State Estimation,Autonomy,Machine Learning &Energy System

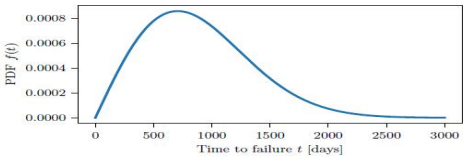
Capstone Report for Course 4

Problem 1

Hint: Feel free to use a computer algebra system such as **sympy** *for this problem. Be sure to explain where you used it in your answer.* After an overhaul, a water pump reliably operates for a random time t before a failure occurs. The probability density function (PDF) for the failure time t is given by

$$f(t) = \begin{cases} \frac{2t}{\lambda^2} \exp\left(-\frac{t^2}{\lambda^2}\right) & \text{if } t \geq 0 \\ 0 & \text{else} \end{cases}$$

where λ is a parameter, with specifically $\lambda = 1000$days. The PDF is plotted below:



What is the probability density function(PDF)?

```
t = symbols('t')
Lambda = symbols('λ')
pdf = (2*t*exp((-((t**2))/(Lambda**2))))/(Lambda**2)
```

```
pdf
```

$$\frac{2te^{-\frac{t^2}{\lambda^2}}}{\lambda^2}$$

```
Lambda = 500 # λ = 500 days
pdf = (2*t*exp((-((t**2))/(Lambda**2))))/(Lambda**2)
```

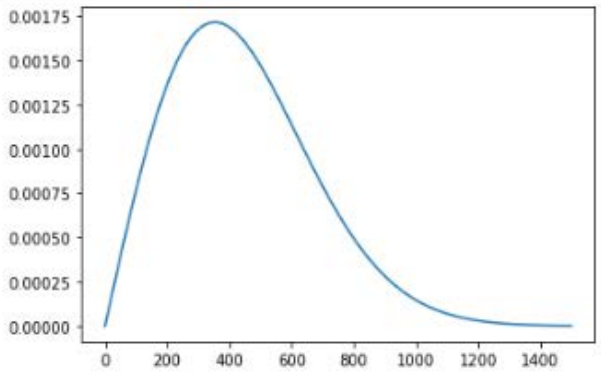
```
pdf
```

$$\frac{te^{-\frac{t^2}{250000}}}{125000}$$

Ploting the PDF:

```
def PDF(t):
    return (2*t*exp((-((t**2))/(Lambda**2))))/(Lambda**2)
```

```
lst = []
for t_ in range(1500):
    lst.append(PDF(t_))

plt.plot(np.array(lst));
```



Check if it's a valid PDF!

- non-negatives

```
array = np.array(lst)
print(f'number of nigative values: {(array<0).sum()}')
```

```
number of nigative values: 0
```

- area under the curve

```
print("area under the curve is:", integrate(pdf, [t,0,oo]))
```

```
area under the curve is: 1
```

How long after installation should we do preventative maintenance if we wish to have the probability of unexpected failure be less than 1%, 10%, 50%, and 99%?

```
def when_to_preventative_maintenance(probability):

    for i in range(1500):
        day = array[:i]

        if np.trapz(day) > probability:
            return i
```

```
probabilities = [0.01,0.1,0.5,0.99]
for probability in probabilities:
    n_day = when_to_preventative_maintenance(probability)
    print(f'We should prevent maintenance in day {n_day} to have the probability of unexpected f
```

```
We should prevent maintenance in day 52 to have the probability of unexpected failure be less
than 1%
We should prevent maintenance in day 164 to have the probability of unexpected failure be les
s than 10%
We should prevent maintenance in day 418 to have the probability of unexpected failure be les
s than 50%
We should prevent maintenance in day 1074 to have the probability of unexpected failure be le
ss than 99%
```

What is the expected lifetime for this pump? What is the probability of failure before the expected lifetime?

```
exp_l_t = integrate(pdf*t, [t,0,oo])
print(f'the expected lifetime for this pump is {int(exp_l_t)} days')
pf = integrate(pdf,[t,0,exp_l_t])
print(f'the probability of failure before the expected lifetime {float(pf)*100}%')
```

```
the expected lifetime for this pump is 443 days
the probability of failure before the expected lifetime 54.40618722340038%
```

What is the variance of the pump's lifetime? What is the range of the lifetime that falls within one standard deviation of the expected value?

```
variance = integrate(pdf*((t - exp_l_t )**2) , [t,0, oo])
print(f'the variance of the pump's lifetime is {round(variance,2)}')
print(f'the range of the lifetime that falls within one standard deviation of the expected value is {round(sqrt(varianc
```

```
the variance of the pump's lifetime is 53650.46
the range of the lifetime that falls within one standard deviation of the expected value is 231.63
```

Write a program that generates samples of t from its distribution!

```
def generator(tm):

    print(f'for number of times = {tm}:')

    RCs = np.zeros((10**6,))
    Ss = np.zeros((10**6,))

    for i in range(10**6):

        uf = np.random.uniform()
        Ss[i] = math.log(-1/(uf-1))*1000

        if Ss[i] <= tm:
            RCs[i] = 250/Ss[i]
        else:
            RCs[i] = 50/tm

    print(f'\tthe average running cost is {round(np.mean(RCs),2)}$')
    print(f'\tthe sample average is {round(np.mean(Ss),2)}')
    print(f'\tthe sample variance is {round(np.var(Ss),2)}')
```

```
for tm in [1,10,100,1000,10000]:
    generator(tm)
```

```
for number of times = 1:
        the average running cost is 71.89$
        the sample average is 1001.8
        the sample variance is 1004503.53
for number of times = 10:
        the average running cost is 7.47$
        the sample average is 998.43
        the sample variance is 995815.84
for number of times = 100:
        the average running cost is 3.47$
        the sample average is 1001.01
        the sample variance is 1000568.6
for number of times = 1000:
        the average running cost is 3.24$
        the sample average is 1000.35
        the sample variance is 999921.15
for number of times = 10000:
        the average running cost is 3.53$
        the sample average is 999.65
        the sample variance is 998694.59
```

Problem 2

In a substation, there are three types of transformers (from three different manufacturers: 1, 2, and 3). The probability that a transformer is operating out of specification during a day is shown in the table below.

manufacturer	1	2	3
number of transformers	8	5	20
prob. of out-of-spec	2/1000	3/1000	4/1000

An inspector chooses a machine at random, inspects it, and determines that it is outside the specifications. Compute the probability that it is from manufacturer 1.

Transformer 1 manufacturer has been cosider it outsdie the specification ,To determine that probability as follow :

- Manufacturer 1 has 5 transformers,Manufacturer 2 has 8 transformers and Manufacturer 3 has 12 transformers.
- Total Transformers in the subsation = 5 + 8 +12 = 25:

$$P = \frac{2}{1000} * \frac{5}{25} + \frac{3}{1000} * \frac{8}{25} + \frac{4}{1000} * \frac{12}{25} = 0.328\%$$

- The Probabilty that inspector choosing tranformer randomly from manufacure 1 is =

$$P(1) = \frac{5}{25} = 0.2 = 20\%$$

```
p = (2*8 + 3*5 + 4*20)/(1000*(8+5+20))
def prob_out(n_trans, p_out,p=p):

    return (p_out*(n_trans/(8+5+20)))/p
```

```
m1 = prob_out(8, 2/1000,p)
m2 = prob_out(5, 3/1000,p)
m3 = prob_out(20, 4/1000,p)
```

```
prob = m1*100/(m1+m2+m3)
print(f"There's a probability that it's from manufacturer 1 of {prob}%")
```

```
There's a probability that it's from manufacturer 1 of 14.414414414414415%
```

Problem 3

A battery's state of charge at time step k is given by $q(k)$, with $q(k) = 1$ corresponding to fully charged and $q(k) = 0$ depleted. In each time step (e.g. each hour) k, the battery powers a process which consumes energy $j(k) = j_0 + v(k)$, where j_0 is the average amount of energy consumed, and $v(k)$ is a random deviation, so that

$$q(k) = q(k-1) - j(k-1)$$

We have perfect knowledge of the battery's initial charge, $q(0) = 1$, and we know that $v(k)$ is white and normally distributed as $\mathcal{N}(0, \sigma_v^2)$.

We want to know how many time steps of the process we can safely use before checking on the battery. *Note that the model allows for arbitrarily states of charge (where $q < 0$ or $q > 1$) though such states are not physically meaningful.*

a. Analytically compute the mean and variance of $q(k)$ across k.

b. After how many time steps does the mean of $q(k)$ drop below zero?

c. As a safety measure, we only want to run the process if we're confident that the battery has sufficient charge remaining to complete the process. How many time steps can I run until the battery is within 3 standard deviations of fully discharged? (Recall: the standard deviation changes over time, and at time k can be computed as $\sigma_q(k) = \sqrt{\mathrm{Var}\,[q(k)]}$; moreover, for a normally distributed random variable the probability of being more than three standard deviations below the mean is approximately 0.15%)

For the remainder of this problem, set $j_0 = 0.1$ and $\sigma_v = 0.05$.

d. Compare the numerical value you got for parts b and c. Comment on the difference.

e. Simulate 10^6 different evolutions, compute their mean & standard deviation across $k \in \{0, 1, \ldots, 20\}$. Make three plots against time: the mean $\mathrm{E}\,[q(k)]$, the standard deviation $\sqrt{\mathrm{Var}\,[q(k)]}$, and the fraction of sample paths where the charge $q(k)$ at time k is less than or equal to zero.

As Given the system Time Step, Battery Functions as follow :

$$q(k) = q(k-1) - j(k-1)$$

$$j(k) = j_0 + v(k)$$

To substitute the above functions :

$$q(k) = q(k-1) - j_0 - v(k-1)$$

To find the mean of above function :

Note that $v(k) = \mathcal{N}(0, \sigma_v^2)$ is white and normally distributed as given.

$$E[q_k] = \sum_{q_k} q_k . f(q_k)$$

$$E[q_k] = Eq(k-1) - E(j(k)$$

$$E[q_k] = Eq(k-1) - j_0 - 0 = Eq(k-1) - j_0$$

The mean value will be =

$$E[k] = 1 - kj_0$$

To Find the Variance :

$$Var[x] = E[(x - E[x])(x - E[x])^T]$$

$$Var[k] = \sum_k E(k - E[k])^2$$

The J0 here is constant and not vary in time , there it consider zero

$$Var[K] = Var[q(k-1)] + Var[v(k-1)]$$

The $q(k-1)$is independed from $v(k-1)$ and it will be zero in this case also j_0 has no Variance , because its constant

The Variance will be determine as follow :

$$Var[K] = k\sigma^2$$

B.

The mean of q(k) drop below zero:

$$q_k = 1 - kj_0 < 0$$

$$k > \frac{1}{j_0}$$

c.

$$Std = \sqrt{Var} = \sigma\sqrt{k}$$

$$E[q_k] - 3\sigma\sqrt{k} = 0$$

$$1 - kj_0 - 3\sigma\sqrt{k} = 0$$

$$-kj_0 - 3\sigma\sqrt{k} + 1 = 0$$

$$\sqrt{k} = \frac{3\sigma \pm \sqrt{9\sigma^2 + 4j_0}}{2(-j_0)}$$

$$k = (\frac{3\sigma \pm \sqrt{9\sigma^2 + 4j_0}}{2(-j_0)})^2$$

$$Assume \ : \ j_0 = 0.1 \ and \ \sigma = 0.05$$

```
j = 0.1
σ = 0.05
K1 = (3*σ + (9*(σ**2) +4*j)**-2/-j*2)**2
K2 = (3*σ - (9*(σ**2) +4*j)**-2/-j*2)**2

print(f'in case 1 it required {round(K1,2)} secound')
print(f'in case 2 it required {round(K2,2)} secound')
```

```
in case 1 it required 12519.57 secound
in case 2 it required 12586.8 secound
```

D. Compare the numerical value you got for parts b and c. Comment on the difference.

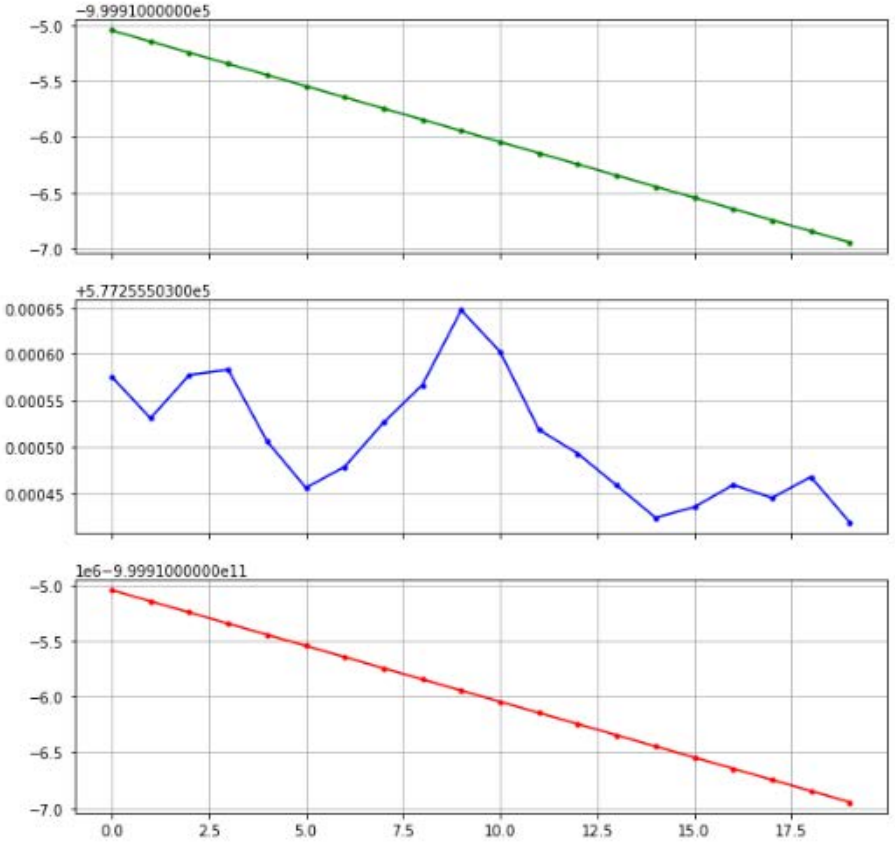
In case 2 when K = 6.25 after the round will assume approximately 6.0 as there is no time step equal to 6.25.

- If we compare parts b and c for the value of (k), the battery charge state q(k) will be < zero when K is > 10, as observed in part b.
- In part c, two values are obtained, but the best value is chosen in case 2,where k = to 6.0 it's recommended to stop using the battery before dropping < zero.

- To Simulate 10^6 different evolutions, compute their mean & standard deviation across $k \in \{0, 1, .. , 20\}$.

```
K = 1
Ks = np.zeros([10**6,20])
times = np.arange(20)

for a in range(10**6):
    for b in range(20):
        Ks[a,b] = K
        v = np.random.normal(0,σ)
        K = K - j - v
```

```
fig, ax = plt.subplots(3,1,sharex=True)
fig.set_size_inches(10,10)
ax[0].plot(times,np.mean(Ks,0),'g.-')
ax[0].grid()
ax[1].plot(times,np.std(Ks,0),'b.-')
ax[1].grid()
ax[2].plot(times,np.sum(Ks,0),'r.-')
ax[2].grid()
```



Problem 4

In this problem we estimate the behavior of a city's water network, where fresh water is supplied by a desalination plant, and consumed at various points in the network. The network contains various consumers, and various reservoirs where water is locally stored.
Our objective is to use noisy information about production and consumption levels, and noisy measurements of the amount of water in the reservoirs, to estimate the state of the network. We will consider 3 networks of increasing complexity.
(Hint – use code to compute numerical solutions, don't compute by hand.)

Let k be a time index, and let $d(k)$ be the fresh water produced by the desalination plant at time k. Each city district i has a reservoir with current level $r_i(k)$, and the district consumes a quantity of water $c_i(k)$. At each time step, we receive a noisy measurement $z_i(k) = r_i(k) + w_i(k)$ of the reservoir level $r_i(k)$, where $w_i(k)$ is the measurement error (with w zero-mean, white, and independent of all other quantities).

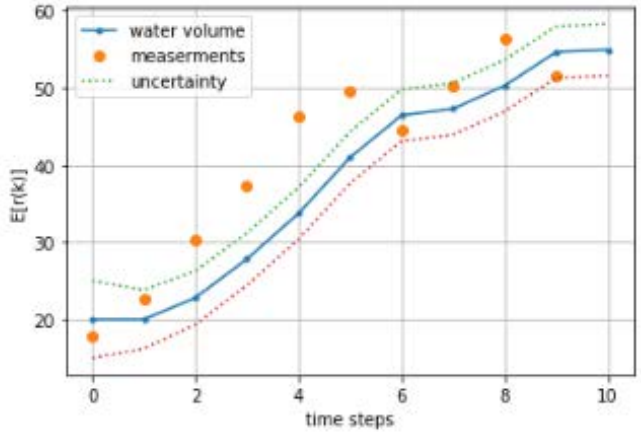
a. We will first investigate a very simplified system, with a single reservoir and single consumer (i.e. we only have one tank level r to keep track of). We will model our consumers as $c(k) = m + v(k)$, where m is the typical consumption, and $v(k)$ is a zero-mean uncertainty, assumed white and independent of all other quantities.


```
steps = 10
er0 = 20
uncertainty_Vr0 = 25
dk = 10
m = 7
uf = dk - m
process_uncertainty = 9
sensor_uncertainty = 25
measurs = [17.8, 22.6, 30.2, 37.3, 46.2, 49.5, 44.6, 50.3, 56.3, 51.6]
dynamic = np.mat([[1]])
model = np.mat([[1]])
noise_variance = np.mat([[25]])
volumes = np.zeros([steps+1])
uncertainties = np.zeros([steps+1])
volumes[0] = er0
uncertainties[0] = uncertainty_Vr0
```

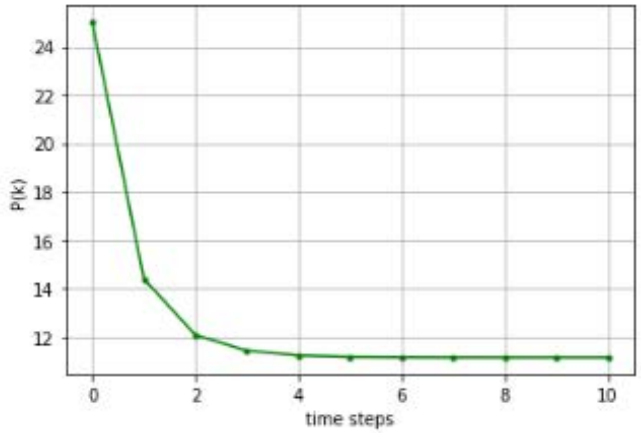
```
for k in range(1,steps+1):

    Kp = uf + dynamic*er0
    K_uncertainty = sensor_uncertainty * (dynamic**2) + process_uncertainty

    measur = measurs[k-1]

    K = K_uncertainty @ model.T @ np.linalg.inv(model @ K_uncertainty @ model.T + noise_variance
    er0 = Kp + K @ (measur - model @ Kp)
    sensor_uncertainty = (np.eye(1) - K @ model) @ K_uncertainty @ (np.eye(1) - K @ model).T + R

    volumes[k] = er0[0]

    uncertainties[k] = sensor_uncertainty[0]
```

(a-i)Write down the model equations for this problem. Make explicit what is the system state, the measurement, the process noise, and the measurement noise.
(a-ii) Design a Kalman filter to estimate the level of the tank. Run the Kalman filter for ten steps, with the following problem data:
At time k = 0, we know E[r(0)] = 20, with uncertainty Var [r(0)] = 25. The desalination plant delivers a constant supply of water, so that d(k) = 10 for all k ≥ 0. The consumer is predicted to use a supply m = 7, and our process uncertainty is Var [v(k)] = 9. Our sensor uncertainty is Var [w(k)] = 25.
We receive the following sequence of measurements z(k):
time k 1 2 3 4 5 6 7 8 9 10
measurement z(k) 17.8 22.6 30.2 37.3 46.2 49.5 44.6 50.3 56.3 51.6
Using the Kalman filter, estimate the actual volume of water r for k $\in$ {1, 2, . . . , 10}. Also provide the associated uncertainty for your estimate of r. Provide this using graphs.

```
plt.plot(volumes,'.-',label="water volume")
plt.plot(measurs,'o',label="measerments")
plt.plot(volumes+np.sqrt(uncertainties),':',label="uncertainty")
plt.plot(volumes-np.sqrt(uncertainties),':')

plt.ylabel('E[r(k)]')
plt.xlabel('time steps')
plt.legend()
plt.grid(True)
```




```
plt.plot(uncertainties,'g.-')
plt.ylabel('P(k)')
plt.xlabel('time steps')
plt.grid(True)
```



We now extend the previous part by also estimating the consumption, c(k). We model the consumption as evolving as c(k) = c(k−1) + n(k−1), where n(k−1) is a zero-mean random variable, which is white and independent of all other quantities.

3

(b-i) Write down the model equations for this, noting that now your state has dimension 2. Make explicit what is the system state, the measurement, the process noise, and the measurement noise.

(b-ii) Again, design a Kalman filter to estimate the level of the tank. Use the same problem data as before (including as given in subproblem aa-ii), except that now Var [n(k)] = 0.1. Also use E[c(0)] = 7 with Var [c(0)] = 1.

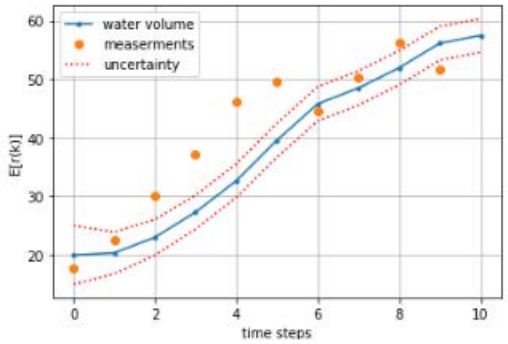
Using the Kalman filter, estimate the actual volume of water r for k $\in$ {1, 2, . . . , 10}, and the consumption rate c(k), and also provide the associated uncertainty for both.

Explicitly provide the filter initialization, and then present the estimate and uncertainty using graphs. Also, provide a comment: How do your answers for r compare to the previous case?

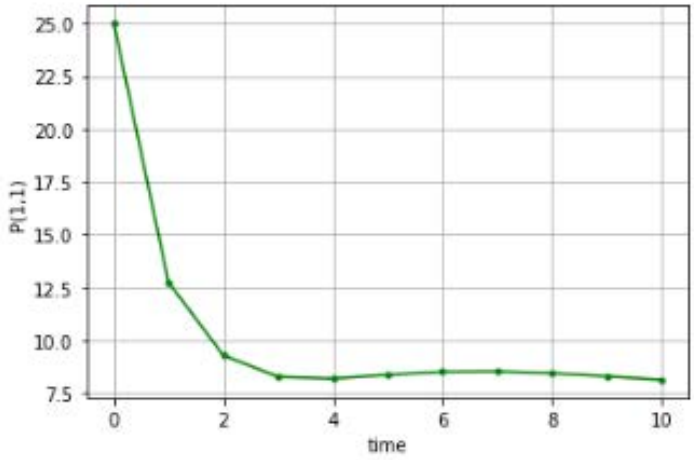
```
dynamics = np.mat([[1,-1],[0,1]])
model = np.mat([[1,0]])
noise_variance = np.mat([[25]])
ii = np.eye(2)
er0 = np.mat([[20],[7]])
uf = np.mat([[10],[0]])
process_uncertainty = np.mat([[0,0],[0,0.1]])
sensor_uncertainty = np.mat([[25,0],[0,1]])
volumes = np.zeros([steps+1,2])
uncertainties = np.zeros([steps+1,2])
volumes[0,0] = er0[0,0]
volumes[0,1] = er0[1,0]
uncertainties[0,0] = sensor_uncertainty[0,0]
uncertainties[0,1] = sensor_uncertainty[1,1]
```

```
for k in range(1,steps+1):

    Kp = dynamics @ er0 + uf

    K_uncertainty = dynamics @ sensor_uncertainty @ dynamics.T + process_uncertainty

    measur = measurs[k-1]

    K = K_uncertainty @ model.T @ np.linalg.inv(model @ K_uncertainty @ model.T + noise_variance)
    er0 = Kp + K @ (measur - model @ Kp)
    sensor_uncertainty = (ii-K @ model)@ K_uncertainty @(ii-K @ model).T + K @ noise_variance @ K.T

    volumes[k,0] = er0[0,0]
    volumes[k,1] = er0[1,0]

    uncertainties[k,0] = sensor_uncertainty[0,0]
    uncertainties[k,1] = sensor_uncertainty[1,1]
```

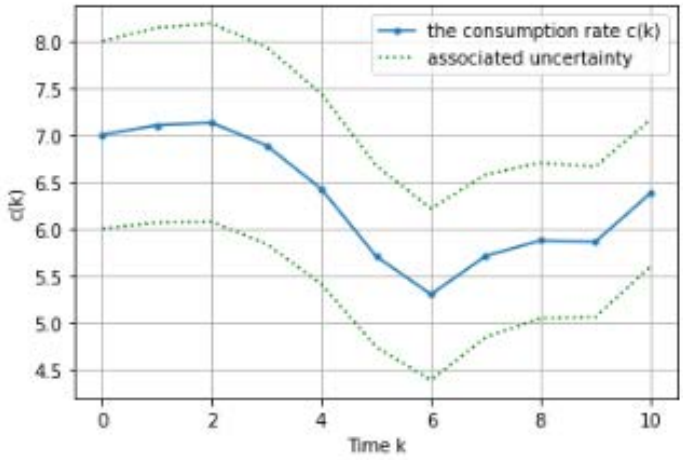
```
plt.plot(volumes[:,0],'.-',label="water volume")
plt.plot(measurs,'o',label="measerments")
plt.plot(volumes[:,0]+np.sqrt(uncertainties[:,0]),'r:',label="uncertainty")
plt.plot(volumes[:,0]-np.sqrt(uncertainties[:,0]),'r:')

plt.ylabel('E[r(k)]')
plt.xlabel('time steps')
plt.legend()
plt.grid(True)
```



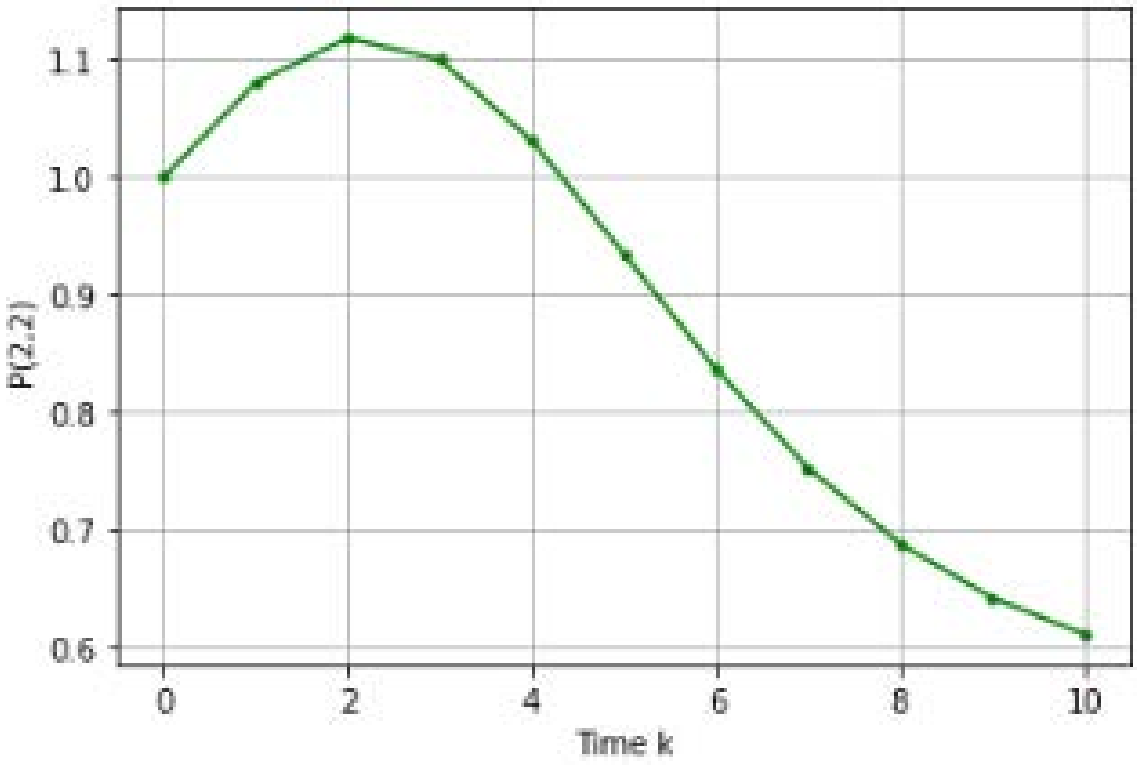
```
plt.plot(uncertainties[:,0],'g.-')
plt.xlabel('time')
plt.ylabel('P(1,1)')
plt.grid(True)
```




```
plt.plot(volumes[:,1],'.-',label="the consumption rate c(k)")
plt.plot(volumes[:,1]+np.sqrt(uncertainties[:,1]),'g:',label="associated uncertainty")
plt.plot(volumes[:,1]-np.sqrt(uncertainties[:,1]),'g:')

plt.xlabel('Time k')
plt.ylabel('c(k)')
plt.grid(True)
plt.legend();
```



```
plt.plot(uncertainties[:,1],'g.-',label="P(k)(2,2)")
plt.xlabel('Time k')
plt.ylabel('P(2,2)')
plt.grid(True)
```



The reservoirs are connected in a network, and an automatic balancing system is in place that pumps water between the reservoirs. If two reservoirs i and j are connected, there is a balancing flow between them that is proportional to the difference in volume between the two, so that fij(k) = α (rj(k) − ri(k)), where α is a constant. For example, for reservoirs #1 and #2 in our network, we have the following dynamics:

r1(k) = r1(k−1) + d(k−1) − c1(k−1) + α (r2(k−1) − r1(k−1)) + α (r3(k−1) − r1(k−1))

r2(k) = r2(k−1) − c2(k−1) + α (r1(k−1) − r2(k−1)) + α (r3(k−1) − r2(k−1))

where, as before, ci(k) = ci(k−1) + ni(k−1). Note that the balancing does not change the total amount of water in the system, it just moves it around. Our sensor uncertainty is Var [wi(k)] = 25 for all tanks, and the consumption uncertainty is Var [ni(k)] = 0.1 for all consumers. Model the consumers as independent but identically distributed; also model the sensors as independent but identically distributed.

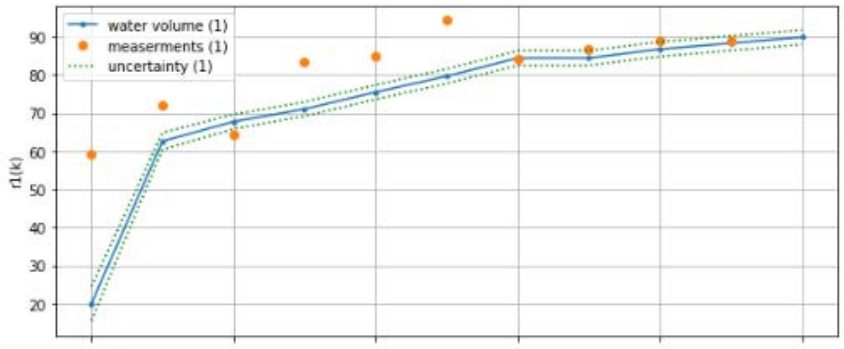
(c-i) Write down the model equations for this problem. Make explicit what is the system state, the measurement, the process noise, and the measurement noise. Write the solution as a linear problem, and clearly give terms of the matrices A, H, etc.

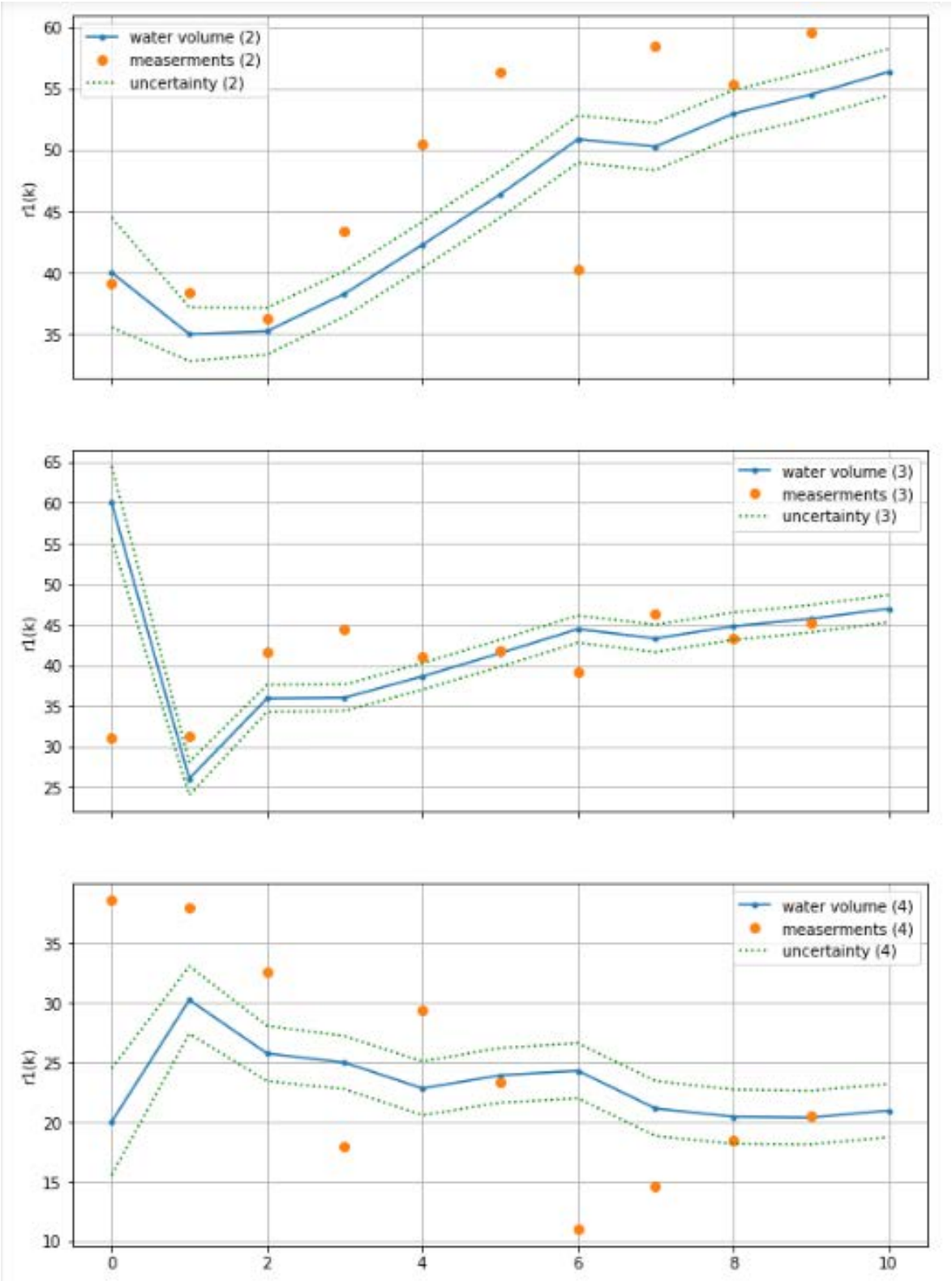
(c-ii) Design a Kalman filter to estimate the level of all the tanks, and the consumption levels. Run the Kalman filter for ten steps, with the following problem data:

```
sigma = 0.3
dynamics = np.matrix([[1-2*sigma, sigma, sigma, 0, -1, 0, 0, 0],
                [sigma, 1-2*sigma, sigma, 0, 0, -1, 0, 0],
                [sigma, sigma, 1-3*sigma, sigma, 0, 0, -1, 0],
                [0, 0, sigma, 1-sigma, 0, 0, 0, -1],
                [0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 0],
                [0,0,0,0,0,0,0,1]])
process_uncertainty=np.diag([0,0,0,0,0.1,0.1,0.1,0.1])
model = np.eye(4,8)
noise_variance = np.eye(4)*25
measur1 = np.array([59.3, 72, 64.4, 83.6, 84.9, 94.3, 84, 86.6, 89, 89.1])
measur2 = np.array([39.1, 38.4, 36.2, 43.4, 50.5, 56.3, 40.3, 58.5, 55.4, 59.6])
measur3 = np.array([31.1, 31.2, 41.6, 44.4, 41, 41.9, 39.2, 46.3, 43.3, 45.3])
measur4 = np.array([38.6, 38, 32.6, 18, 29.4, 23.3, 11, 14.6, 18.4, 20.5])
measurs_list = [measur1,measur2,measur3,measur4]
measurs = np.mat(measurs_list)
er0 = np.mat([[20],[40],[60],[20],[7],[7],[7],[7]])
sensor_uncertainty= np.diag([20,20,20,20,1,1,1,1])
uf = [[30],[0],[0],[0],[0],[0],[0],[0]]
volumes = np.zeros([steps+1,8])
uncertainties = np.zeros([steps+1,8])
volumes[0,0] = er0[0,0]
volumes[0,1] = er0[1,0]
volumes[0,2] = er0[2,0]
volumes[0,3] = er0[3,0]
volumes[0,4] = er0[4,0]
volumes[0,5] = er0[5,0]
volumes[0,6] = er0[6,0]
volumes[0,7] = er0[7,0]
uncertainties[0,0] = sensor_uncertainty[0,0]
uncertainties[0,1] = sensor_uncertainty[1,1]
uncertainties[0,2] = sensor_uncertainty[2,2]
uncertainties[0,3] = sensor_uncertainty[3,3]
uncertainties[0,4] = sensor_uncertainty[4,4]
uncertainties[0,5] = sensor_uncertainty[5,5]
uncertainties[0,6] = sensor_uncertainty[6,6]
uncertainties[0,7] = sensor_uncertainty[7,7]
```

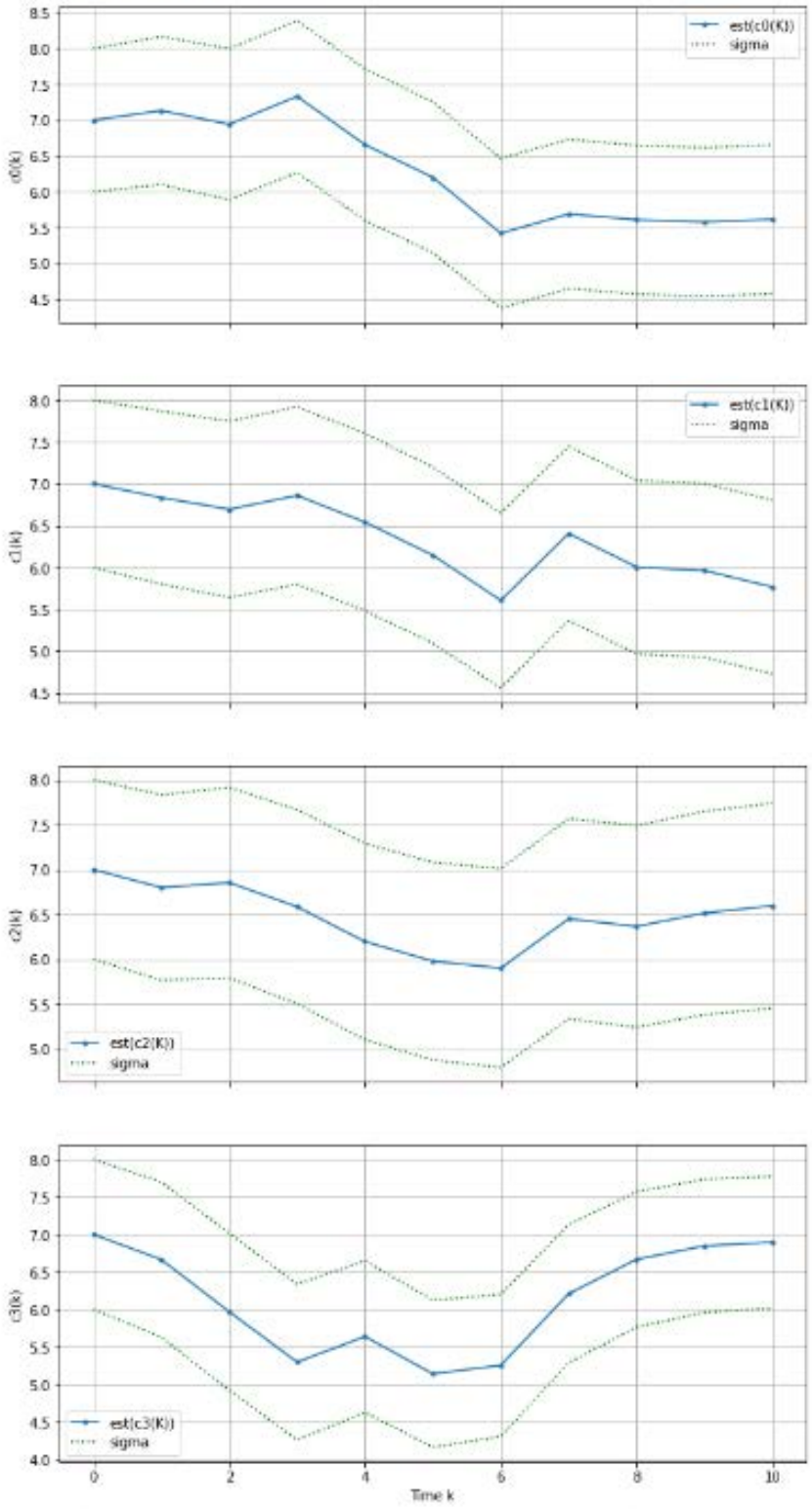
```
for k in np.arange(1,steps+1):

    Kp = dynamics @ er0 + uf
    K_uncertainty = dynamics @ sensor_uncertainty @ dynamics.T + process_uncertainty

    measur = measurs[:,k-1]

    K = K_uncertainty @ model.T @ np.linalg.inv(model @ K_uncertainty @ model.T + noise_variance)
    er0 = Kp + K @ (measur - model @ Kp)
    sensor_uncertainty = (np.eye(8) - K @ model) @ K_uncertainty @ (np.eye(8) - K @ model).T + K @ noise_variance @ K.T

    volumes[k,0] = er0[0,0]
    volumes[k,1] = er0[1,0]
    volumes[k,2] = er0[2,0]
    volumes[k,3] = er0[3,0]
    volumes[k,4] = er0[4,0]
    volumes[k,5] = er0[5,0]
    volumes[k,6] = er0[6,0]
    volumes[k,7] = er0[7,0]
    uncertainties[k,0] = sensor_uncertainty[0,0]
    uncertainties[k,1] = sensor_uncertainty[1,1]
    uncertainties[k,2] = sensor_uncertainty[2,2]
    uncertainties[k,3] = sensor_uncertainty[3,3]
    uncertainties[k,4] = sensor_uncertainty[4,4]
    uncertainties[k,5] = sensor_uncertainty[5,5]
    uncertainties[k,6] = sensor_uncertainty[6,6]
    uncertainties[k,7] = sensor_uncertainty[7,7]
```

```
fig, ax = plt.subplots(4,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(4):

    ax[n].plot(volumes[:,n],'.-',label=f"water volume ({n+1})")
    ax[n].plot(measurs_list[n],'o',label=f"measerments ({n+1})")
    ax[n].plot(volumes[:,n]+np.sqrt(uncertainties[:,n]),'g:',label=f"uncertainty ({n+1})")
    ax[n].plot(volumes[:,n]-np.sqrt(uncertainties[:,n]),'g:',)

    ax[n].set_ylabel('r1(k)')
    ax[n].legend()
    ax[n].grid(True)
```




water volume (2)
measerments (2)
uncertainty (2)
r1(k)
water volume (3)
measerments (3)
uncertainty (3)
r1(k)
water volume (4)
measerments (4)
uncertainty (4)
r1(k)

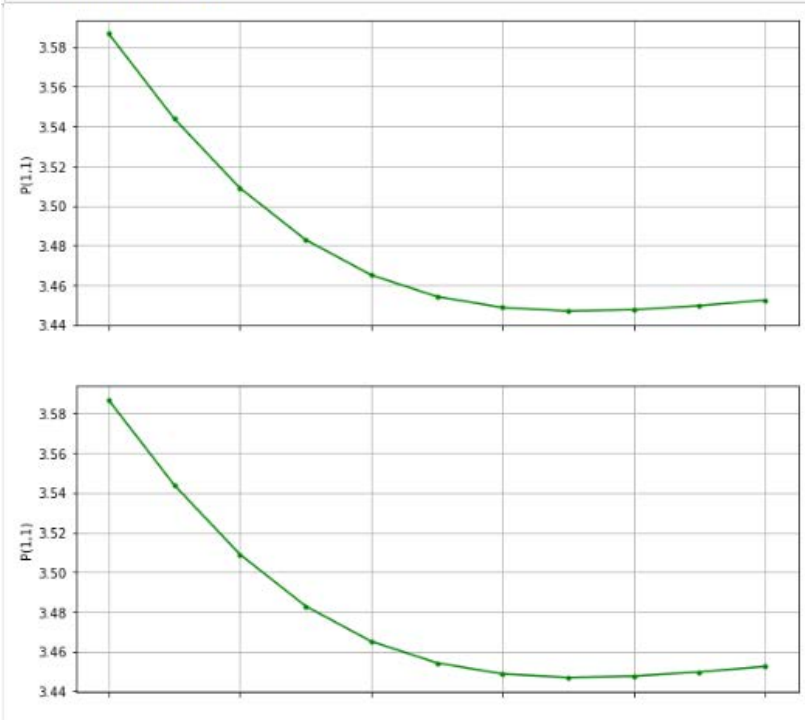
```
fig, ax = plt.subplots(4,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(4):
    ax[n].plot(volumes[:,n+4],'.-',label=f"est(c{n}(K))")
    ax[n].plot(volumes[:,n+4]+np.sqrt(uncertainties[:,n+4]),'g:',label="sigma")
    ax[n].plot(volumes[:,n+4]-np.sqrt(uncertainties[:,n+4]),'g:',)

    ax[3].set_xlabel('Time k')
    ax[n].set_ylabel(f'c{n}(k)')
    ax[n].legend()
    ax[n].grid(True)
```

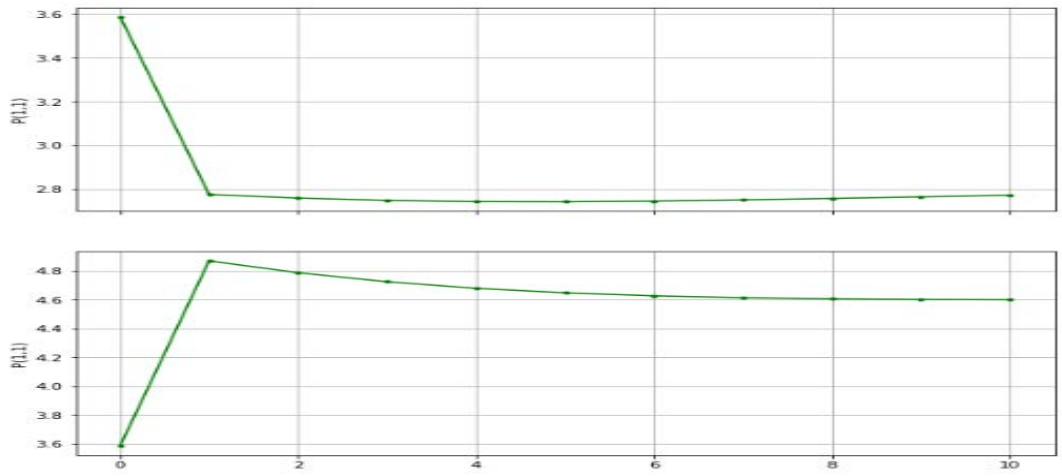


```
uncertainty_list = [np.zeros([steps+1,1]),np.zeros([steps+1,1]),np.zeros([steps+1,1]),np.zeros([steps+1,1])]
```

```
for i in range(4):

    uncertainty_list[i][0,0] = sensor_uncertainty[0,0]

for s in np.arange(1,steps+1):

    K_uncertainty = dynamics @ sensor_uncertainty @ dynamics.T + process_uncertainty

    K = K_uncertainty @ model.T @ np.linalg.inv(model @ K_uncertainty @ model.T + noise_variance)
    sensor_uncertainty = (np.eye(8) - K @ model) @ K_uncertainty @ (np.eye(8) - K @ model).T + K @ noise_variance @ K.T

    for i in range(4):
        uncertainty_list[i][s,0] = sensor_uncertainty[i,i]
```

```
fig, ax = plt.subplots(4,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(4):

    ax[n].plot(uncertainty_list[n][:,0],'g.-',label="P(1,1)")
    ax[n].set_ylabel('P(1,1)')
    ax[n].grid(True)
```



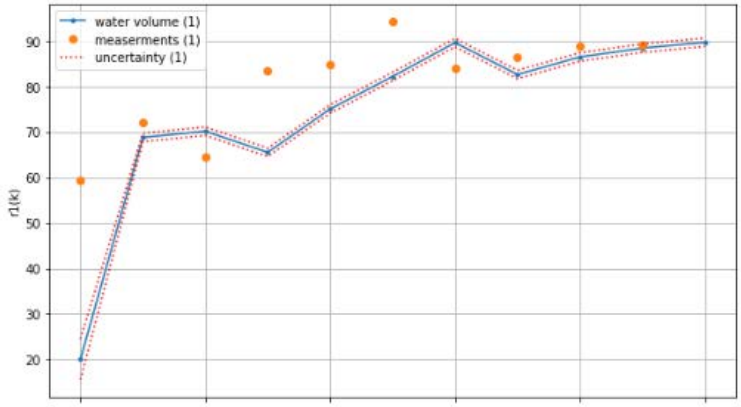


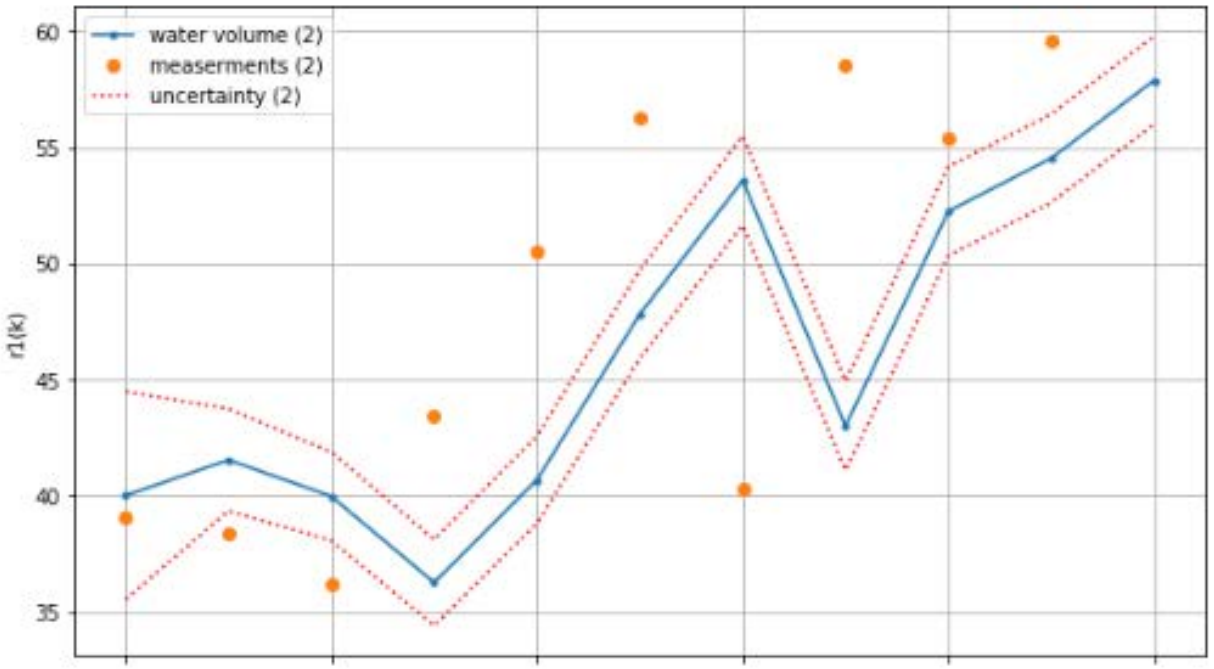
(c-iii) We will now repeat the previous problem, except that now the sensor of tank 3 has failed, and thus no longer provides any measurements. Modify your Kalman filter from before (reduce your sensor model to remove this), and run this using the same data as before (except that you remove the measurements z3(k)). Estimate the actual volume of water ri for k $\in$ {1, 2, . . . , 10} for all tanks, and also provide the associated uncertainty. How does this compare to before

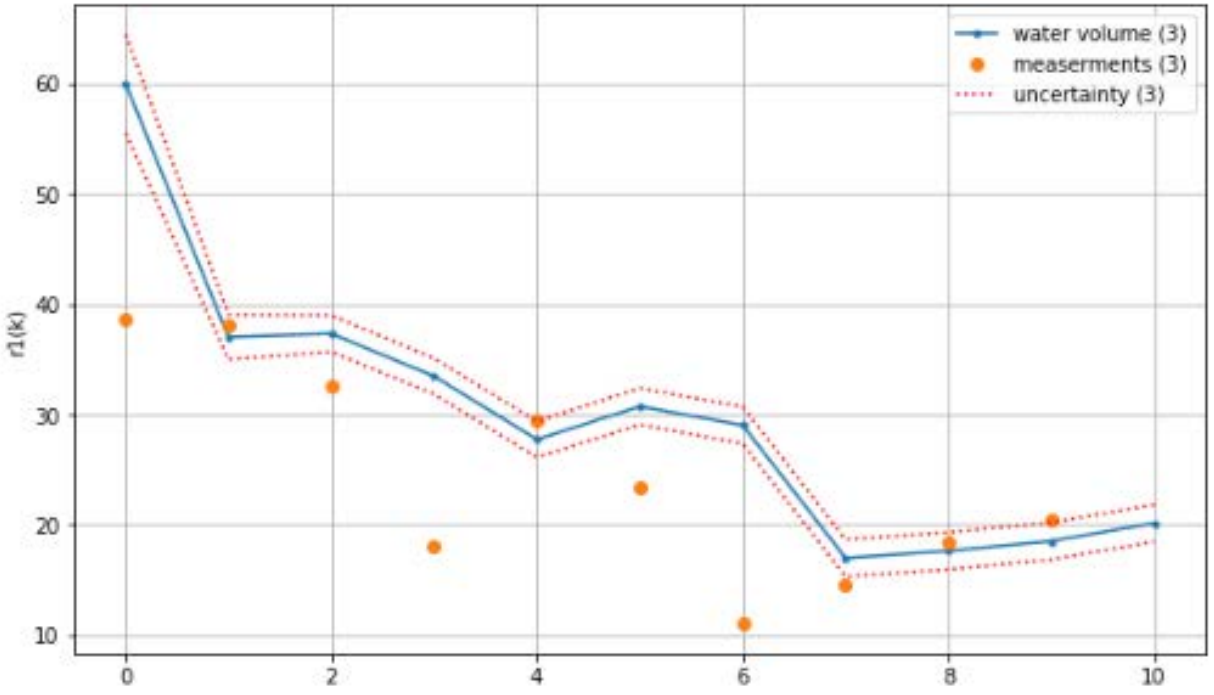
```
model = np.eye(3,8)
noise_variance = np.eye(3)*2
measurs_list = [measur1,measur2,measur4]
measurs = np.mat(measurs_list)
```

```
for k in np.arange(1,steps+1):
    Kp = dynamics @ er0 + uf
    K_unc0tainty = dynamics @ sensor_uncertainty @ dynamics.T + process_uncertainty
    measur = measurs[:,k-1]

    K = K_uncertainty @ model.T @ np.linalg.inv(model @ K_uncertainty @ model.T + noise_variance)
    er0 = Kp + K @ (measur - model @ Kp)
    sensor_uncertainty = (np.eye(8) - K @ model) @ K_uncertainty @ (np.eye(8) - K @ model).T + K @ noise_variance @ K.T

    for i in range(8):
        volumes[k,i] = er0[i,0]
        uncertainties[k,0]  = sensor_uncertainty[i,i]
```

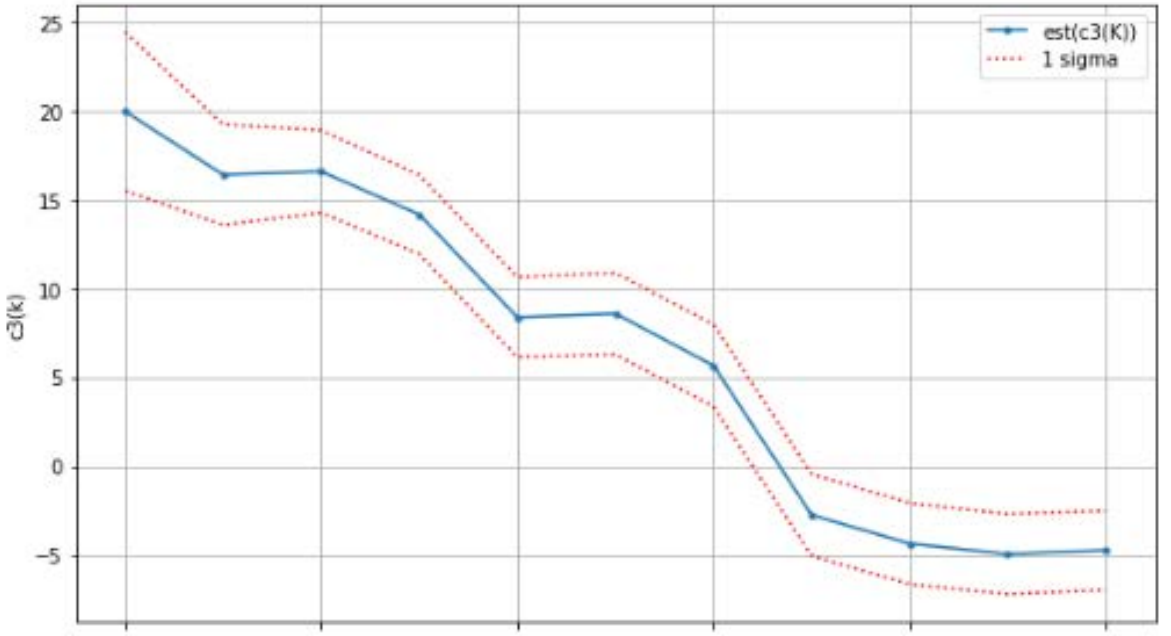
```
fig, ax = plt.subplots(3,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(3):
    ax[n].plot(volumes[:,n],'.-',label=f"water volume ({n+1})")
    ax[n].plot(measurs_list[n],'o',label=f"measerments ({n+1})")
    ax[n].plot(volumes[:,n]+np.sqrt(uncertainties[:,n]),'r:',label=f"uncertainty ({n+1})")
    ax[n].plot(volumes[:,n]-np.sqrt(uncertainties[:,n]),'r:',)

    ax[n].set_ylabel('r1(k)')
    ax[n].legend()
    ax[n].grid(True)
```

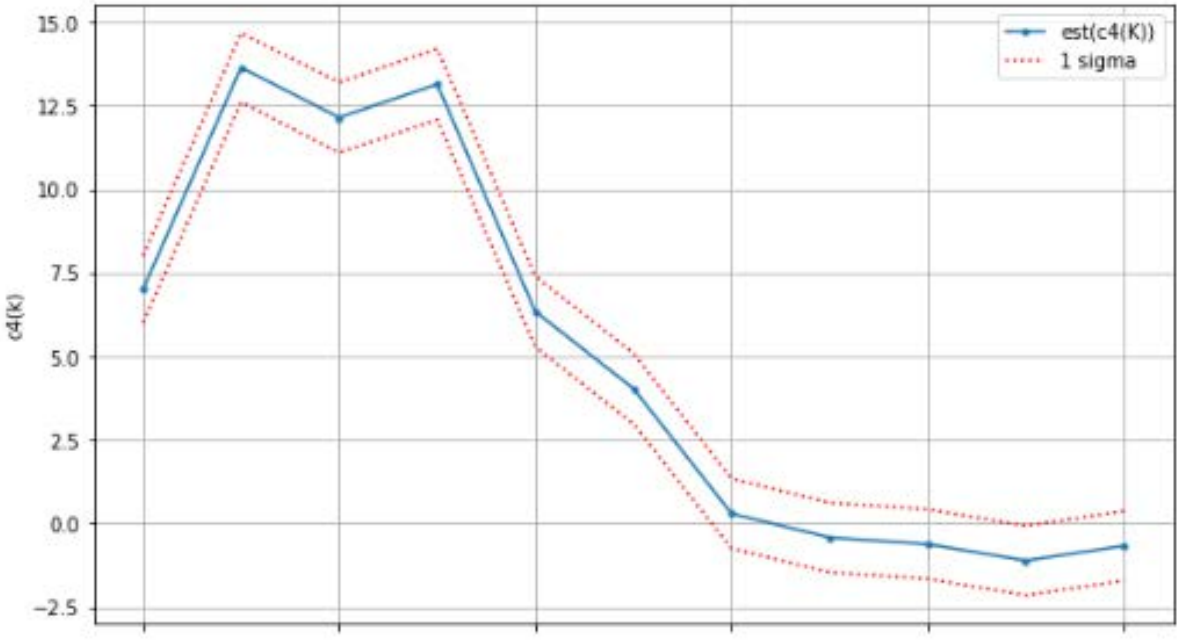


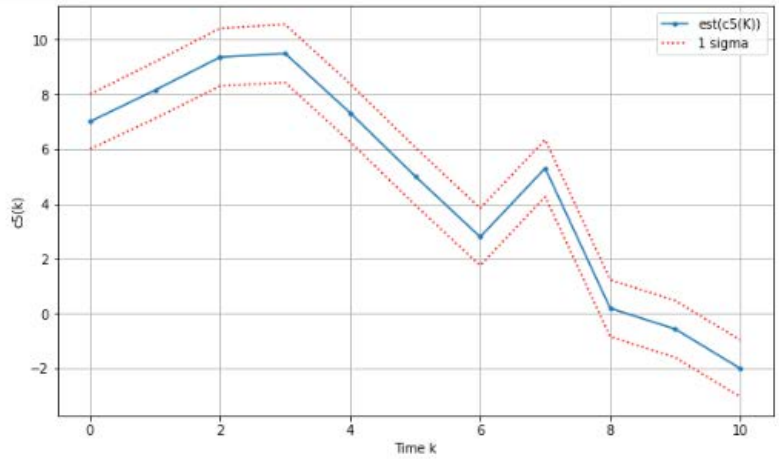

water volume (2)
measerments (2)
uncertainty (2)
r1(k)
60
55
50
45
40
35


water volume (3)
measerments (3)
uncertainty (3)
r1(k)
60
50
40
30
20
10
0
2
4
6
8
10

```
fig, ax = plt.subplots(3,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(3):

    ax[n].plot(volumes[:,n+3],'.-',label=f"est(c{n+3}(K))")
    ax[n].plot(volumes[:,n+3]+np.sqrt(uncertainties[:,n+3]),'r:',label="1 sigma")
    ax[n].plot(volumes[:,n+3]-np.sqrt(uncertainties[:,n+3]),'r:',)

    ax[2].set_xlabel('Time k')
    ax[n].set_ylabel(f'c{n+3}(k)')
    ax[n].legend()
    ax[n].grid(True)
```



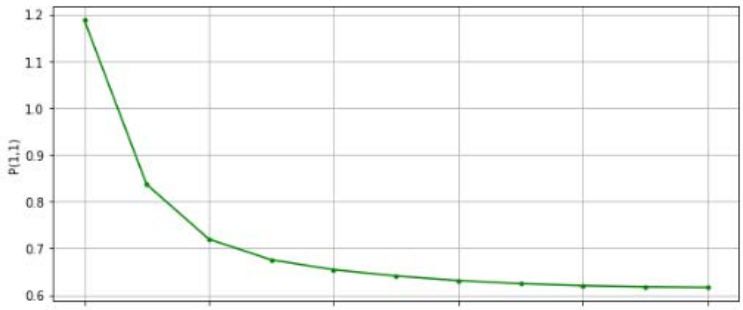


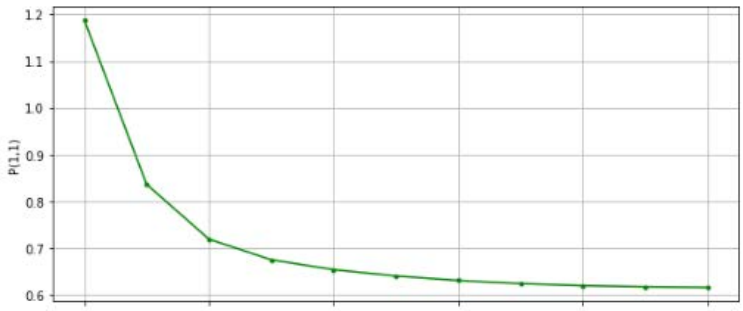


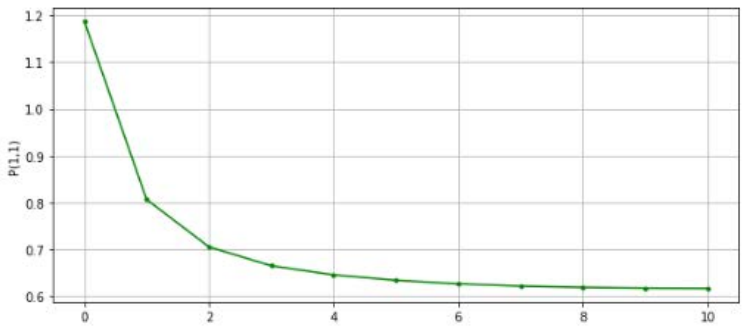
3]:
```
uncertainties = [np.zeros([steps+1,1]),np.zeros([steps+1,1]),np.zeros([steps+1,1])]
```

4]:
```
for i in range(3):
    uncertainties[i][0,0] = sensor_uncertainty[0,0]

for s in np.arange(1,steps+1):

    K_uncertainty = dynamics @ sensor_uncertainty @ dynamics.T + process_uncertainty

    K = K_uncertainty @ model.T @ np.linalg.inv(model @ K_uncertainty @ model.T + noise_variance)
    sensor_uncertainty = (np.eye(8) - K @ model) @ K_uncertainty @ (np.eye(8) - K @ model).T + K @ noise_variance @ K.T

    for i in range(3):
        uncertainties[i][s,0] = sensor_uncertainty[i,i]
```

```
fig, ax = plt.subplots(3,1,sharex=True)
fig.set_size_inches(10,15)

for n in range(3):
    ax[n].plot(uncertainties[n][:,0],'g.-')

    ax[n].set_ylabel('P(1,1)')
    ax[n].grid(True)
```







Problem 5

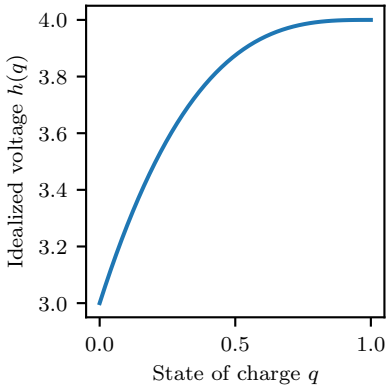
Optional, extra credit. For the battery system of Problem 3, we now add a voltage sensor which gives us a noisy reading of the battery voltage after each time cycle. The measurement is $z(k) = h(q(k)) + w(k)$, with $w(k) \sim \mathcal{N}\left(0, \sigma_w^2\right)$, and where $h(q)$ is a nonlinear function mapping from current state of charge to voltage, plotted below, and with

$$h(q) = 4 + (q-1)^3.$$

The battery is subject to the same discharging process as before, with

$$q(k) = q(k\text{–}1) - j(k\text{–}1)$$

where $q(0) = 1$ with no uncertainty; and $j(k) = j_0 + v(k)$ with $v(k)$ zero mean with variance σ_v^2



For our system, we set $j_0 = 0.1$, $\sigma_v = 0.05$, and $\sigma_w = 0.1$.

a. Design an extended Kalman filter (EKF) to estimate the state of charge.

We note that the amount of information that the EKF gets from the measurement is determined by the function h, and specifically its slope with respect to the state, $H = \frac{\partial h}{\partial q}$. Because our system and measurements are scalar, all quantities are also scalar. We can thus reason about how useful a measurement is, by comparing the state variance after getting the measurement P_m to the variance before the measurement, P_p. Note that we're neglecting the time index (k), as all quantities are at the same time step.

b. Show that the reduction in variance due to a measurement (i.e. a measure of its usefulness) can be described as below:

$$\frac{P_p - P_m}{P_p} = \frac{H^2 P_p}{\sigma_w^2 + H^2 P_p}$$

where a value of 1 means that the measurement removed all uncertainty, and a value of 0 means that the measurement made no difference to the uncertainty.

c. Using this metric, make a plot of the usefulness of the voltage measurement as a function of the estimated state of charge, for $\hat{q} \in [0, 1]$ (with usefulness as defined in the subproblem b). Set $P_p = 0.1$, and $\sigma_w^2 = 0.1$. Where is the measurement most informative? Where is it least informative?

For the remainder of the problem, given is the following sequence of measurements:

time k	1	2	3	4	5	6	7	8	9
measurement $z(k)$	4.21	3.83	3.92	3.89	3.88	3.89	3.91	3.57	3.21

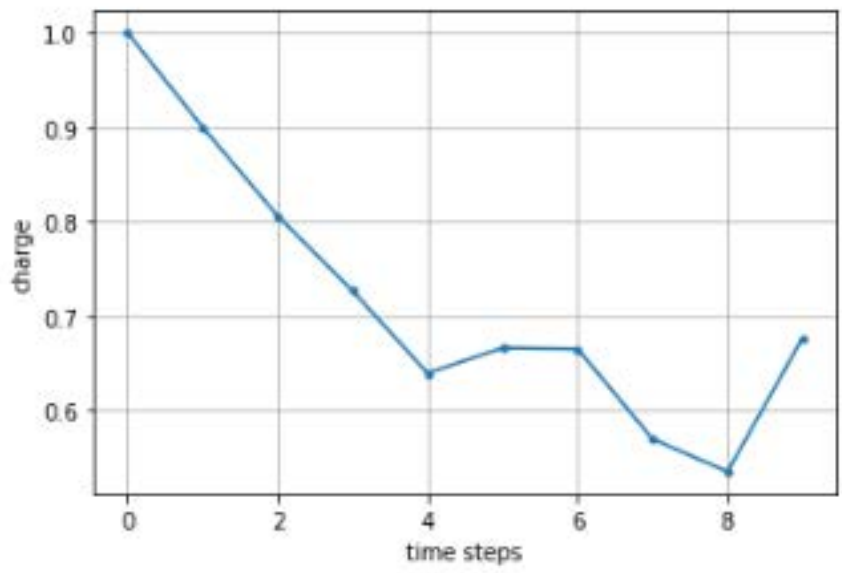
d. Run your extended Kalman filter with this data, and generate two plots: the estimated state of charge, and the variance of this estimate, across k.

e. After 9 steps, what would the mean and variance be if you did not have the voltage measurements (use your results from Problem 3). How does this compare to your EKF output?

```
steps = 9
dynamic = 1
sigma_v2 = 0.05**2
sigma_w2 = 0.1**2
q = 1
Vr0 = 0
real_state = np.random.normal(1,0)
charges = np.zeros([steps+1,1])
uncertainties = np.zeros([steps+1,1])
measurments = np.zeros([steps+1,1])
charges[0,0] = q
uncertainties[0,0] = Vr0
```

```
for k in np.arange(1,steps+1):

    noise = np.random.normal(0, 0.05)
    real_state -= noise - 0.1
    W = np.random.normal(0, 0.1)

    measurments[k,0] = 4 + ((real_state)-1)**3 + W
    Kp = q - 0.1
    K_uncertainty = dynamic*Vr0*dynamic + sigma_v2
    H = 3*(Kp-1)**2
    K = K_uncertainty*H*1/(H*K_uncertainty*H + sigma_w2)
    means = 4 + ((Kp)-1)**3
    q = Kp + K*(measurments[k,0]-means)
    Vr0 = (1 - K*H)*K_uncertainty*(1 - K*H) + K*sigma_w2*K

    charges[k,0] = q
    uncertainties[k,0] = Vr0
```

```
plt.plot(charges[:,0],'.-')
plt.xlabel('time steps')
plt.ylabel('charge')
plt.grid(True)
```

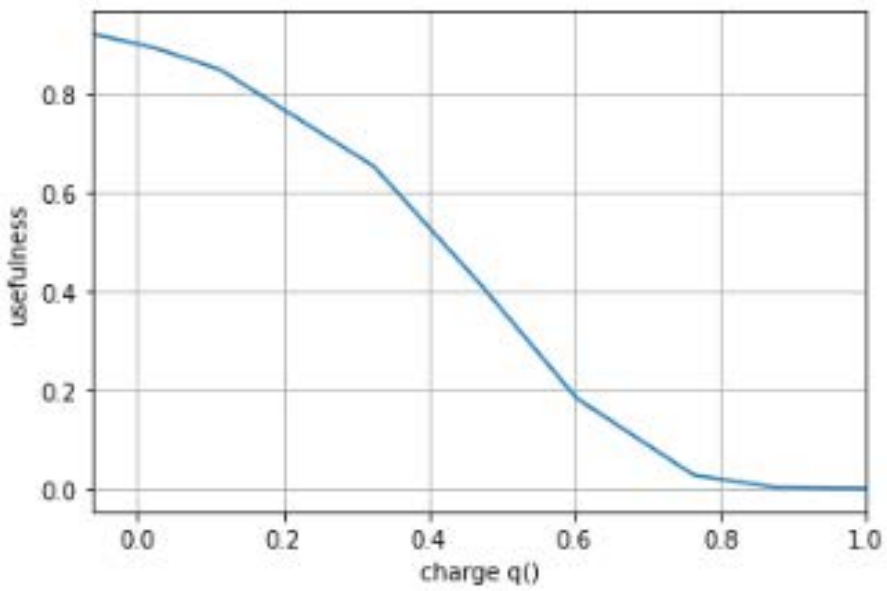


```
q = 1
sigma_w = u = 0.1

time_steps = np.arange(steps+1)
Xs = np.zeros(steps+1)
Hs = np.zeros(steps+1)
Us = np.zeros(steps+1)

Xs[0] = q
Hs[0] = (q-1)**2
Us[0]=(Hs[0]**2*0.1)/(Hs[0]**2*0.1+sigma_w)
```

```
def q(x, u, v):
    return  x - u - v
```

```
for k in range(steps):
        v = np.random.normal(0,0.05)
        Xs[k+1] = q(Xs[k], u, v)
        Hs[k+1] = 3*(Xs[k+1]-1)**2
        Us[k+1]=(Hs[k+1]**2*0.1)/(Hs[k+1]**2*0.1+sigma_w)
```

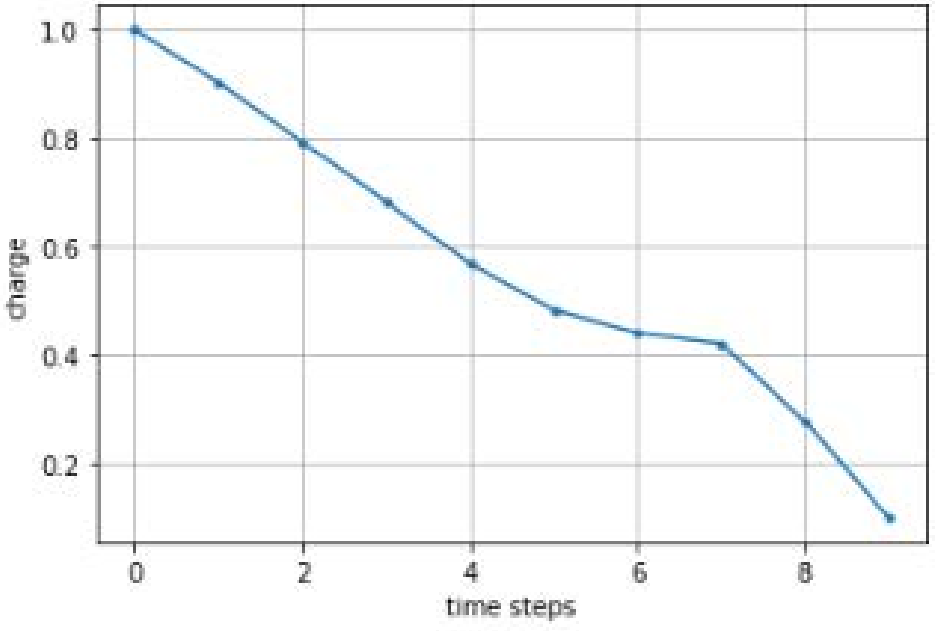
```
plt.plot(Xs, Us)
plt.xlim([Xs[-1], Xs[0]])
plt.xlabel('charge q()')
plt.ylabel('usefulness')
plt.grid(True)
```




```
q = 1
uncertainty_Vr0 = 0
states = np.zeros([steps+1,1])
true_state = np.random.normal(1,0)
states[0,0] = true_state
measurs = [4.21, 3.83, 3.92, 3.89, 3.88, 3.89, 3.91, 3.57, 3.21]
```

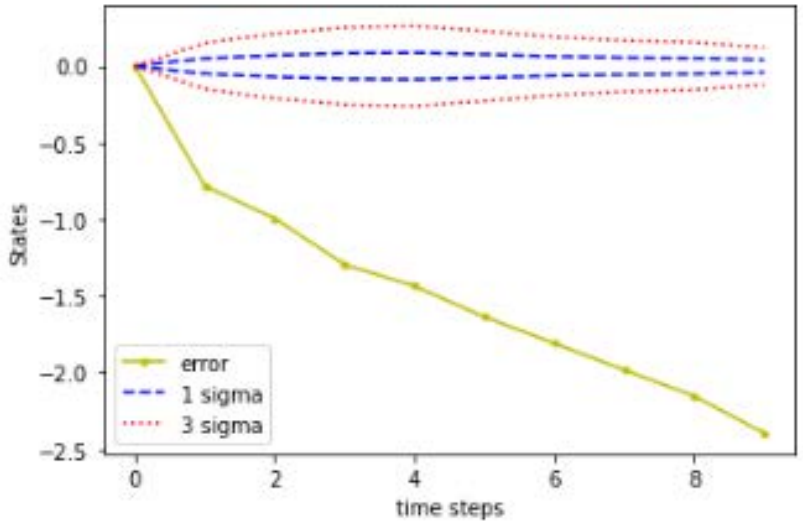
```
for k in np.arange(1,steps+1):

    noise = np.random.normal(0, 0.05)

    real_state -= noise - 0.1

    w = np.random.normal(0, 0.1)

    measurments[k,0] = measurs[k-1]

    Kp = q - 0.1
    K_uncertainty = dynamic*uncertainty_Vr0*dynamic + sigma_v2

    H = 3*(Kp-1)**2
    K = K_uncertainty*H*1/(H*K_uncertainty*H + sigma_w2)

    AVGs = 4 + ((Kp)-1)**3

    q = Kp + K*(measurments[k,0]-AVGs)
    uncertainty_Vr0 = (1 - K*H)*K_uncertainty*(1 - K*H) + K*sigma_w2*K

    charges[k,0] = q
    uncertainties[k,0] = uncertainty_Vr0
    states[k,0] = real_state
```

```
plt.plot(charges[:,0],'.-')
plt.xlabel('time steps')
plt.ylabel('charge')
plt.grid(True)
```

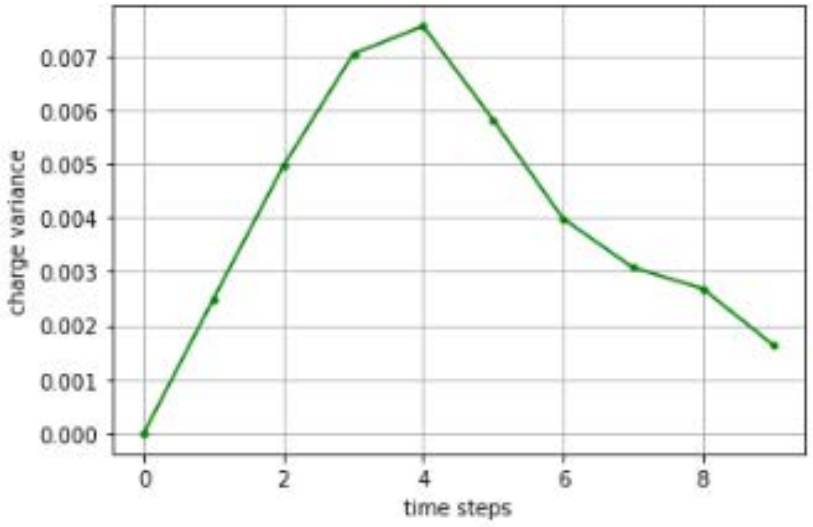


```
plt.plot(charges[:,0]-states[:,0],'y.-',label="error")
plt.plot(np.sqrt(uncertainties[:,0]),'b--',label="1 sigma")
plt.plot(-np.sqrt(uncertainties[:,0]),'b--',)
plt.plot(3*np.sqrt(uncertainties[:,0]),'r:',label="3 sigma")
plt.plot(-3*np.sqrt(uncertainties[:,0]),'r:',)
plt.xlabel('time steps')
plt.ylabel('States')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1aec7a1feb0>
```




```
plt.plot(uncertainties[:,0],'g.-')
plt.xlabel('time steps')
plt.ylabel('charge variance')
plt.grid(True)
```




```
j=0.1
sigma=0.05
avg=1-9*j
var=9*sigma**2
print(f'the mean with no voltage measurements is {round(avg,2)}')
print(f'the variance with no voltage measurements is {round(var,2)}')
```

```
the mean with no voltage measurements is 0.1
the variance with no voltage measurements is 0.02
```

```
plt.plot(measurments[:,0],label="z")
plt.plot([0] + measurments,'o',label="z_m")

plt.xlabel('time steps')
plt.ylabel('measurements')
plt.legend();
```