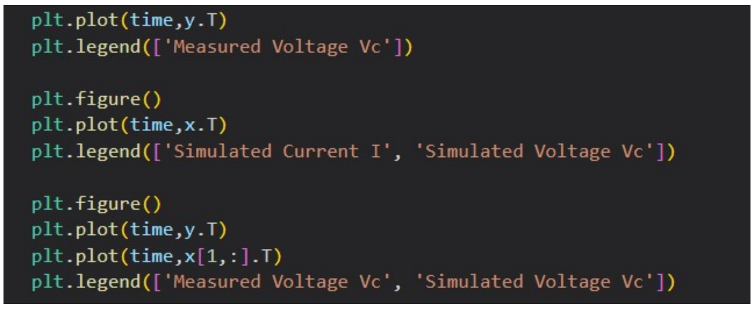


```python
N = 200 # 200 steps in total
nx = 2
ny = 1
x = np.mat(np.zeros((nx,N)))
w = np.mat(np.zeros((nx,N)))
u = np.mat(np.zeros((ny,N)))
v = np.mat(np.zeros((ny,N)))
y = np.mat(np.zeros((ny,N)))

for k in range(N-1):
    v[:,k] = la.sqrtm(R)@np.random.randn(1,1); #  measurement noise (zero mean Gaussian with covariance R)
    w[:,k] = la.sqrtm(Q)@np.random.randn(2,1);  # process noise (zero mean Gaussian with covariance Q)
    u[:,k] = np.cos(4*np.pi*k/200)
    y[:,k] = C@x[:,k] + v[:,k];  # take measurement at time k
    x[:,k+1] = A@x[:,k] + B@u[:,k] + w[:,k]; # evolve state (here we don't have control input)

time = np.arange(N)
```

(3) Third, save the result as ground truth and plot them as comparison



```python
plt.plot(time,y.T)
plt.legend(['Measured Voltage Vc'])

plt.figure()
plt.plot(time,x.T)
plt.legend(['Simulated Current I', 'Simulated Voltage Vc'])

plt.figure()
plt.plot(time,y.T)
plt.plot(time,x[1,:].T)
plt.legend(['Measured Voltage Vc', 'Simulated Voltage Vc'])
```

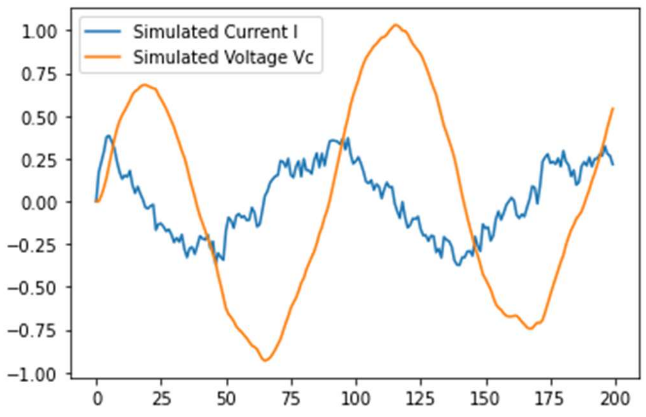


figure 1: the plotting of x_k (simulated current & simulated voltage)

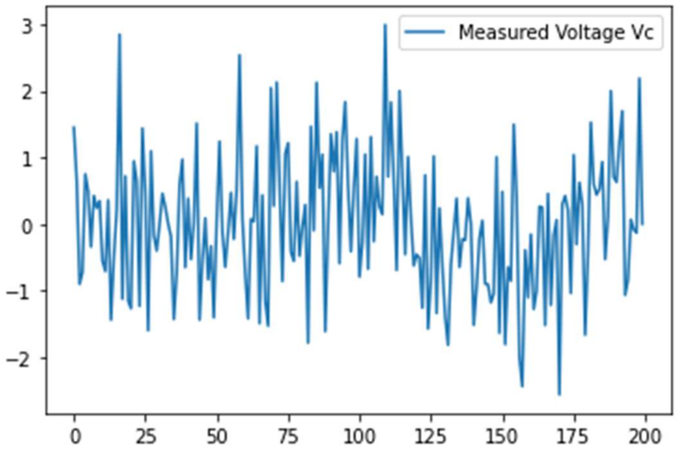


figure 2: the plotting of y_k (Measured voltage)

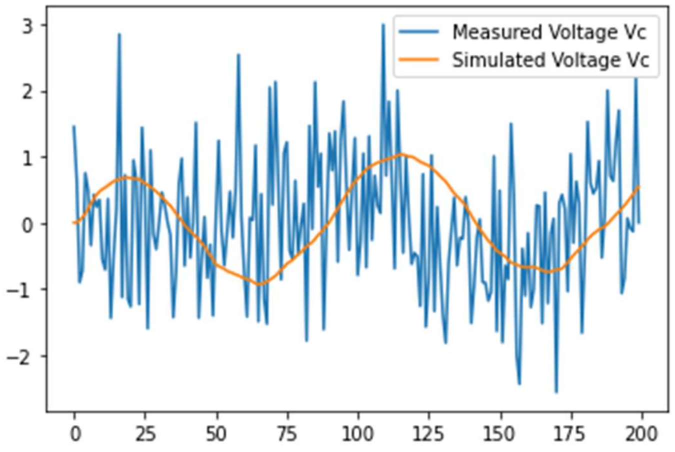
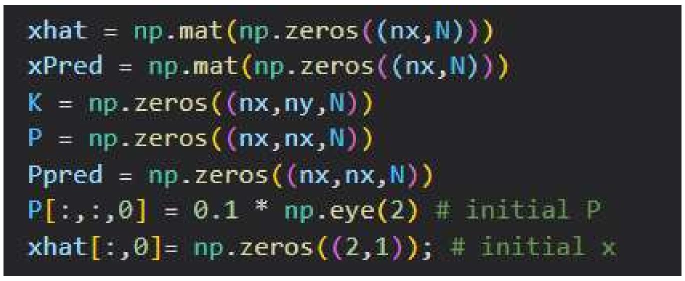


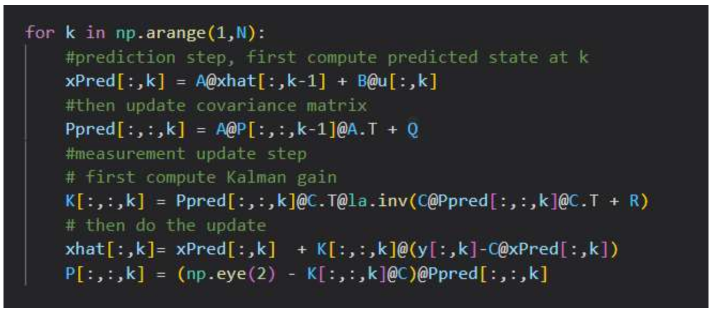
figure 3: the comparison of $x_{k,2}$ & y_k (simulated voltage & measured voltage)

4. Implement of the Kalman Filter
solution:
$\hat{x}_0 = [0 \quad 0]^T$ & $P_0 = 0.1I$



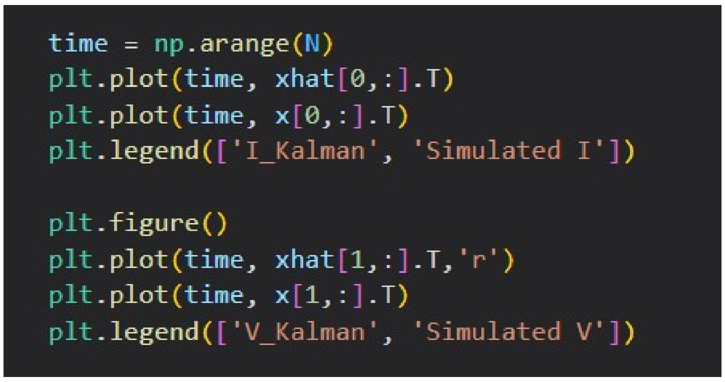
```
xhat = np.mat(np.zeros((nx,N)))
xPred = np.mat(np.zeros((nx,N)))
K = np.zeros((nx,ny,N))
P = np.zeros((nx,nx,N))
Ppred = np.zeros((nx,nx,N))
P[:,:,0] = 0.1 * np.eye(2) # initial P
xhat[:,0]= np.zeros((2,1)); # initial x
```

The simulation process with Kalman Filter:



```
for k in np.arange(1,N):
    #prediction step, first compute predicted state at k
    xPred[:,k] = A@xhat[:,k-1] + B@u[:,k]
    #then update covariance matrix
    Ppred[:,:,k] = A@P[:,:,k-1]@A.T + Q
    #measurement update step
    # first compute Kalman gain
    K[:,:,k] = Ppred[:,:,k]@C.T@la.inv(C@Ppred[:,:,k]@C.T + R)
    # then do the update
    xhat[:,k]= xPred[:,k]  + K[:,:,k]@(y[:,k]-C@xPred[:,k])
    P[:,:,k] = (np.eye(2) - K[:,:,k]@C)@Ppred[:,:,k]
```

The code of the plotting of the result:



```
time = np.arange(N)
plt.plot(time, xhat[0,:].T)
plt.plot(time, x[0,:].T)
plt.legend(['I_Kalman', 'Simulated I'])

plt.figure()
plt.plot(time, xhat[1,:].T,'r')
plt.plot(time, x[1,:].T)
plt.legend(['V_Kalman', 'Simulated V'])
```

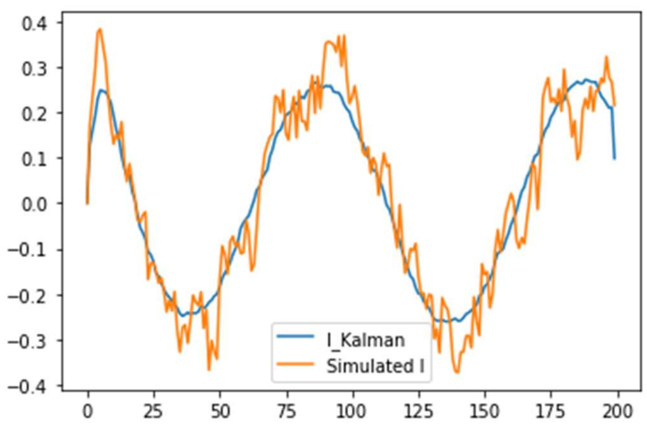


figure 4: the plotting of estimated current & simulated current

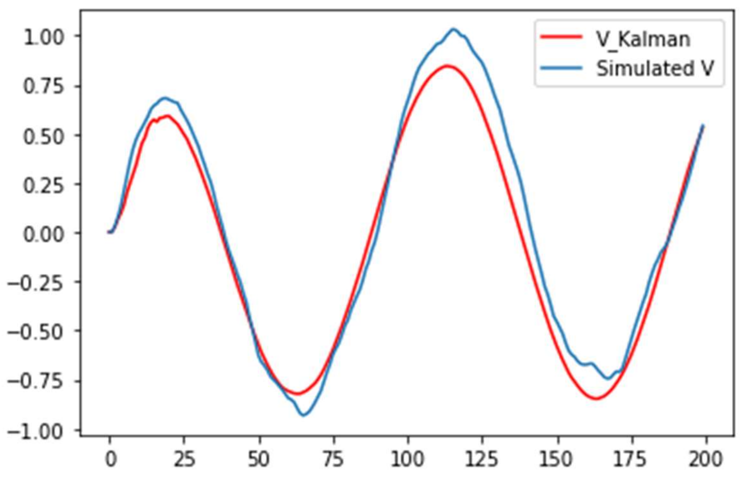
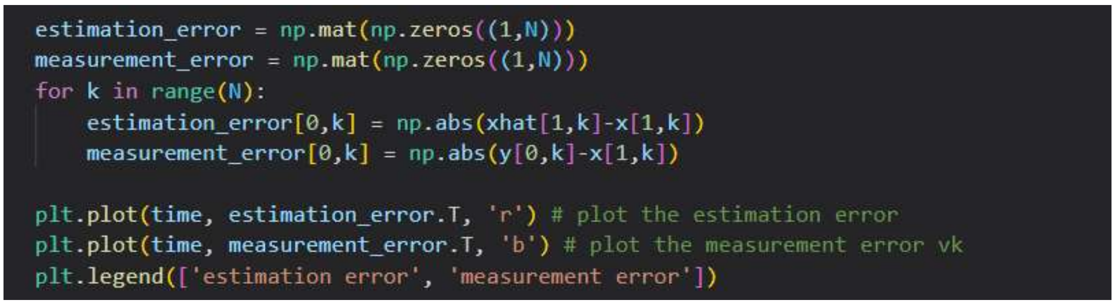


figure 5: the plotting of estimated voltage & simulated voltage

Compute and plot the measurement error v_k and estimation error (Here I used the **absolute value** to define the error):



```
estimation_error = np.mat(np.zeros((1,N)))
measurement_error = np.mat(np.zeros((1,N)))
for k in range(N):
    estimation_error[0,k] = np.abs(xhat[1,k]-x[1,k])
    measurement_error[0,k] = np.abs(y[0,k]-x[1,k])

plt.plot(time, estimation_error.T, 'r') # plot the estimation error
plt.plot(time, measurement_error.T, 'b') # plot the measurement error vk
plt.legend(['estimation error', 'measurement error'])
```

The result is shown as below:

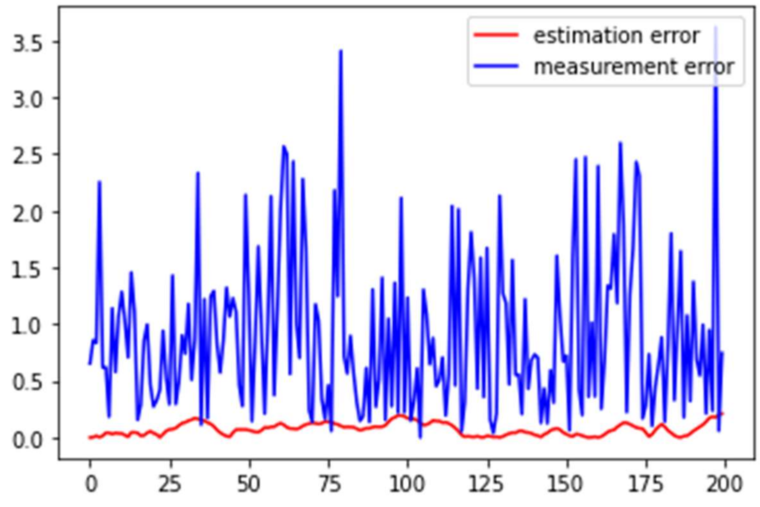
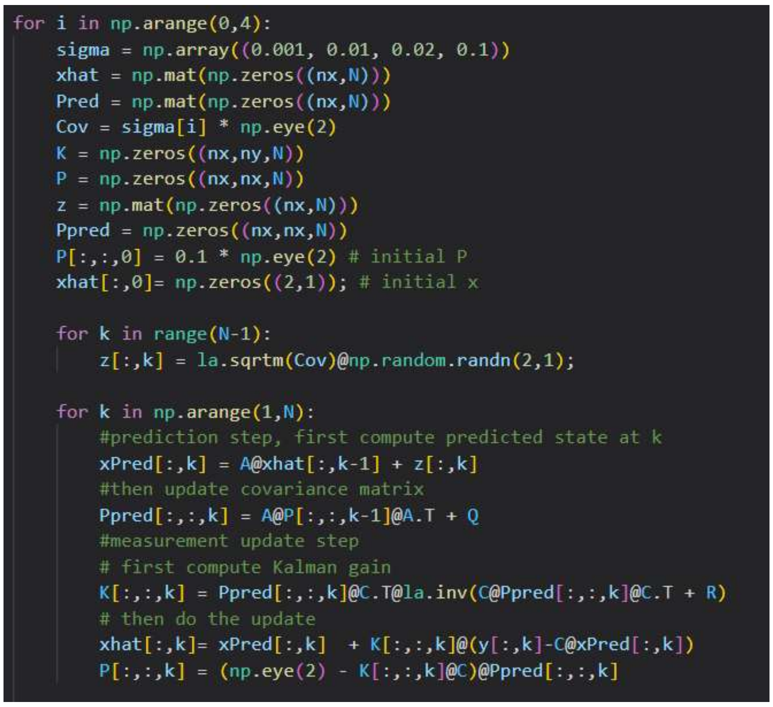


figure 6: the plotting of estimation error & measurement error

From the figure above we can see the result of the estimation error and the measurement error. We can see that **the measurement error without Kalman Filter is larger than the estimation error after Kalman Filter**, which shows that Kaiman Filter has the precision of estimate the data involving the measurement and the prediction.

5. Modified Kalman Filter
solution:
(1) Modify the Kalman Filter and add it into the new update model:

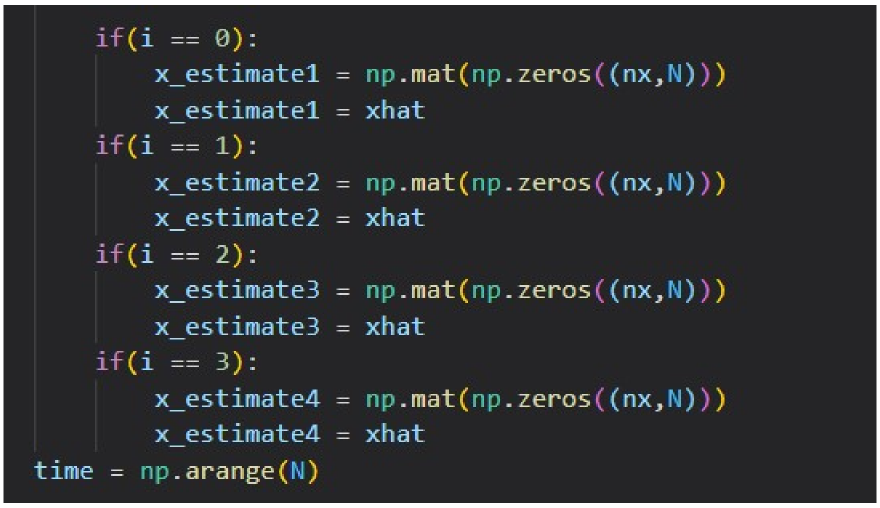


```python
for i in np.arange(0,4):
    sigma = np.array((0.001, 0.01, 0.02, 0.1))
    xhat = np.mat(np.zeros((nx,N)))
    Pred = np.mat(np.zeros((nx,N)))
    Cov = sigma[i] * np.eye(2)
    K = np.zeros((nx,ny,N))
    P = np.zeros((nx,nx,N))
    z = np.mat(np.zeros((nx,N)))
    Ppred = np.zeros((nx,nx,N))
    P[:,:,0] = 0.1 * np.eye(2) # initial P
    xhat[:,0]= np.zeros((2,1)); # initial x

    for k in range(N-1):
        z[:,k] = la.sqrtm(Cov)@np.random.randn(2,1);

    for k in np.arange(1,N):
        #prediction step, first compute predicted state at k
        xPred[:,k] = A@xhat[:,k-1] + z[:,k]
        #then update covariance matrix
        Ppred[:,:,k] = A@P[:,:,k-1]@A.T + Q
        #measurement update step
        # first compute Kalman gain
        K[:,:,k] = Ppred[:,:,k]@C.T@la.inv(C@Ppred[:,:,k]@C.T + R)
        # then do the update
        xhat[:,k]= xPred[:,k]  + K[:,:,k]@(y[:,k]-C@xPred[:,k])
        P[:,:,k] = (np.eye(2) - K[:,:,k]@C)@Ppred[:,:,k]
```

(2) Store the value for σ = 0.001, 0.01, 0.02, 0.1



```python
    if(i == 0):
        x_estimate1 = np.mat(np.zeros((nx,N)))
        x_estimate1 = xhat
    if(i == 1):
        x_estimate2 = np.mat(np.zeros((nx,N)))
        x_estimate2 = xhat
    if(i == 2):
        x_estimate3 = np.mat(np.zeros((nx,N)))
        x_estimate3 = xhat
    if(i == 3):
        x_estimate4 = np.mat(np.zeros((nx,N)))
        x_estimate4 = xhat
time = np.arange(N)
```

(3) Plot the result

σ = 0.001

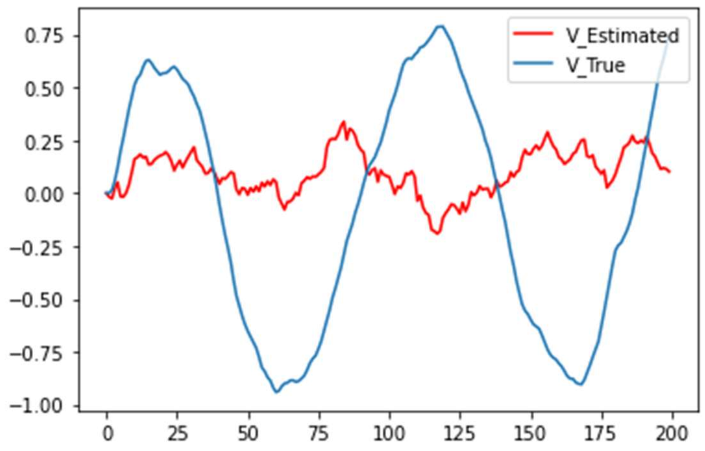


figure 7: the plotting of estimation voltage & true voltage under σ = 0.001

σ = 0.01

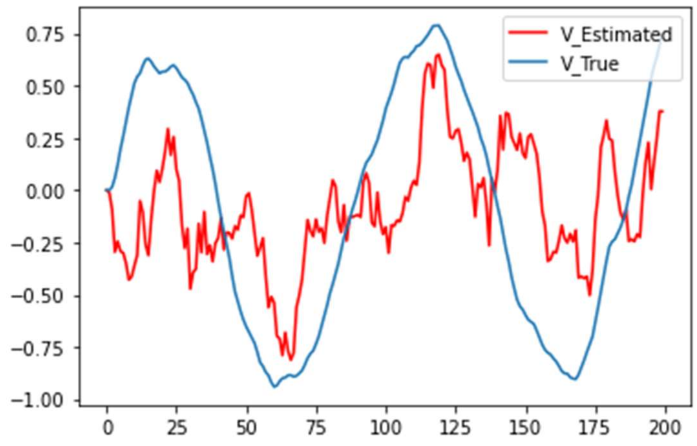


figure 8: the plotting of estimation voltage & true voltage under σ = 0.01

σ = 0.02

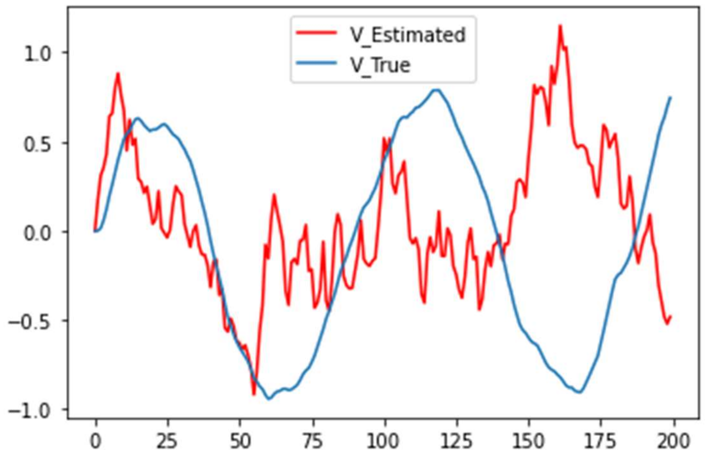


figure 9: the plotting of estimation voltage & true voltage under σ = 0.02

$\sigma = 0.1$

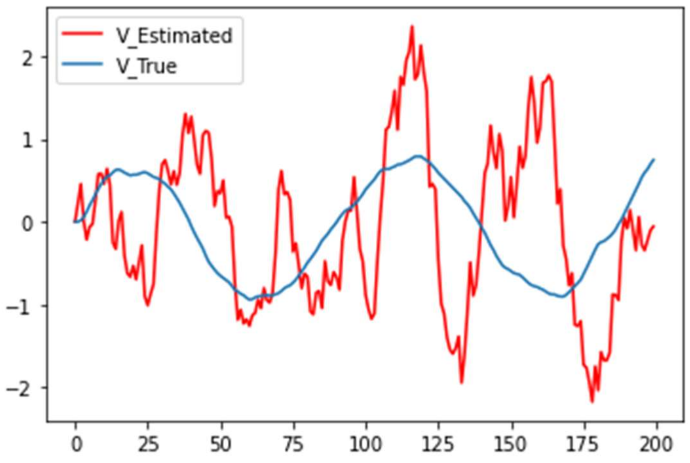
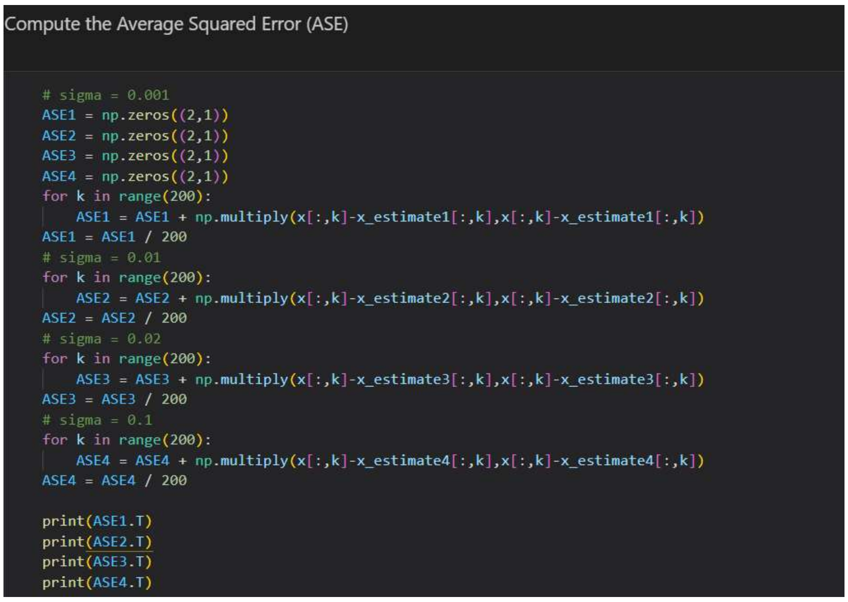


figure 10: the plotting of estimation voltage & true voltage under $\boldsymbol{\sigma} = 0.1$

(4) compute the average squared error:

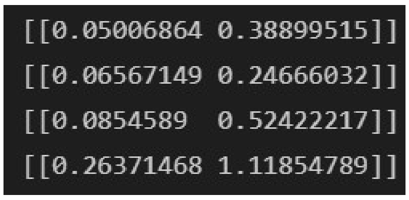


Compute the Average Squared Error (ASE)

```
# sigma = 0.001
ASE1 = np.zeros((2,1))
ASE2 = np.zeros((2,1))
ASE3 = np.zeros((2,1))
ASE4 = np.zeros((2,1))
for k in range(200):
    ASE1 = ASE1 + np.multiply(x[:,k]-x_estimate1[:,k],x[:,k]-x_estimate1[:,k])
ASE1 = ASE1 / 200
# sigma = 0.01
for k in range(200):
    ASE2 = ASE2 + np.multiply(x[:,k]-x_estimate2[:,k],x[:,k]-x_estimate2[:,k])
ASE2 = ASE2 / 200
# sigma = 0.02
for k in range(200):
    ASE3 = ASE3 + np.multiply(x[:,k]-x_estimate3[:,k],x[:,k]-x_estimate3[:,k])
ASE3 = ASE3 / 200
# sigma = 0.1
for k in range(200):
    ASE4 = ASE4 + np.multiply(x[:,k]-x_estimate4[:,k],x[:,k]-x_estimate4[:,k])
ASE4 = ASE4 / 200

print(ASE1.T)
print(ASE2.T)
print(ASE3.T)
print(ASE4.T)
```

The result is shown as below:



```
[[0.05006864 0.38899515]]
[[0.06567149 0.24666032]]
[[0.0854589  0.52422217]]
[[0.26371468 1.11854789]]
```

The first one is the average squared error of $\boldsymbol{\sigma} = \mathbf{0.001}$;
The second one is the average squared error of $\boldsymbol{\sigma} = \mathbf{0.01}$;
The third one is the average squared error of $\boldsymbol{\sigma} = \mathbf{0.02}$;
The last one is the average squared error of $\boldsymbol{\sigma} = \mathbf{0.1}$;

Explanation:

We can see that as σ grows, the value of the average squared error shows a increasing trend although there might be some fluctuation. Since σ is called scale parameter in the Gauss' distribution, which defined the degree of the discretization of the Gauss' distribution model with the data. As σ grows, the value of a larger input z_k have a larger value more prospectively, which will cause a larger error.