2D Image Processing - Exercise Sheet 5: Bayes and Kalman

The notebook guides you through the exercise. Read all instructions carefully.

- **Deadline**: 14.07.2022 @ 23:59
- **Contact**: Michael.Fuerst@dfki.de
- Points: 21 total, 12.5 passing
- Submission: As PDF Printout; filename is `ex05_2DIP_group_XY.pdf`, where XY is replaced by your group number.
- Allowed Libraries: Numpy, OpenCV and Matplotlib (unless a task specifically states differently).
- Copying or sharing code is NOT permitted and results in failing the exercise. However, you could compare produced outputs if you want to. (Btw, this includes copying code from the internet.)

Submission as PDF printout. You can generate a PDF directly from jupyterlab or if that does not work, export as HTML and then use your webbrowser to convert the HTML to a PDF. For the printout make sure, that all text/code is visible and readable. And that the figures have an appropriate size. (Check your file before submitting, without outputs you will not pass!)

---

0. Infrastructure

This is an image loader function, that loads the images needed for the exercise from the dfki-cloud into an opencv usable format. This allows for easy usage of colab, since you only need this notebook and no other files.

In [1]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

---

1. Recursive Bayes Filter (13 Pt)

Often we want to know the state of a dynamical system. Sometimes the state can be directly observed, e.g. chess figures, whereas in other cases the state is unknown, e.g. location of an autonomous robot.

Here we will have a look at the case where we cannot directly observe the case, but have to estimate it. In this case the system has a hidden state x (sometimes μ) and we can take observations y of the world which can be used to estimate the state $\hat{x}$. If we have a dynamic system we also have a control input to the system u which can change the state x.

Let's look at a concrete example: An autonomus train.

- The state x of the train would be the distance along the track covered $d \rightarrow x_t = d_t$.
- The train can measure if it is at a train station or on the open track, resulting in a measurement model $p(z_t|x_t)$. But it cannot directly measure where it is on the track.
- The speed at which the train moves can be controlled. However, the speed lever is a bit loose, so the actual speed is a gaussian distribution with C_u around a mean μ_u. Thus, we have a motion model $p(x_t|x_{t-1}, u_t)$ which returns a probability distribution of the motion.
- Initially the train does not know where it is, thus we assume a equal distribution for he belief initially.

The task is to estimate the position of the robot iteratively using recursive bayes filter.

Theory (1 Pt): Define the recursive bayes formula using the symbols introduced in the text above.

Solution:

$$\texttt{bel}(x_t) = \eta\, p(z_t|x_t) \int p(x_t|x_{t-1}, u_t)\, bel(x_{t-1})\, dx_{t-1}$$

We later want to implement this though, so we need a discrete version as we do not want to integrate. To make a discrete version of the continuous formulation replace the integral $\int dx$ by a sum $\sum_x$.

Theory (1 Pt): Now rewrite the formula as a discrete formula.

Solution:

$$\texttt{bel}(x_t) = \eta\, p(z_t|x_t) \sum_x p(x_t|x_{t-1}, u_t)\, bel(x_{t-1})$$

Motion Model

Now that you have defined the recursive bayes formula it is time to actually implement it. The motion model will be given, but for the measurement model, we have to define it first.

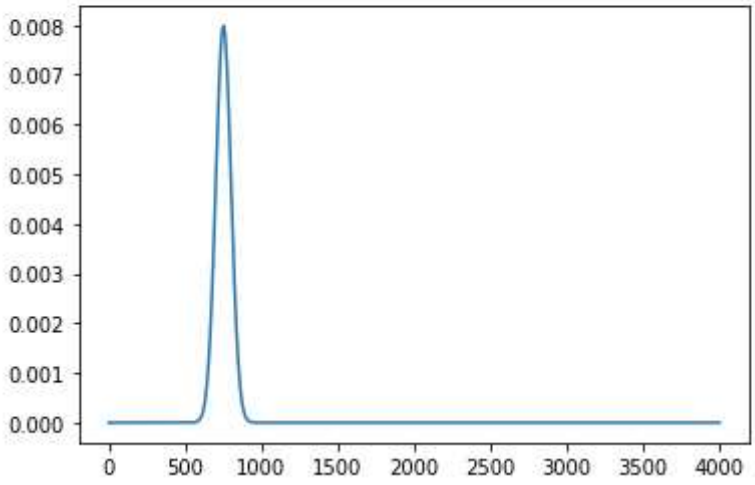
In the the formula we need a motion model $p(x_t|x_{t-1}, u_t)$ but as the lever is loose, this is a gaussian distribution. This means we will have to implement a gaussian distribution first.

The motion model is mathematically defined as: $p(x_t|x_{t-1}, u_t) = \texttt{norm}(x_{t-1} + u_t, C_u)$

Programming Task (1 Pt): Implement a function called `p_motion_model(x_old, u) -> arr` which returns an array that contains the probability distribution as a 1d array. The indices of the output correspond to the probabilities for x*t $\rightarrow p(x*{t}|x*{t-1},u*{t}) = p_motion_model(x_old, u)[x_t]$

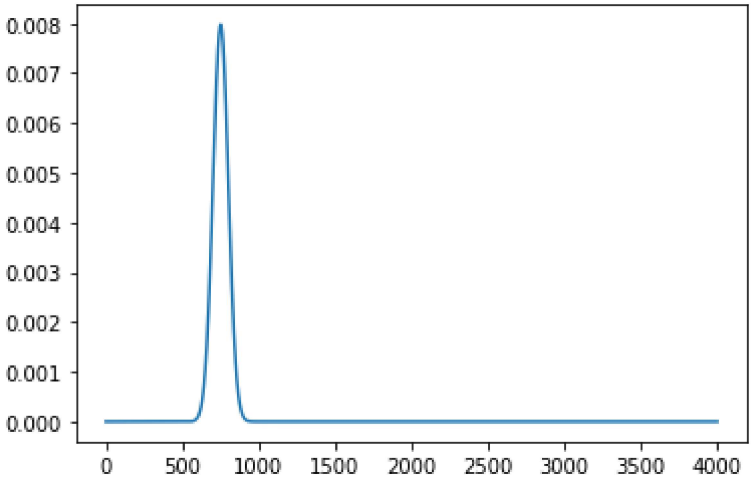
Hint: You are allowed to use `scipy.stats`, but preferably you do not need it. The output should look exactly like this!

```
Shape (4000,)
Max 0.007978845608028654
Argmax 750
Array [1.10614191e-51 1.49283680e-51 2.01391020e-51 ... 0.00000000e+00
 0.00000000e+00 0.00000000e+00]
```



In [58]:
```
# Solution
import numpy as np
import matplotlib.pyplot as plt
C_u = 50

def nm(x, m, s):
    nm = 1/(s*np.sqrt(2*np.pi)) * np.exp(-0.5 * ((np.array(x) - m)/s) *
                                         ((np.array(x) - m)/s))
    return nm

def p_motion_model(x_old, u):
    arr = np.arange(4000)
    s = C_u
    m = x_old + u
    arr = nm(np.array(arr),m,s)
    return arr

out = p_motion_model(x_old=500, u=250)
print("Shape", out.shape)
print("Max", out.max())
print("Argmax", out.argmax())
print("Array", out)
plt.plot(out)
plt.show()
```

```
Shape (4000,)
Max 0.007978845608028654
Argmax 750
Array [1.10614191e-51 1.49283680e-51 2.01391020e-51 ... 0.00000000e+00
 0.00000000e+00 0.00000000e+00]
```



Explanation (1 Pt): (TODO, if your results do not look like shown in the hint, explain what is happening in the visualization of the hint.

The resulting output resembles the given hint. Any cause for deviation can occur when the normal distribution parameters differ in input.

Measurement Model

Next we need to define where the trainstations are and thus our measurement model. Trainstations are positioned at 1000, 1500 and 3500. The train observes the station when it is at the position of the trainstation with a variance of 50 units.

Generally we have two cases to differentiate:

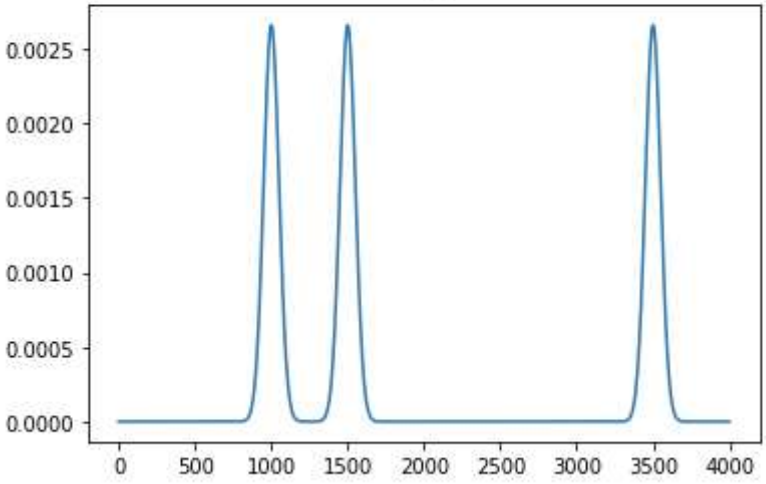
1. We observe the station.
2. We do not observe it.

In the case we observe the station it means we have a gaussian mixture of where we could be. We mix the 3 gaussians centered at 1000, 1500 and 3500 each with a variance of 50. We weight all gassians by a weight of 1/3.

Programming Task (1 Pt): Implement a function `p_measured_station()` returning an array representing gaussian mixture distribution.

Hint: The output should look like this!

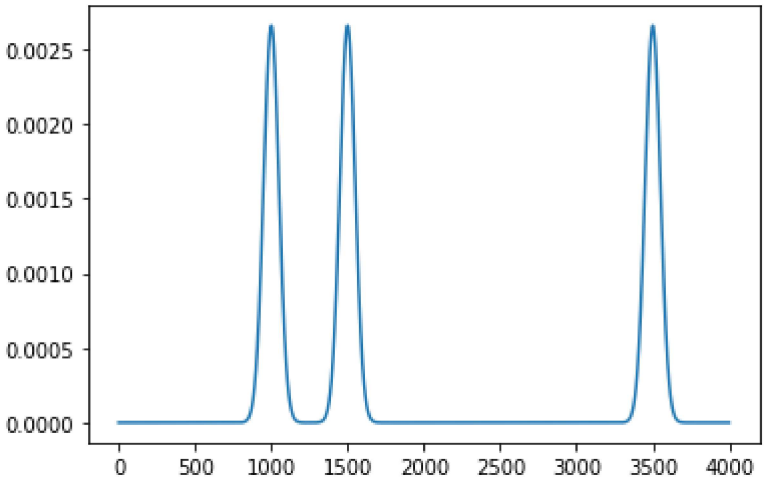
```
Shape (4000,)
Max 0.002659615202676218
Argmax 1000
Array [3.68063224e-90 5.48976002e-90 8.18484720e-90 ... 9.33017245e-25
 7.64654183e-25 6.26421636e-25]
```



In [57]:

```python
# Solution
import scipy
from scipy import stats
C_m = 50


def p_measured_station():
    arr = np.arange(4000)
    # TODO fill array with correct values

    s1 = normal(1000,C_m)
    s2 = normal(1500,C_m)
    s3 = normal(3500,C_m)

    arr = (s1+s2+s3)/3
    return arr

def normal(mean,std):
    #r"Returns an array of length 4000 with a normal distribution"
    denom = (std * np.sqrt(2*np.pi))
    arr = np.array([1/denom*np.e**( -1/2 * ((x-mean)/std )**2) for x in range(4000)])
    return arr

out = p_measured_station()
print("Shape", out.shape)
print("Max", out.max())
print("Argmax", out.argmax())
print("Array", out)
plt.plot(out)
plt.show()
```

```
Shape (4000,)
Max 0.002659615202676218
Argmax 1000
Array [3.68063224e-90 5.48976002e-90 8.18484720e-90 ... 9.33017245e-25
 7.64654183e-25 6.26421636e-25]
```



Explanation (1 Pt): (TODO, if your results do not look like shown in the hint, explain what is happening in the visualization of the hint.

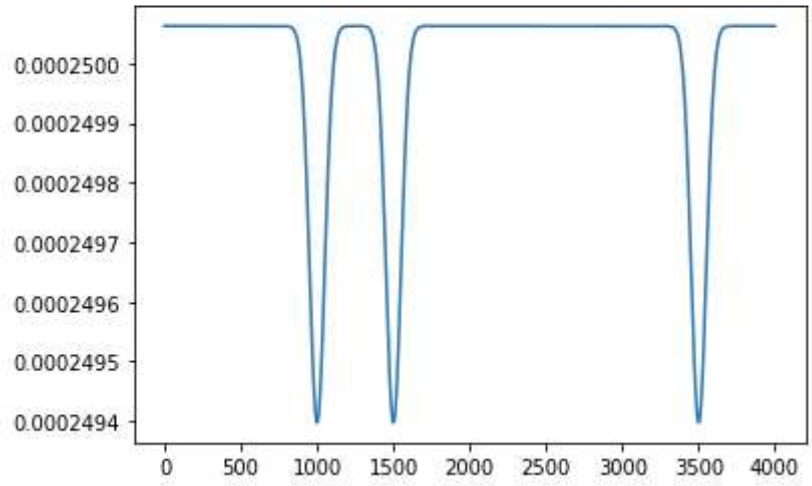
The above graph represents how measurements are generated as function of state. Multiple maxima are observed which is a noisy projection since guassian misxture is used. The maxima is observed at the mean values, where the transitions are postioned at 1000, 1500 and 3500 with variance C_m = 50.

Now we just need the inverse case of not observing a station.

Programming Task (1 Pt): Implement a function `p_not_measured_station()`.

Hint: Compute 1-array and normalize so that the sum is 1 again (so it is a PDF). Solution should look like this!

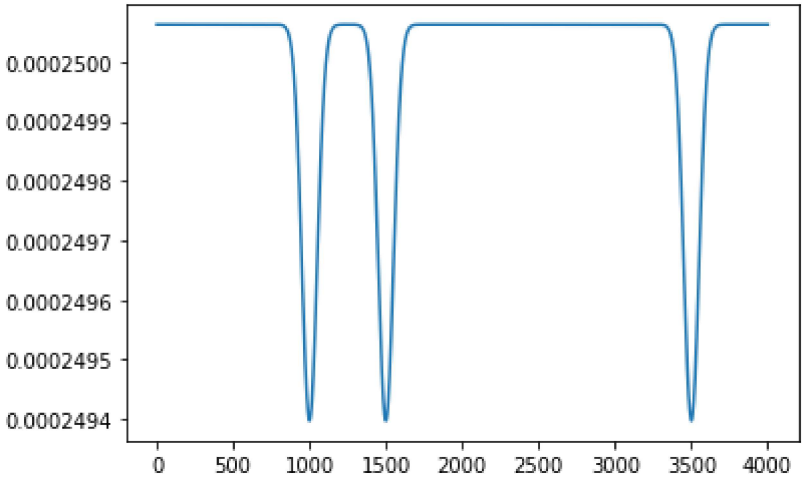
```
Shape (4000,)
Max 0.0002500625156289072 5
Argmax 0
Array [0.00025006 0.00025006 0.00025006 ... 0.00025006 0.00025006 0.00025006]
```



In [55]:
```
# Solution
import scipy
from scipy import stats
C_m = 50
```

```
def p_not_measured_station():
    array = p_measured_station()
    # TODO invert p_measured_station
    array = 1- p_measured_station()
    array = array/array.sum()
    return array

out = p_not_measured_station()
print("Shape", out.shape)
print("Max", out.max())
print("Argmax", out.argmax())
print("Array", out)
plt.plot(out)
plt.show()
```

```
Shape (4000,)
Max 0.00025006251562890725
Argmax 0
Array [0.00025006 0.00025006 0.00025006 ... 0.00025006 0.00025006 0.00025006]
```



Explanation (1 Pt): (TODO, if your results do not look like shown in the hint, explain what is happening in the visualization of the hint.

The above graph represents how measurements are generated as function of state. Multiple maxima are observed which is a noisy projection since guassian misxture is used. The maxima is observed at the mean values, where the transitions are postioned at 1000, 1500 and 3500 with variance C_m = 50.

Recursive Bayes Prediction

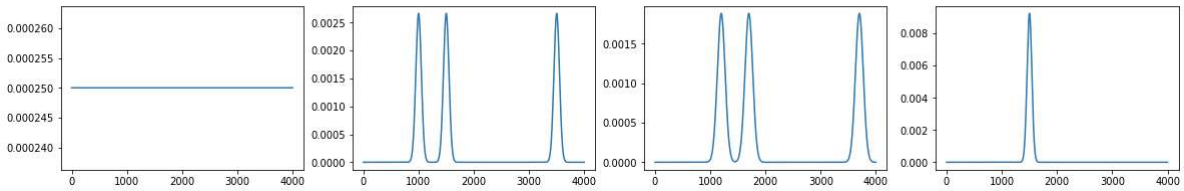
Now that we have all components it is time to finaly localize the train. Let's assume the train does the following:

1. Move 50 then observe a station
2. Move 200 then observe no station
3. Move 300 then observe a station

Programming Task (3 Pt): Implement the function `bel_x = update_belief(bel_x, u, sees_station)` using the recursive bayes method and the functions implemented above.

Hint: The result should look like this.

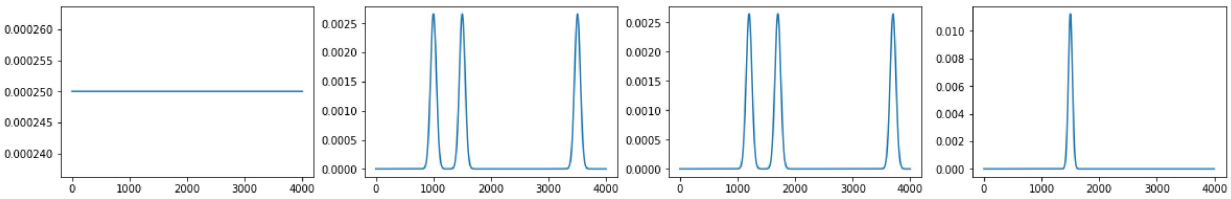
```
Step 0
Max 0.00025
Argmax 0
Sum 1.0000000000000004
Step 1
Max 0.0026596152026762214
Argmax 1000
Sum 1.0000000000000002
Step 2
Max 0.0018808404729843504
Argmax 1700
Sum 1.0
Step 3
Max 0.009213237896155342
Argmax 1500
Sum 1.0000000000000002
```



In [52]:
```python
# Solution
import scipy
from scipy import stats
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from functools import partial

C_u = 5
C_m = 50

def p_motion_model(x_old, u):
    arr = np.arange(4000)
    result = stats.norm.pdf(arr, x_old + u, C_u)

    return result

def p_measured_station():
    arr = np.arange(4000)
    stations = [1000, 1500, 3500]
    result1= stats.norm.pdf(arr, stations[0],C_m)
    result2= stats.norm.pdf(arr, stations[1],C_m)
    result3= stats.norm.pdf(arr, stations[2], C_m)

    return (result1+result2+result3)/3

def p_not_measured_station():
    arr = p_measured_station()

    return (1-arr)/(1-arr).sum()

initial_belief = np.ones(4000) / 4000

def update_belief(bel_x, u, sees_station):
    belief=np.zeros(len(bel_x))
    for k, term in enumerate(bel_x):
        belief+=p_motion_model(k,u)*term
    if sees_station:
```

```
        belief_measure=p_measured_station()
    else:
        belief_measure=p_not_measured_station()
    temp_arr=belief*belief_measure

    return temp_arr/temp_arr.sum()

history = [
    (50, True),
    (200, False),
    (300, True),
]

belief = initial_belief
print(f"Step 0")
print("Max", belief.max())
print("Argmax", belief.argmax())
print("Sum", belief.sum())
plt.figure(figsize=(20,3))
plt.subplot(1,4,1)
plt.plot(belief)
for i, (u, station) in enumerate(history):
    belief = update_belief(belief, u, station)
    print(f"Step {i+1}")
    print("Max", belief.max())
    print("Argmax", belief.argmax())
    print("Sum", belief.sum())
    plt.subplot(1,4,i+2)
    plt.plot(belief)
plt.show()
```

```
Step 0
Max 0.00025
Argmax 0
Sum 1.0000000000000004
Step 1
Max 0.002659615202676218
Argmax 1000
Sum 1.0000000000000002
Step 2
Max 0.0026465066513932465
Argmax 1700
Sum 1.0000000000000002
Step 3
Max 0.011228451726811195
Argmax 1500
Sum 1.0000000000000002
```



Explanation (1 Pt): (TODO, if your results do not look like shown in the hint, explain what is happening in the visualization of the hint.
We can visualize from the output that the probability is lower at points that are further away and higher around mean.

Theory (1 Pt): Where did the train initially start in the above task? Give the most likely X position. *Note: Technically we do not know where it started exactly, we just can compute what is most likely.*

Solution:

(TODO give the number and explain how you deducted it.)

Most likely initial X position = 1500 - 300 - 200 - 50 = 950
After 3rd step it's most likely that train is at the position 1500. So we subtract all the movement to get start position.

---

2. Kalman Filter (8 Pt)

Above example is actually a quite amazing example, because we can localize the train without any idea where it is and even the measurements give us just a multimodal distribution where we could be. Just by combining all information the train can be localized.

In the next case, we have a much simpler problem setting (despite a more complicated explanation), which allows us to implement a more efficient and fast solution: The Kalman Filter. It is used to predict the state of a linear system with gaussian distribution.

Imagine an autonomous robot driving indoors.

- The state is the position of the robot $x = (p_x, p_y)$. And the state does not change without control input $\rightarrow A = I$ (identity matrix for state transition matrix).
- The robot can move in x and y direction resulting in a control sequence $u = (v_x, v_y)$. The velocity is linearily applied assuming a constant time delay (Δ_t) $\rightarrow B = \Delta_t \cdot I$.
- As measurements we can measure z using a measurement function H (H defined later): $z = Hx$.

Theory (1 Pt): Write the matrix A and B.

Solution:

(TODO beneath are example matrices)
$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
$B = \begin{bmatrix} \Delta_t & 0 \\ 0 & \Delta_t \end{bmatrix}$
Considering there is constant time delay, so replacing the time difference Δ_t with identity matrix
$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$$A = \begin{bmatrix} 42 & -1 \\ 13 & 37 \end{bmatrix}$$

$$B = \begin{bmatrix} 42 & -1 \\ 13 & 37 \end{bmatrix}$$

Theory (1 Pt): Define the Time Update (aka Prediction) step of the kalman filter using A, B, x, u, estimated system covariance matrix $\hat{C}$ and motion covariance matrix C^u.

Solution:

(TODO use tech, and symbols from question not copy internet or lecture!)

$x_{t_p} = A_t x_{t-1} + B_t u_t$
$\hat{C}_p = A_t \hat{C}_{t-1} A_t^T + C^u$
In time update or the prediction state, we are considering the position of the object from the previous state and going to predict the new state. We use the u vector - it is the control variable matrix which controls the object. A and B matrix are the matrix used to convert the input state matrix into the new state matrix. These are also called as the adaptation matrix.
Also, we have $\hat{C}$ which is called as the process covariance matrix or system covariance matrix. We will find the dot product of matrix A, covariance matrix and the transpose of the A matrix by adding the motion covariance matrix C^u which is the noise present in the system.

Theory (1 Pt): Define the Measurement Update (aka Measurement) step of the kalman filter using H, x, z, estimated system covariance matrix $\hat{C}$ and measurement covariance matrix C^m. Define an intermediate variable for the Kalman Gain K.

Solution:

(TODO use tech, and symbols from question not copy internet or lecture!)

$x_t = x_{t_p} + K_t(z_t - H_t x_{t_p})$ -> 1
$\hat{C} = (I - K_t C_t)\hat{C}_p$ -> 2
In the above equations we are going to find the update in the measurement with respect to the new state of the object. In the equation we have z_t which is the dot product measured position and the H matrix which converts into the right format and gives us the z_t vector. In both equation we have K which is known as Kalman Gain. Kalman gain K actually decides how much of the estimate we have to impart on the measurement and the predicted new state. Kalman gain decides what fraction to use from predicted state and measurement input to gives new state.
Kalman gain involves,
$K = \hat{C}_p H_t^T (H_t \hat{C}_p H_t^T + C_t^m)^{-1}$
In Kalman Gain, we take measurement covariance matrix C^m into consideration along with the predicted covariance. To calculate how much variance in the measurement and how much variance in the estimate from which we used to calculate Kalman Gain K, which is then used to predict the new state of the object. In equation 1, it takes measurement which is subtracted

from previous predicted state and assigns the Kalman gain to that. After that it is added to the predicted position to get the new position of the object.
In equation 2, we need to update the process or measurement covariance by using the kalman gain K which is going to be the input to the next cycle

Programming Task (4 Pt): Now implement the time update and measurement update using numpy (you only need np.dot and basic operators). Assume H to be:

$$H = \begin{bmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{bmatrix}$$

In [59]:
```
# Solution
# Covariance for measurement and motion
C_u = np.array([
    [0.2, 0],
    [0, 0.2]
])
C_m = np.array([
    [0.2, 0.01],
    [0.01, 0.2]
])
H = np.array([
    [0.8, 0.2],
    [0.2, 0.8]
])
A = np.array([[1., 0.],[0.,1.]]) # A matrix
B = np.array([[1., 0.],[0.,1.]]) # B matrix
I = np.identity(2)

# Implement the filter
def predict(x, C_hat, u):

    x = np.dot(A,x) + np.dot(B,u) # Predicting the object position

    C_hat = np.dot(A, np.dot(C_hat, A.T)) + C_u #Finding and updating the C
    return x, C_hat

def measure(x, C_hat, z):

    IS = np.dot(H, np.dot(C_hat, H.T)) + C_m
    # Calculating the Kalman gain
    k_gain = np.dot(C_hat, np.dot(H.T, np.linalg.inv(IS)))
    IM = z - np.dot(H, x)
    x = x + np.dot(k_gain, IM)
    IG = I - np.dot(k_gain, H)
    C_hat = np.dot(IG, C_hat)
    return x, C_hat

# Initial estimate (wide distribution as we have no clue)
x = np.array([5.,5.])
C_hat = np.array([
    [10., 0.],
    [0., 10.]
])

# Simulate and run filter
```

```
x_true = np.array([4.,4.])
motions = np.array([
    [3., 1.],
    [-3., 3.],
    [1., -2.],
    [1., -3.],
])
for i, u in enumerate(motions):
    x_true += u
    z = np.dot(H, x_true)
    print(f"X_True", x_true)
    x, C_hat = predict(x, C_hat, u)
    print("Pred", x, C_hat.flatten())
    x, C_hat = measure(x, C_hat, z)
    print("Measure", x, C_hat.flatten())
```

```
X_True [7. 5.]
Pred [8. 6.] [10.2  0.   0.  10.2]
Measure [7.02017291 5.02017291] [ 0.35378811 -0.14802442 -0.14802442  0.35378811]
X_True [4. 8.]
Pred [4.02017291 8.02017291] [ 0.55378811 -0.14802442 -0.14802442  0.55378811]
Measure [4.00687977 8.00687977] [ 0.21981051 -0.0814289  -0.0814289   0.21981051]
X_True [5. 6.]
Pred [5.00687977 6.00687977] [ 0.41981051 -0.0814289  -0.0814289   0.41981051]
Measure [5.00263457 6.00263457] [ 0.19333237 -0.06375082 -0.06375082  0.19333237]
X_True [6. 3.]
Pred [6.00263457 3.00263457] [ 0.39333237 -0.06375082 -0.06375082  0.39333237]
Measure [6.00102535 3.00102535] [ 0.1866083 -0.0583383 -0.0583383  0.1866083]
```

Explanation (1 Pt): (TODO explain the outputs. Where did the robot start, where did it move, what do the observations mean?)

In the above program, the object or the robot actually starts from the position X_True. In the first cycle robot starts from [7, 5] as per the prediction the robot moves to the position [8, 6]. From the output of the measurement by calculating the Kalman gain and deciding how much the prediction is correct the actual movement of the robot was [7.02, 5.02]. And in the next cycle the robot is at [4, 8] and the measurement is almost the same position where there is slight variation in decimal points.