State Estimation,Autonomy,Machine Learning &Energy System

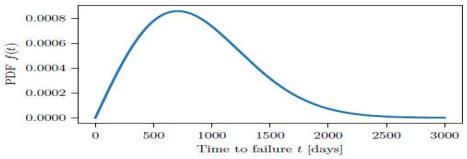
Capstone Report for Course 4

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:



```
t = symbols('t')
Lambda = symbols('λ')

pdf = 2*t*exp((-(t**2))/(Lambda**2))/Lambda**2

pdf
```

✓ 0.1s

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

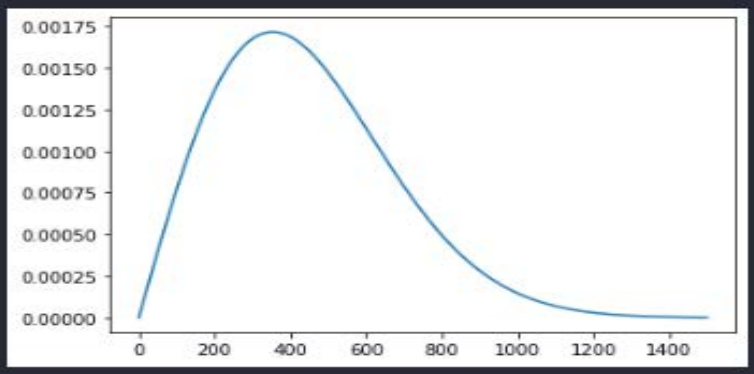
```
# since λ = 500 days.
Lambda = 500

pdf = 2*t*exp((-(t**2))/(Lambda**2))/Lambda**2
pdf
```

✓ 0.1s

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

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

array = np.array([])
for t_ in range(1500):
    array = np.append(array,PDF(t_))

plt.plot(array);
```

✓ 0.3s



A- Is it a valid PDF?

Are there non-negatives anywhere?

```
print(f'N nigatives = {(array<0).sum()}')
```

✓ 0.1s

```
N nigatives = 0
```

Is the area under the curve = 1?

```
print("AUC =", integrate(pdf, [t,0,oo]))
```

✓ 0.1s

```
AUC = 1
```

B- 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 time_after_installation(p0):

    for i in range(len(array)):

        n = array[:i]

        if np.trapz(n) >= p0:

            return i-1

days = []

for p in [0.01,0.1,0.5,0.99]:

    ndays = time_after_installation(p)
    days.append(ndays)
    print(f'For a less than {int(p*100)}% prbability of unexpected failure, you should preventate maintenance in day number {
```

✓ 0.1s Python

```
For a less than 1% prbability of unexpected failure, you should preventate maintenance in day number 51
For a less than 10% prbability of unexpected failure, you should preventate maintenance in day number 163
For a less than 50% prbability of unexpected failure, you should preventate maintenance in day number 417
For a less than 99% prbability of unexpected failure, you should preventate maintenance in day number 1073
```

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

```
expected_lifetime = integrate(pdf*t, [t,0,oo])
pf = integrate(pdf,[t,0,expected_lifetime])

print(f'Expected lifetime: {round(expected_lifetime)} days')
print(f'Probability of failure before the expected lifetime: {round(pf,4)*100}%')
```

✓ 0.2s

```
Expected lifetime: 443 days
Probability of failure before the expected lifetime: 54.41%
```

D- 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 - expected_lifetime )**2) , [t,0, oo])
print(f'Variance of the pump's lifetime: {round(variance,2)}')
print(f'Standard deviation of the pump's lifetime: {round(sqrt(variance),2)}')
```

✓ 0.2s

```
Variance of the pump's lifetime: 53650.46
Standard deviation of the pump's lifetime: 231.63
```

E- Write a program that generates samples of t from its distribution.

```
def average(Tm ,n_samples = 10**6, Cr = 250, Cm= 50):

    runnig_cost = np.zeros((n_samples,))
    samples = np.zeros((n_samples,))

    for i in range(n_samples):

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

        if samples[i] <= Tm:
            runnig_cost[i] = Cr/samples[i]
        else:
            runnig_cost[i] = Cm/Tm

    avg_R = round(np.mean(runnig_cost),4)
    avg_smpl = round(np.mean(samples),4)
    var_smpl = round(np.var(samples),4)

    print(f'For Tm = {Tm}, the average cost was: {avg_R}$,')
    print(f'sample average was: {avg_smpl}')
    print(f'and sample variance was: {var_smpl}')
```

✓ 0.1s

```
for tm in [1,10,100,1000,10000]:
    average(tm)
    print("="*40)
```

✓ 26.9s

```
For Tm = 1, the average cost was: 54.2265$,
sample average was: 1000.7759
and sample variance was: 1000859.0303
========================================
For Tm = 10, the average cost was: 7.5091$,
sample average was: 999.9387
and sample variance was: 998138.8593
========================================
For Tm = 100, the average cost was: 3.1945$,
sample average was: 998.5204
and sample variance was: 994108.0131
========================================
For Tm = 1000, the average cost was: 3.8163$,
sample average was: 998.6611
and sample variance was: 999264.7262
========================================
For Tm = 10000, the average cost was: 4.2852$,
sample average was: 1000.0124
and sample variance was: 1001717.4318
========================================
```

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 =

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

```python
def transformer_prob_of_out_of_spec(n_transformers, prob_of_out_of_spec,p):

    return prob_of_out_of_spec*(n_transformers/33)/p
```

0.1s Python

```python
p = (0.002*8 + 0.003*5 + 0.004*20)/33
m1 = transformer_prob_of_out_of_spec(8, 2/1000,p)
m2 = transformer_prob_of_out_of_spec(5, 3/1000,p)
m3 = transformer_prob_of_out_of_spec(20, 4/1000,p)
```

0.1s Python

```python
import pandas as pd
import plotly.express as px

ms = np.array([m1,m2,m3])
labels = ["manufacturer 1", "manufacturer 2", "manufacturer 3"]

d = pd.DataFrame(ms,index=labels)
px.pie(d,names=d.index,values=0)
```

0.2s Python

There's a probability that it's from manufacturer 1 of 14.4%

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.

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

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

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

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

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

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

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

```
j0 = 0.1
sigma = 0.05

K1 = (sigma*3 + (sigma**2*9 +j0*4)**-2/-j0*2)**2
K2 = (sigma*3 - (sigma**2*9 +j0*4)**-2/-j0*2)**2

print(f'Case one requires {round(K1,2)} secound')
print(f'Case two requires {round(K2,2)} secound')
```

✓ 0.1s

```
Case one requires 12519.57 secound
Case two requires 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.

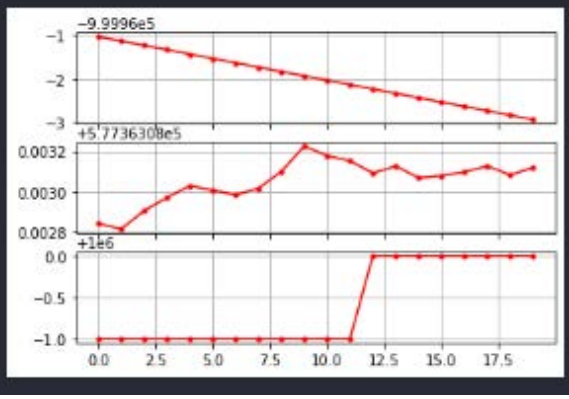
- To Simulate 10^6 different evolutions, compute their mean & standard deviation across $k \in \{0, 1, .. , 20\}$.

```
K0 = 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] = K0
        v = np.random.normal(0,sigma)
        K0 -= j0-v
```

✓ 46.2s

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

for n,m in enumerate([np.mean(Ks,0),np.std(Ks,0),np.sum(Ks<0,0)]):

    ax[n].plot(times,m,'r.-',label=f"volume of water ({n+1})")
    ax[n].grid()
```

✓ 1.1s

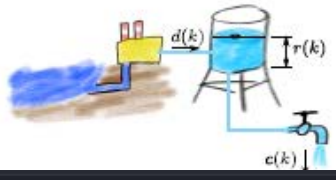


Problem 4

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.




```
n_steps = 10 # 10 steps
Er0 = 20 # E[r(0)] = 20
Vr0 = 25 # Var[r(0)] = 25
dk = 10 # d(k) = 10
m = 7 # predicted supply m = 7
uf = dk - m
Vvk = 9 # Var [v(k)] = 9.
Vwk = 25 # Var [w(k)] = 25.
```
✓ 0.1s

```
measurements = [17.8, 22.6, 30.2, 37.3, 46.2, 49.5, 44.6, 50.3, 56.3, 51.6]

water_volume = np.zeros([n_steps+1]) # to store the actual volume of water.
water_volume[0] = Er0 # since E[r(0)] = 20 at time k = 0.

uncertainty = np.zeros([n_steps+1]) # to provide the associated uncertainty for my estimate.
uncertainty[0] = Vr0 # since Var[r(0)] = 25 at time k = 0.
```
✓ 0.1s

```
dynamic = np.mat([[1]])

measurement_model = np.mat([[1]])

measurement_noise_variance = np.mat([[25]])
```
✓ ✓ ✓ 0.1s

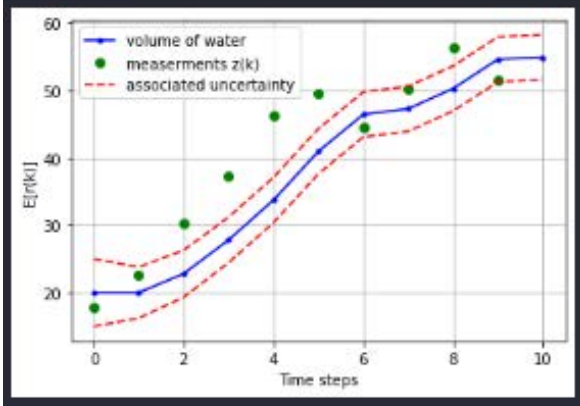
```
for k in range(1,n_steps+1):

    Kp = uf + dynamic*Er0
    K_uncertainty = Vwk * (dynamic**2) + Vvk

    measurement = measurements[k-1]

    K = K_uncertainty @ measurement_model.T @ np.linalg.inv(measurement_model @ K_uncertainty @ measurement_model.T + measurement_noise_variance)
    Er0 = Kp + K @ (measurement - measurement_model @ Kp)
    Vwk = (np.eye(1) - K @ measurement_model) @ K_uncertainty @ (np.eye(1) - K @ measurement_model).T + K @ measurement_noise_variance @ K.T

    water_volume[k] = Er0[0]

    uncertainty[k] = Vwk[0]
```
✓ ✓ ✓ 0.2s

+ Code + Markdown

(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.

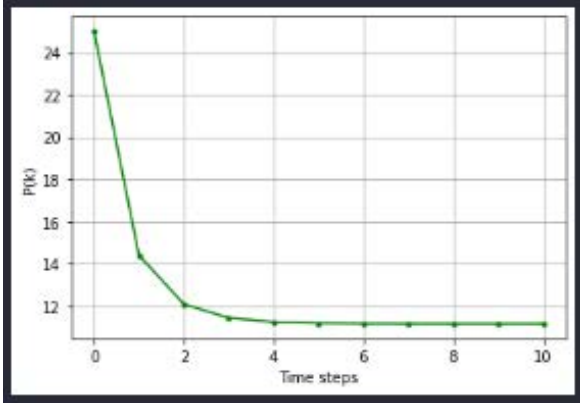
```
plt.plot(water_volume,'b.-',label="volume of water")
plt.plot(measurements,'go',label="measerments z(k)")
plt.plot(water_volume+np.sqrt(uncertainty),'r--',label="associated uncertainty")
plt.plot(water_volume-np.sqrt(uncertainty),'r--')
plt.ylabel('E[r(k)]')
plt.xlabel('Time steps')
plt.legend()
plt.grid(True)
```

✓ ✓ 0.4s




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

✓ ✓ 0.4s



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]])

measurement_model = np.mat([[1,0]])

measurement_noise_variance = np.mat([[25]])

ii = np.eye(2)
```
✓ ✓ 0.1s

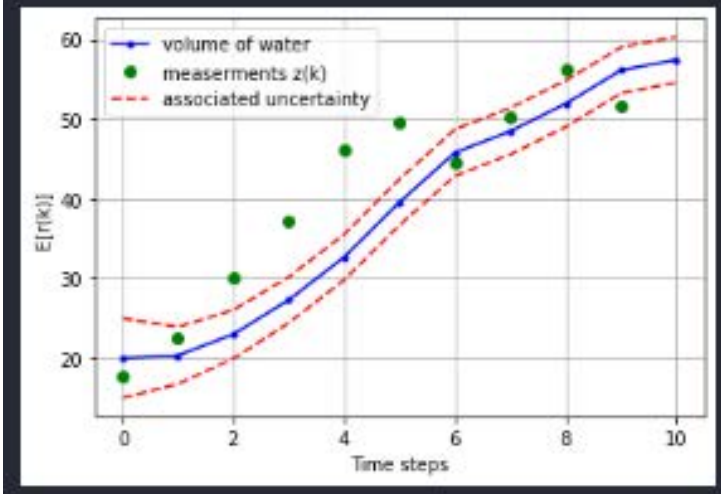
```
Er0 = np.mat([[20],[7]]) # E[r(0)] = 20, E[c(0)] = 7
uf = np.mat([[10],[0]])
process_uncertainty = np.mat([[0,0],[0,0.1]]) #  Var [n(k)] = 0.1
sensor_uncertainty = np.mat([[25,0],[0,1]]) # Var [w(k)] = 25, Var [c(0)] = 1
```
✓ ✓ ✓ 0.7s

```
water_volume = np.zeros([n_steps+1,2])

water_volume[0,0] = Er0[0,0]
water_volume[0,1] = Er0[1,0]

uncertainty = np.zeros([n_steps+1,2])

uncertainty[0,0] = sensor_uncertainty[0,0]
uncertainty[0,1] = sensor_uncertainty[1,1]
```
✓ ✓ 0.1s

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

    # Kalman filter prediction:
    Kp = dynamics @ Er0 + uf
    # Kalman filter predynamicsiction uncertainty:
    K_uncertainty = dynamics @ sensor_uncertainty @ dynamics.T + process_uncertainty

    measurement = measurements[k-1]

    K = K_uncertainty @ measurement_model.T @ np.linalg.inv(measurement_model @ K_uncertainty @ measurement_model.T + measurement_noise_variance)
    Er0 = Kp + K @ (measurement - measurement_model @ Kp)
    sensor_uncertainty = (ii-K @ measurement_model)@ K_uncertainty @(ii-K @ measurement_model).T + K @ measurement_noise_variance @ K.T

    # store the variables for plotting:
    water_volume[k,0] = Er0[0,0]
    water_volume[k,1] = Er0[1,0]

    uncertainty[k,0] = sensor_uncertainty[0,0]
    uncertainty[k,1] = sensor_uncertainty[1,1]
```
✓ 0.2s

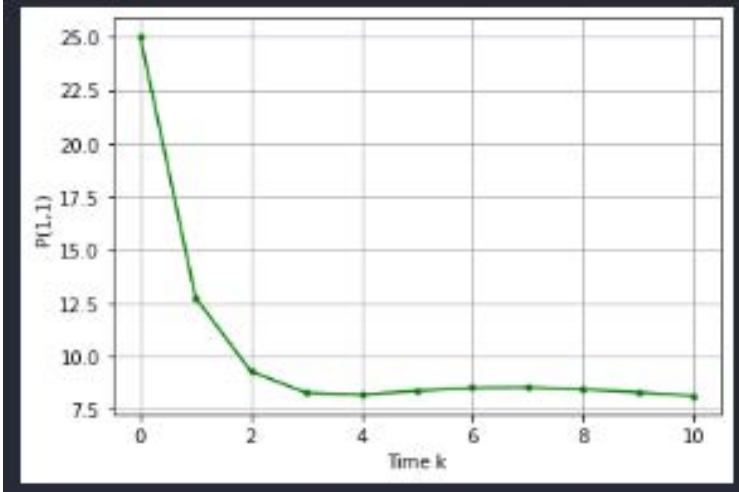
```
plt.plot(water_volume[:,0],'b.-',label="volume of water")
plt.plot(measurements,'go',label="measerments z(k)")
plt.plot(water_volume[:,0]+np.sqrt(uncertainty[:,0]),'r--',label="associated uncertainty")
plt.plot(water_volume[:,0]-np.sqrt(uncertainty[:,0]),'r--')
plt.ylabel('E[r(k)]')
plt.xlabel('Time steps')
plt.legend()
plt.grid(True)
```

✓ 0.3s



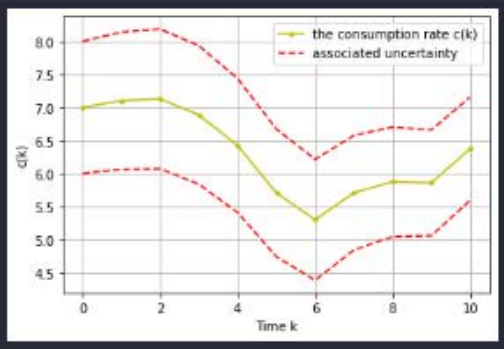

```
plt.plot(uncertainty[:,0],'g.-',label="P(k)(1,1)")
plt.xlabel('Time k')
plt.ylabel('P(1,1)')
plt.grid(True)
```

✓ 0.4s



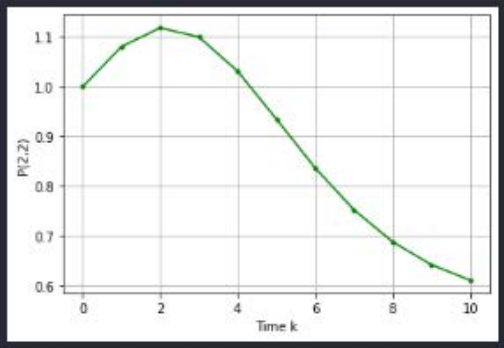
```
plt.plot(water_volume[:,1],'y.-',label="the consumption rate c(k)")
plt.plot(water_volume[:,1]+np.sqrt(uncertainty[:,1]),'r--',label="associated uncertainty")
plt.plot(water_volume[:,1]-np.sqrt(uncertainty[:,1]),'r--')

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

✓ 0.4s




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

✓ 0.4s



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:

```python
a = 0.3
dynamics = np.matrix([[1-2*a, a, a, 0, -1, 0, 0, 0],
                      [a, 1-2*a, a, 0, 0, -1, 0, 0],
                      [a, a, 1-3*a, a, 0, 0, -1, 0],
                      [0, 0, a, 1-a, 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])

measurement_model = np.eye(4,8)

measurement_noise_variance = np.eye(4)*25
```

✓ ✓ 0.1s Python

```python
z1 = np.array([59.3, 72, 64.4, 83.6, 84.9, 94.3, 84, 86.6, 89, 89.1])
z2 = np.array([39.1, 38.4, 36.2, 43.4, 50.5, 56.3, 40.3, 58.5, 55.4, 59.6])
z3 = np.array([31.1, 31.2, 41.6, 44.4, 41, 41.9, 39.2, 46.3, 43.3, 45.3])
z4 = np.array([38.6, 38, 32.6, 18, 29.4, 23.3, 11, 14.6, 18.4, 20.5])

list_of_measurements = [z1,z2,z3,z4]
measurements = np.mat(list_of_measurements)
```

✓ 0.1s Python

```python
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]]
```

✓ ✓ 0.1s Python

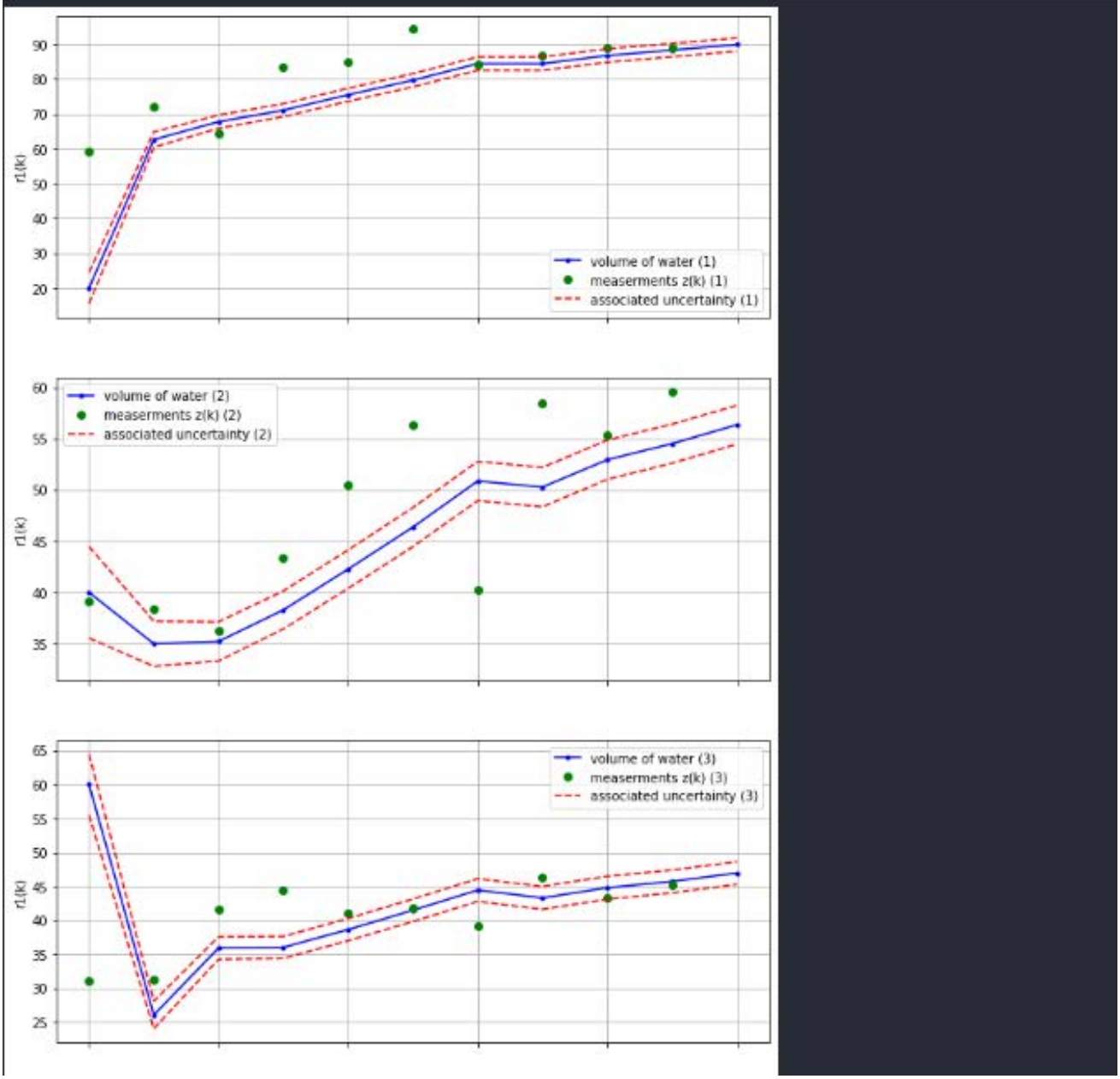
```python
water_volume = np.zeros([n_steps+1,8])

uncertainty = np.zeros([n_steps+1,8])

for i in range(8):
    water_volume[0,i] = Er0[i,0]
    uncertainty[0,i] = sensor_uncertainty[i,i]
```

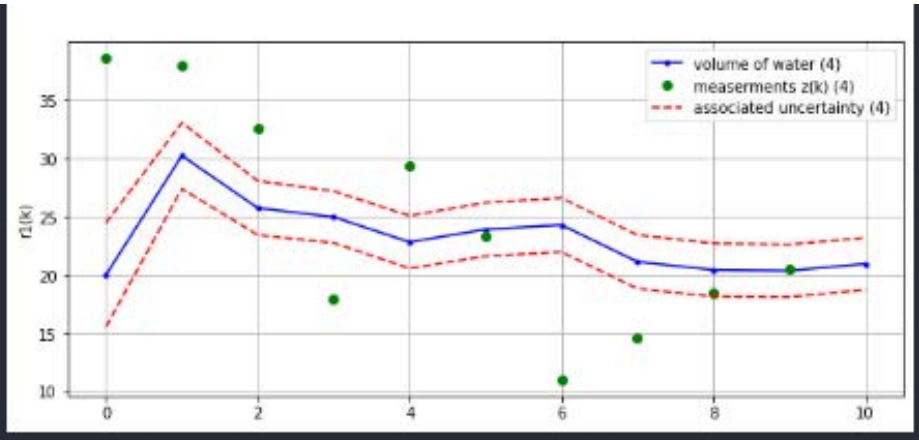
✓ ✓ 0.1s Python

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

    Kp = dynamics @ Er0 + uf
    K_uncertainty = d @ sensor_uncertainty @ d.T + process_uncertainty

    measurement = measurements[:,k-1]

    K = K_uncertainty @ measurement_model.T @ np.linalg.inv(measurement_model @ K_uncertainty @ measurement_model.T + measurement_noise_variance)
    Er0 = Kp + K @ (measurement - measurement_model @ Kp)
    sensor_uncertainty = (np.eye(8) - K @ measurement_model) @ K_uncertainty @ (np.eye(8) - K @ measurement_model).T + K @ measurement_noise_variance @ K.

    for i in range(8):
        water_volume[k,i] = Er0[i,0]
        uncertainty[k,i] = sensor_uncertainty[i,i]
```

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

for n in range(4):

    ax[n].plot(water_volume[:,n],'b.-',label=f"volume of water ({n+1})")
    ax[n].plot(list_of_measurements[n],'go',label=f"measerments z(k) ({n+1})")
    ax[n].plot(water_volume[:,n]+np.sqrt(uncertainty[:,n]),'r--',label=f"associated uncertainty ({n+1})")
    ax[n].plot(water_volume[:,n]-np.sqrt(uncertainty[:,n]),'r--',)

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

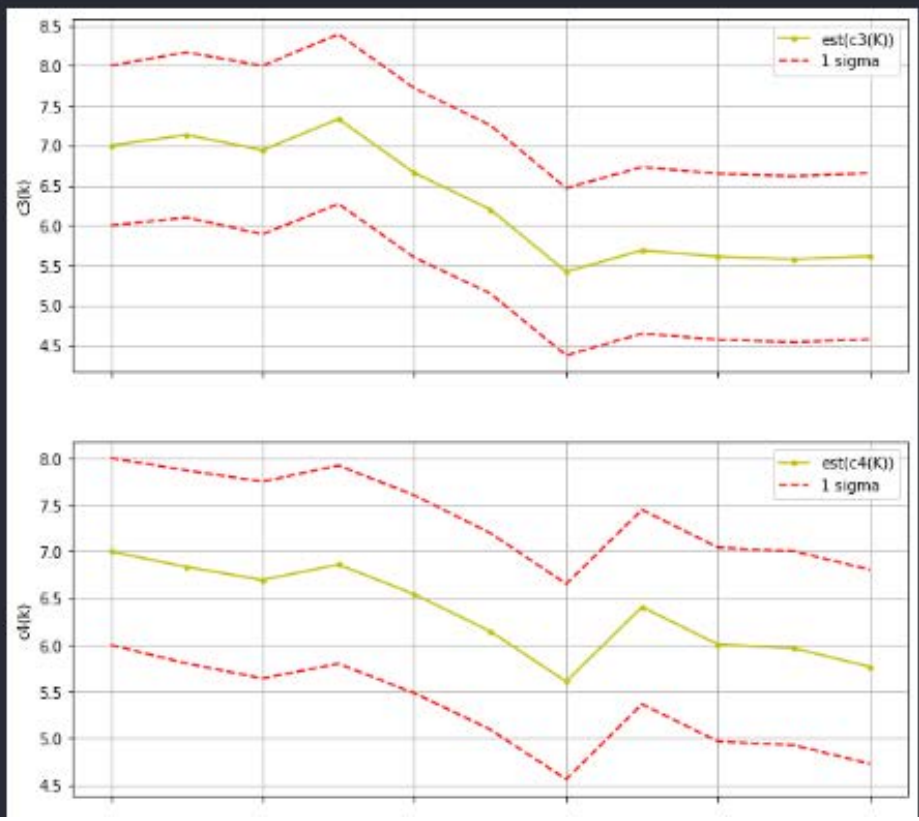
✓ 0.8s

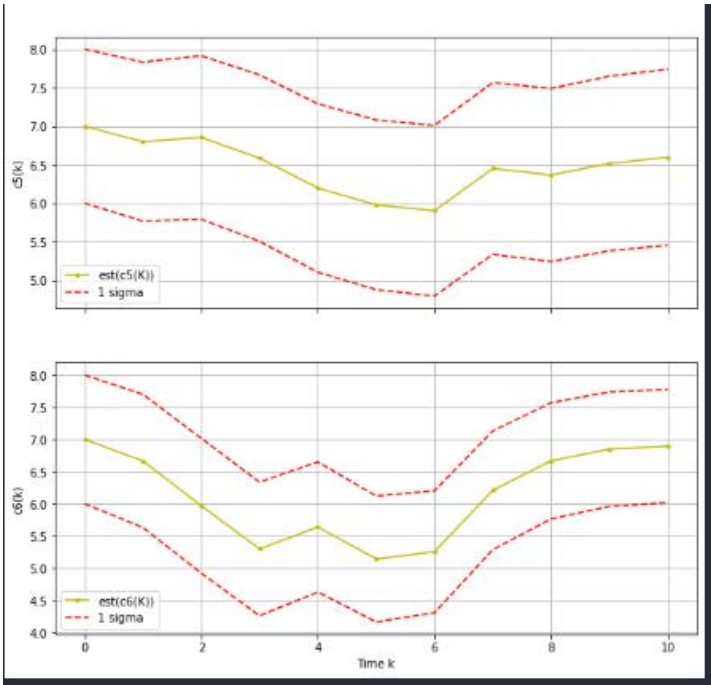





```
fig, ax = plt.subplots(4,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(4):

    ax[n].plot(water_volume[:,n+4],'y.-',label=f"est(c{n+3}(K))")
    ax[n].plot(water_volume[:,n+4]+np.sqrt(uncertainty[:,n+4]),'r--',label="1 sigma")
    ax[n].plot(water_volume[:,n+4]-np.sqrt(uncertainty[:,n+4]),'r--',)

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

✓ 1.7s

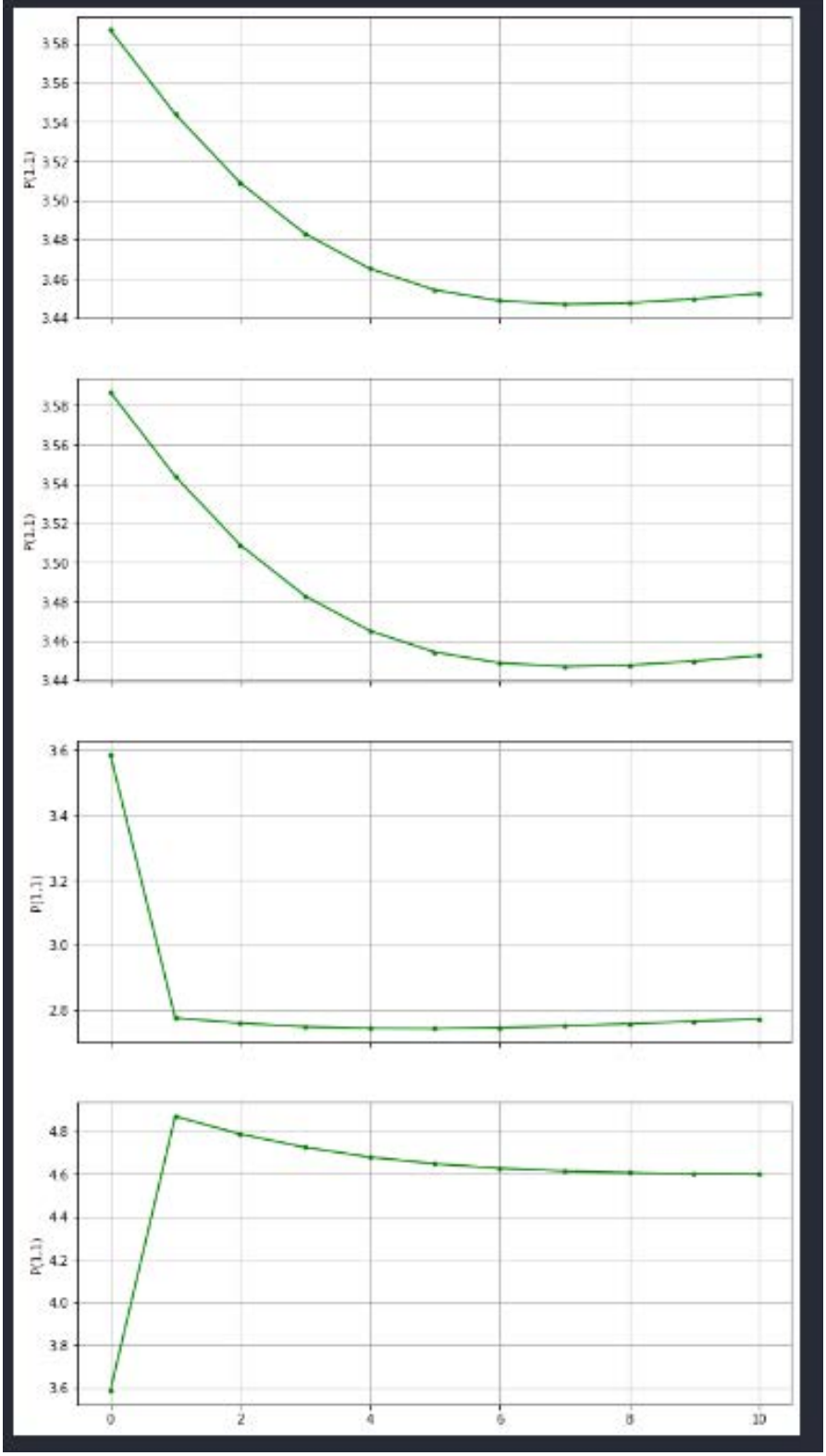





```
uncertainties = [np.zeros([n_steps+1,1]),np.zeros([n_steps+1,1]),np.zeros([n_steps+1,1]),np.zeros([n_steps+1,1])]

for i in range(4):

    uncertainties[i][0,0] = sensor_uncertainty[0,0]

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

    K_uncertainty = d @ sensor_uncertainty @ d.T + process_uncertainty

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

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

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

for n in range(4):

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

✓ 0.8s



```
for i in range(4):
    print(1+i)
    print(uncertainties[i][:,0])
```

✓ 0.1s

```
1
[3.58678078 3.54391255 3.50896257 3.4828954  3.46511245 3.45425284
 3.44870439 3.4469224  3.44758705 3.44967182 3.45244104]
2
[3.58678078 3.54391255 3.50896257 3.4828954  3.46511245 3.45425284
 3.44870439 3.4469224  3.44758705 3.44967182 3.45244104]
3
[3.58678078 2.77486303 2.75857296 2.74780805 2.74249816 2.74184656
 2.74474987 2.75007506 2.75682296 2.76420467 2.77165648]
4
[3.58678078 4.86927961 4.78708804 4.72452577 4.67915383 4.64770437
 4.62687624 4.61374186 4.60591418 4.60157046 4.59939703]
```

(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

```
measurement_model = np.eye(3,8)

measurement_noise_variance = np.eye(3)*2

list_of_measurements = [z1,z2,z4]
measurements = np.mat(list_of_measurements)
```

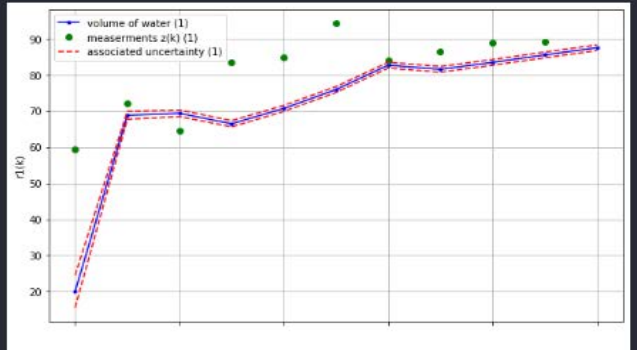
✓ 0.1s Python

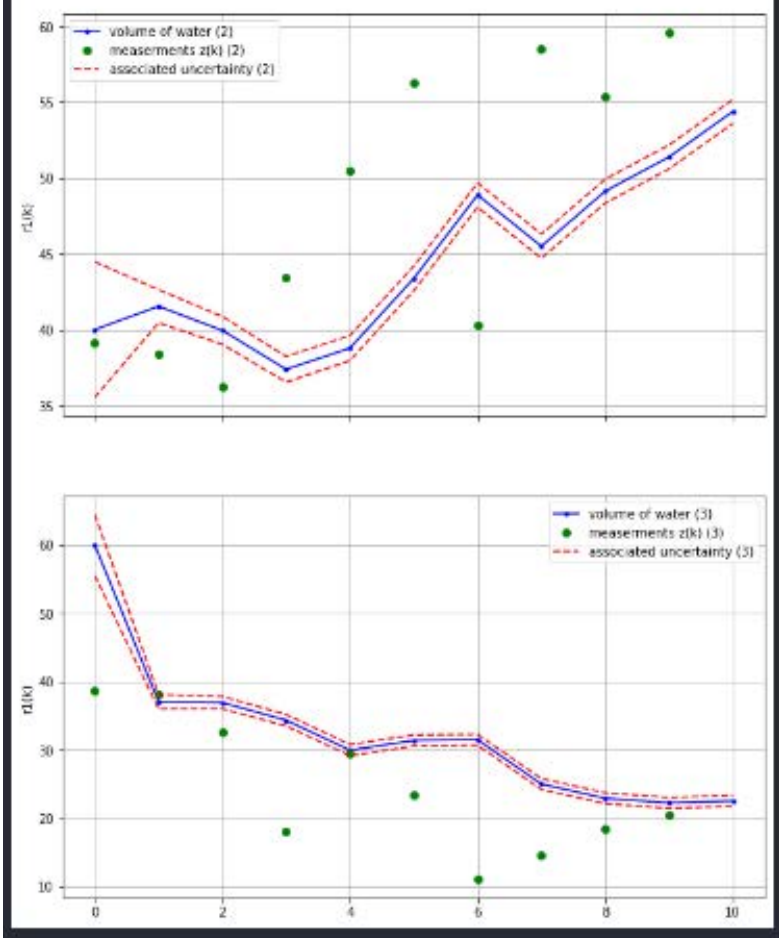
```
for k in np.arange(1,n_steps+1):

    Kp = dynamics @ Er0 + uf
    K_uncertainty = dynamics @ sensor_uncertainty @ d.T + process_uncertainty

    measurement = measurements[:,k-1]

    K = K_uncertainty @ measurement_model.T @ np.linalg.inv(measurement_model @ K_uncertainty @ measurement_model.T + measurement_noise_variance)
    Er0 = Kp + K @ (measurement - measurement_model @ Kp)
    sensor_uncertainty = (np.eye(8) - K @ measurement_model) @ K_uncertainty @ (np.eye(8) - K @ measurement_model).T + K @ measurement_noise_variance @ K.

    for i in range(8):
        water_volume[k,i] = Er0[i,0]
        uncertainty[k,i] = sensor_uncertainty[i,i]
```

✓ 0.1s Python

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

for n in range(3):

    ax[n].plot(water_volume[:,n],'b.-',label=f"volume of water ({n+1})")
    ax[n].plot(list_of_measurements[n],'go',label=f"measerments z(k) ({n+1})")
    ax[n].plot(water_volume[:,n]+np.sqrt(uncertainty[:,n]),'r--',label=f"associated uncertainty ({n+1})")
    ax[n].plot(water_volume[:,n]-np.sqrt(uncertainty[:,n]),'r--',)

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

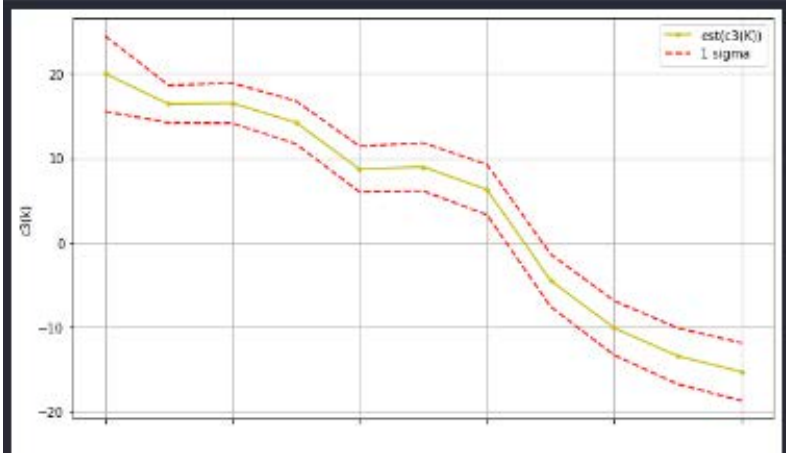
✓ 1.3s Python

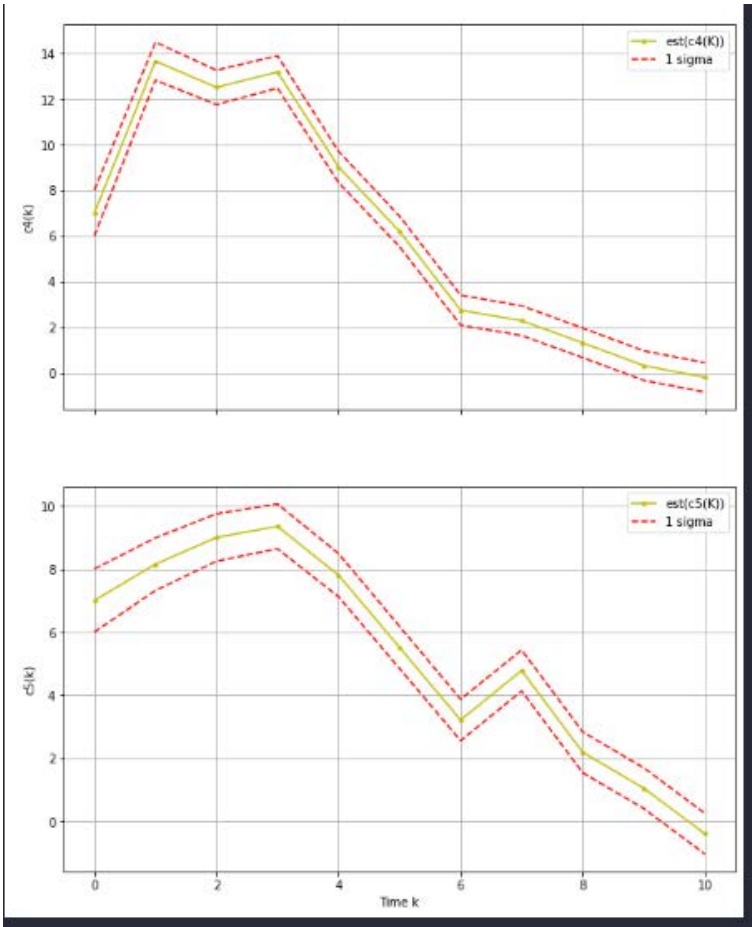





```
fig, ax = plt.subplots(3,1,sharex=True)
fig.set_size_inches(10,20)

for n in range(3):

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

✓ 0.9s

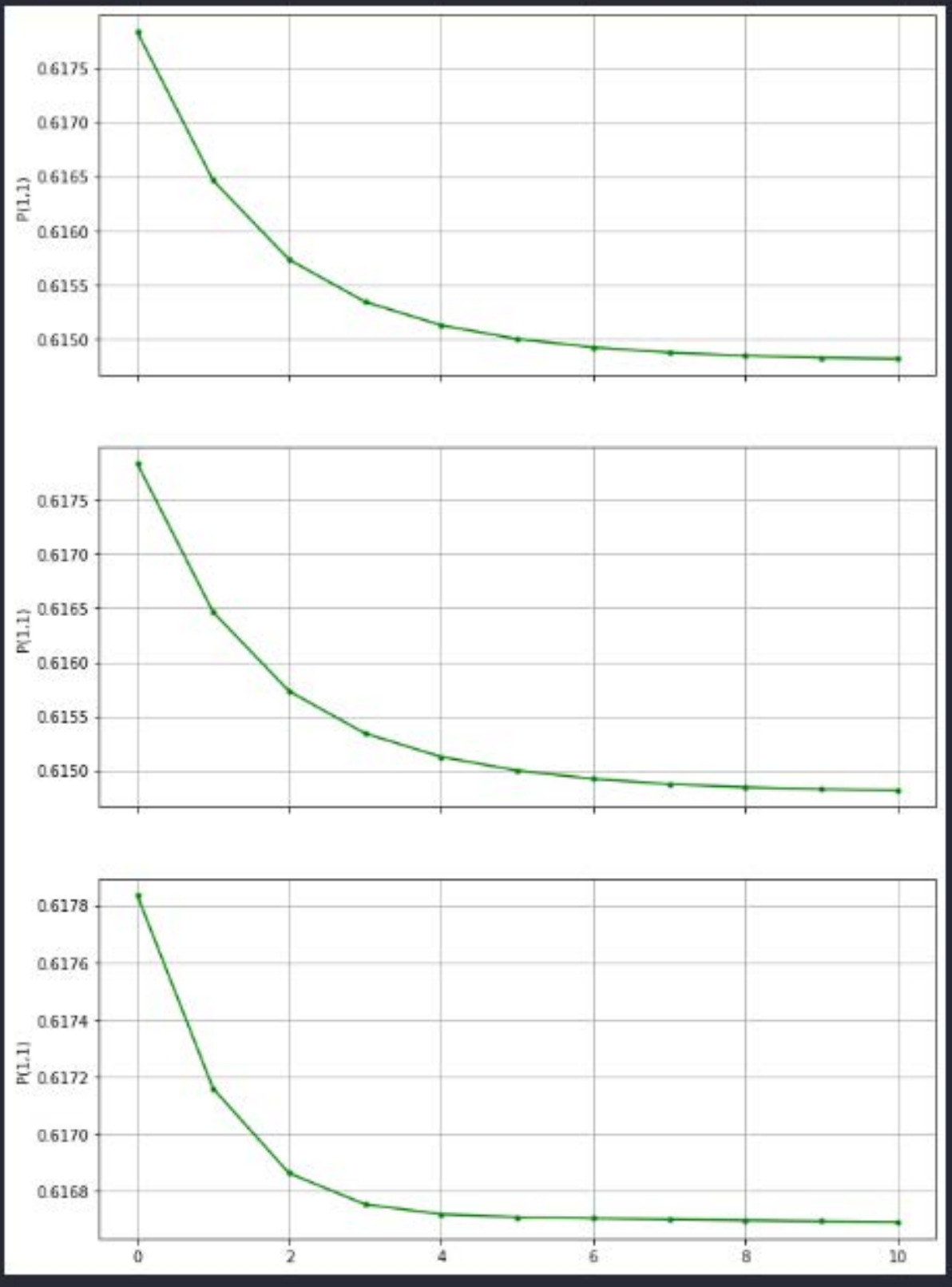





```
uncertainties = [np.zeros([n_steps+1,1]),np.zeros([n_steps+1,1]),np.zeros([n_steps+1,1])]

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

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

    K_uncertainty = d @ sensor_uncertainty @ d.T + process_uncertainty

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

    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.-',label="P(1,1)")

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

✓ 0.8s



Problem 5

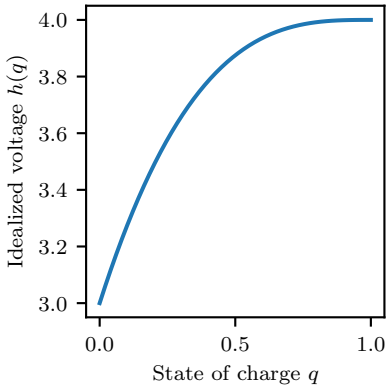
Optional, extra credit. For the battery system of Problem 3, we now add a voltage sensor which gives us a noisy reading of the battery voltage after each time cycle. The measurement is $z(k) = h(q(k)) + w(k)$, with $w(k) \sim \mathcal{N}\left(0, \sigma_w^2\right)$, and where $h(q)$ is a nonlinear function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

```
n_steps = 9

dynamic = 1

ov2 = 0.05**2
ow2 = 0.1**2

q0 = 1  # q(0) = 1
uncertainty_Vr0 = 0

true_state = np.random.normal(1,0)
```

✓ ✓ 0.1s

```
charges = np.zeros([n_steps+1,1]) # to store the charges.
charges[0,0] = q0

uncertainty = np.zeros([n_steps+1,1])
uncertainty[0,0] = uncertainty_Vr0

measurments_ = np.zeros([n_steps+1,1]) # to store measurments.
```

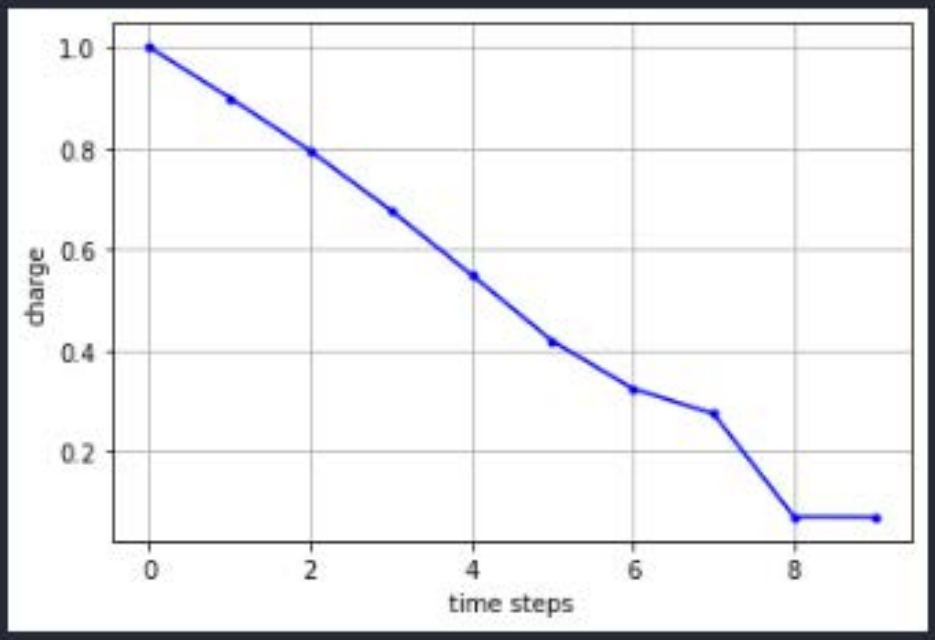
✓ ✓ 0.1s

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

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

    true_state = true_state - noise - 0.1

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

    measurments_[k,0] = 4 + ((true_state)-1)**3 + w


    Kp = q0 - 0.1
    K_uncertainty = dynamic*uncertainty_Vr0*dynamic + ov2

    H = 3*(Kp-1)**2
    K = K_uncertainty*H*1/(H*K_uncertainty*H + ow2)

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

    q0 = Kp + K*(measurments_[k,0]-AVGs)
    uncertainty_Vr0 = (1 - K*H)*K_uncertainty*(1 - K*H) + K*ow2*K

    charges[k,0] = q0
    uncertainty[k,0] = uncertainty_Vr0
```

✓ ✓ 0.2s

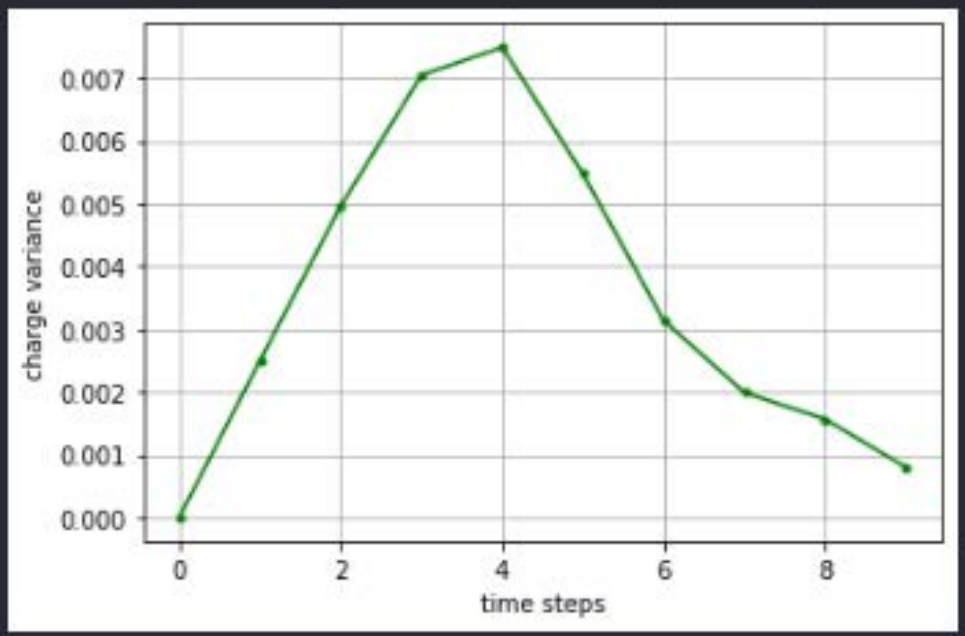
```
plt.plot(charges[:,0],'b.-')
plt.xlabel('time steps')
plt.ylabel('charge')
plt.grid(True)
```

✓ 0.4s




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

✓ 0.4s



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

q0 = 1
ow = u = 0.1

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

Xs[0] = q0
Hs[0] = (q0-1)**2
Us[0]=(Hs[0]**2*0.1)/(Hs[0]**2*0.1+ow)
```

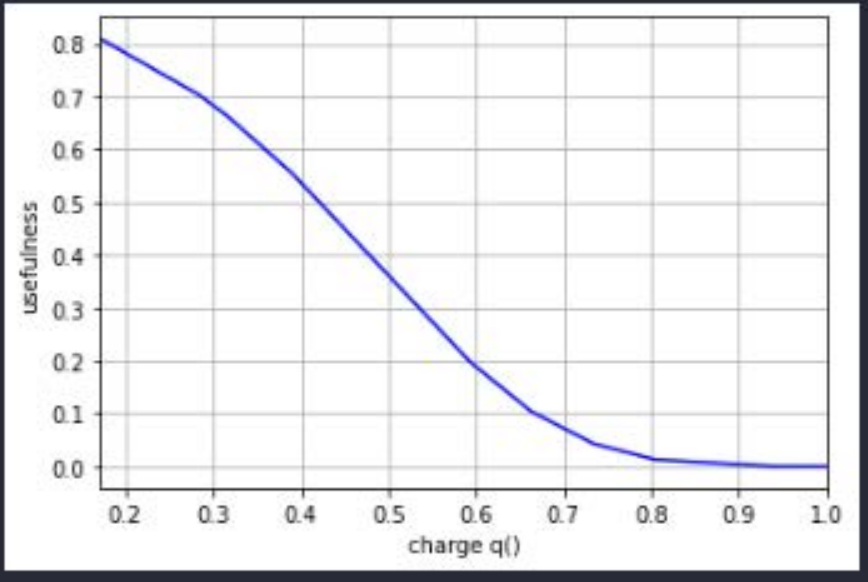
✓ ✓ 0.1s

```python
for k in range(n_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+ow)
```

✓ ✓ 0.1s

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

✓ 0.5s



```
q0 = 1  # q(0) = 1
uncertainty_Vr0 = 0

states = np.zeros([n_steps+1,1]) # to store the history of the true states.
true_state = np.random.normal(1,0)
states[0,0] = true_state
```

✓ ✓ 0.1s

```
measurments = [4.21, 3.83, 3.92, 3.89, 3.88, 3.89, 3.91, 3.57, 3.21]
```

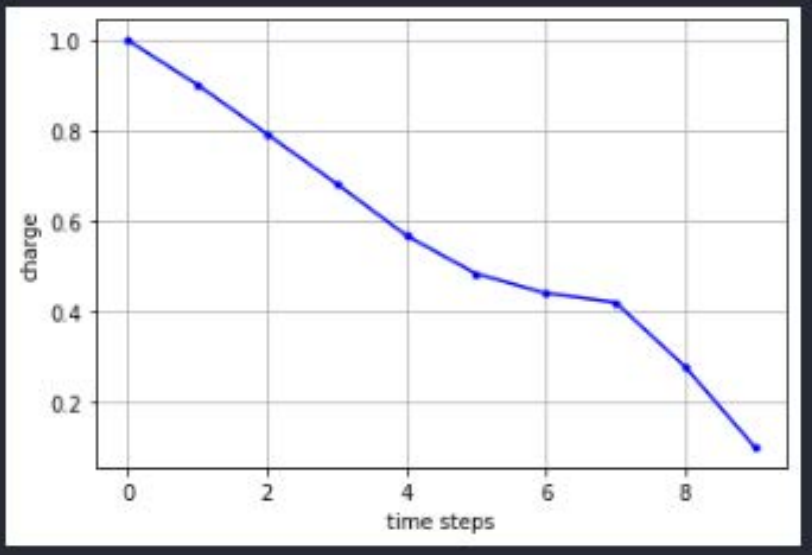
✓ ✓ 0.1s

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

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

    true_state = true_state - noise - 0.1

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

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

    Kp = q0 - 0.1
    K_uncertainty = dynamic*uncertainty_Vr0*dynamic + ov2

    H = 3*(Kp-1)**2
    K = K_uncertainty*H*1/(H*K_uncertainty*H + ow2)

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

    q0 = Kp + K*(measurments_[k,0]-AVGs)
    uncertainty_Vr0 = (1 - K*H)*K_uncertainty*(1 - K*H) + K*ow2*K

    charges[k,0] = q0
    uncertainty[k,0] = uncertainty_Vr0
    states[k,0] = true_state
```

✓ ✓ 0.1s

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

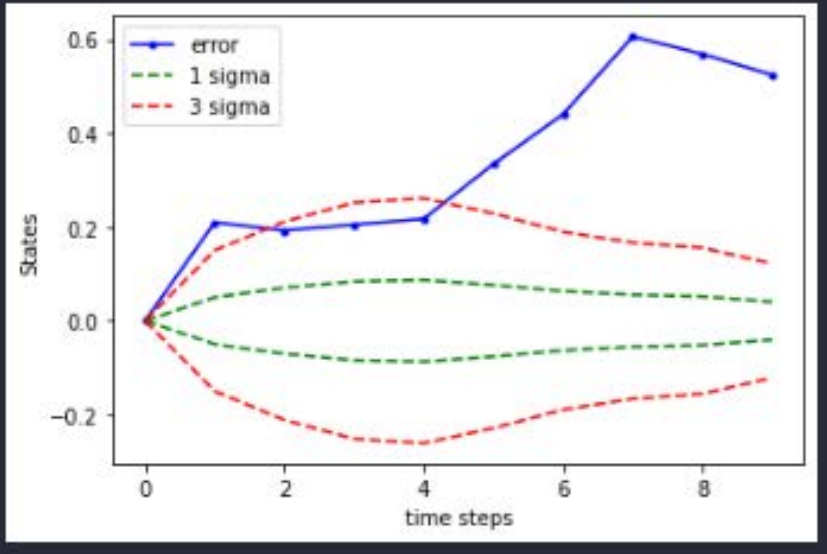
✓ 0.4s




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

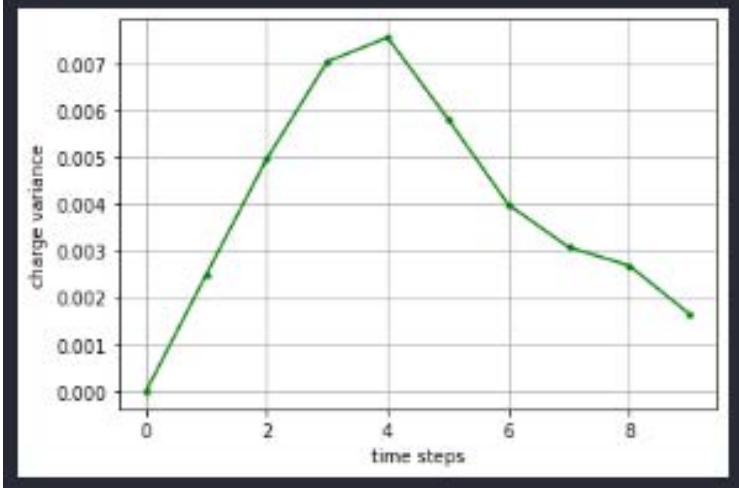
✓ ✓ 0.3s

```
<matplotlib.legend.Legend at 0x24e23b50460>
```



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

✓ 0.3s




```
j0=0.1
sigma=0.05
mean=1-9*j0
var=9*sigma**2

print(f'mean with no voltage measurements: {round(mean,2)}')
print(f'variance with no voltage measurements: {round(var,2)}')
```

✓ 0.1s

```
mean with no voltage measurements: 0.1
variance with no voltage measurements: 0.02
```

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

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

✓ 0.3s