augmented VISION | TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

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:

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

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:

$\texttt{bel}(x_t) = \eta \cdot p(z_t|x_t) \int p(x_t|x_{t-1}, u_t) \cdot p(x_{t-1}|z_{1:t-1}, u_{1:t-1}) dx$

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.

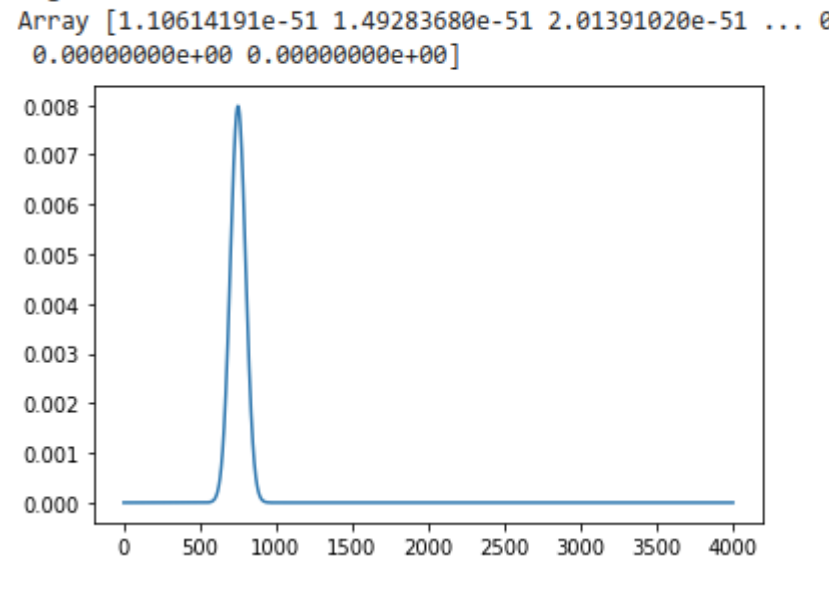
In the the formula we need a motion model $p(x_t|x_{t-1}, u_t)$ but as the lever is loose, this is a gaussian distribution. This means we will have to implement a gaussian distribution first.

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 *xt $\rightarrow p(x*{t}|x*{t-1},u*{t}) = p_motion_model(x_old, u)[x_t]$

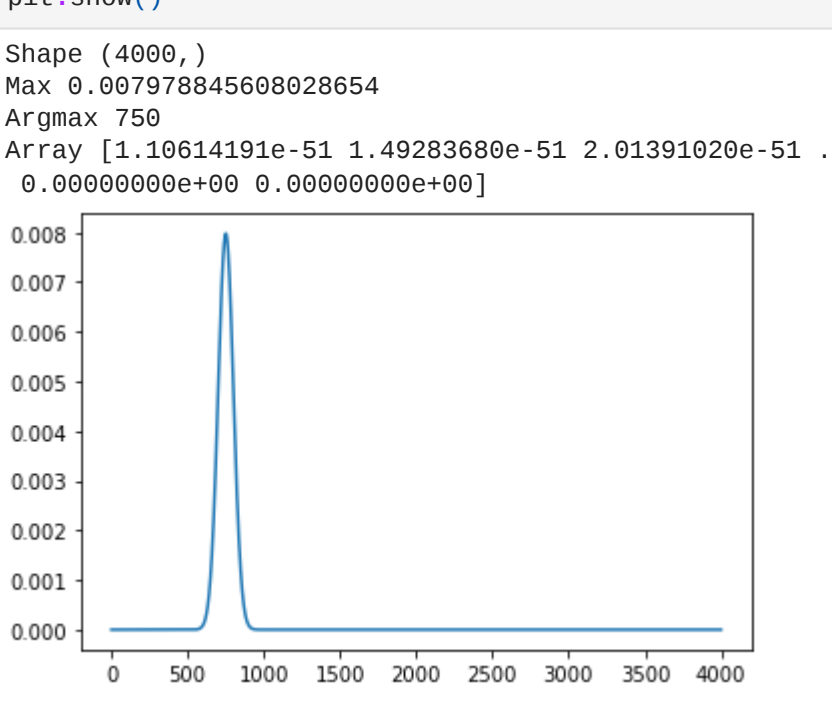
Hint: You are allowed to use `scipy.stats`, but preferably you do not need it. The output should look exactly like this!

```
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 [2]:
```
# Solution
import scipy
from scipy import stats
C_u = 50

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

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): The gaussian peak can be seen near a point where the probability distribution is very dense. *Location parameter is used to fit the distribution for the motion model. **Scale* parameter is used to include the error added by the "Loose speed lever" in the distribution.

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:

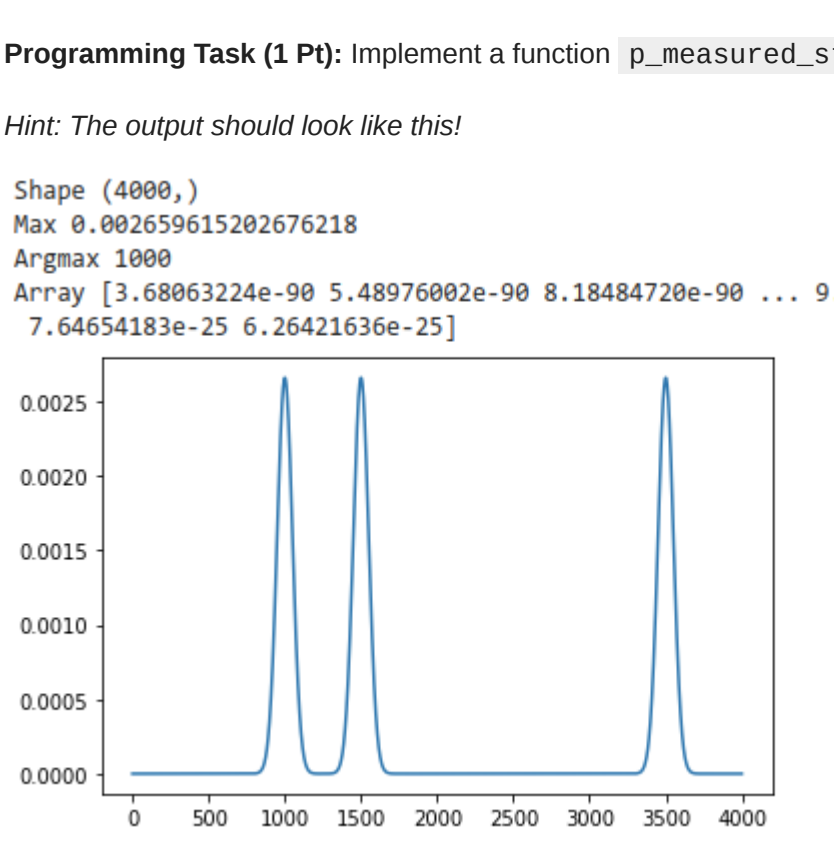
1. We observe the station.
2. We do not observe it.

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.

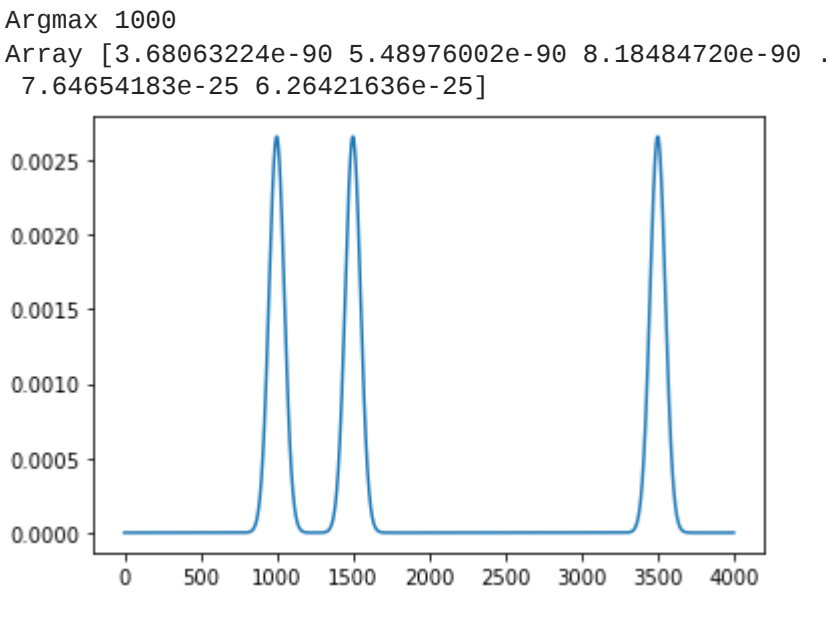
Hint: The output should look like this!

```
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 [3]:
```
# Solution
import scipy
from scipy import stats
C_m = 50

def p_measured_station():
    arr = np.arange(4000)
    return (stats.norm.pdf(arr, loc = 1000, scale = C_m)  +
           stats.norm.pdf(arr, loc = 1500, scale = C_m) +stats.norm.pdf(arr, loc = 3500, scale = C_m))/3

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]
```



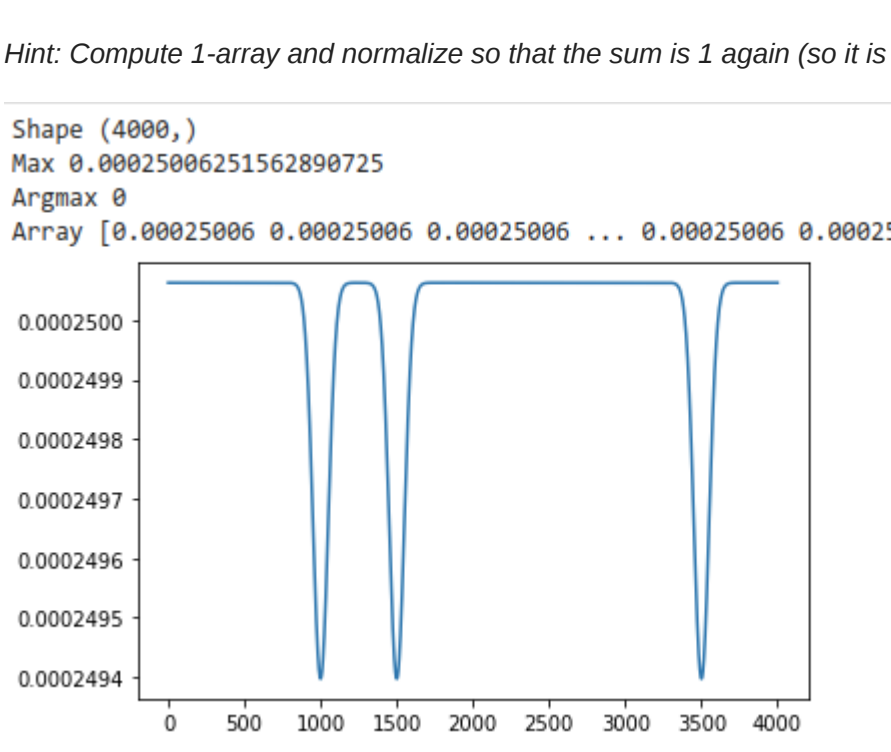
Explanation (1 Pt): We can see the plot showing where the train stations are and how the distribution starts getting dense when the train starts observing the train stations when it is 50 units far from it. It also shows that the distribution is maximum dense when the train reaches and hence observes the train station.

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

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

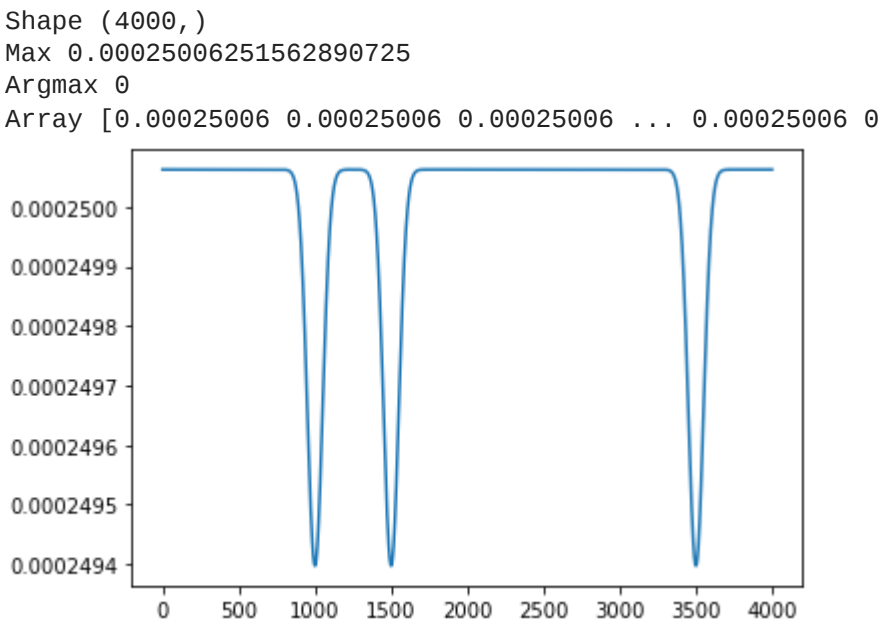
Hint: Compute 1-array and normalize so that the sum is 1 again (so it is a PDF). Solution should look like this!

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



In [4]:
```
# Solution
import scipy
from scipy import stats
C_m = 50

def p_not_measured_station():
    arr = p_measured_station()

    arr= (1-arr)/(1-arr).sum()
    # TODO invert p_measured_station

    return arr

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): In contrast to the task above, here plot shows where the train does not observe the train stations and the distribution is highest when the train is around 50 units far from the train station. The distribution starts getting shallow when the train approaches the train station from 50 units away from it.

Recursive Bayes Prediction

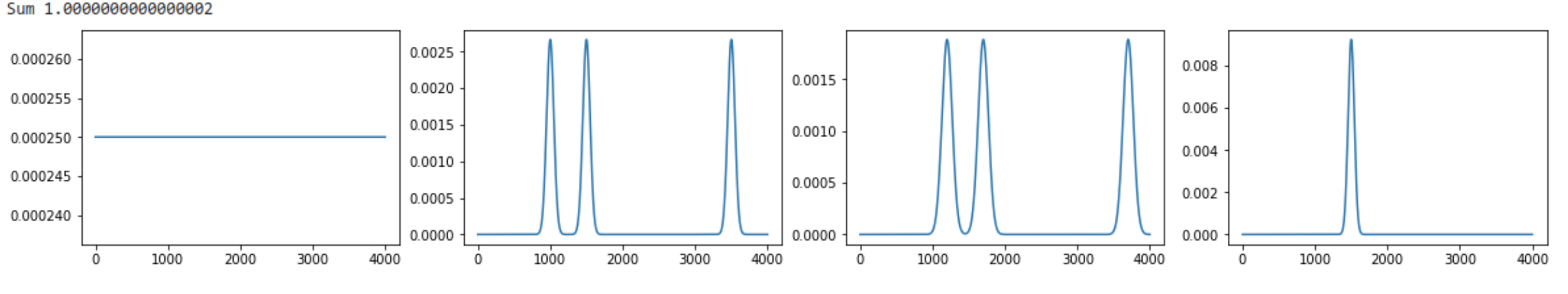
Now that we have all components it is time to finaly localize the train. Let's assume the train does the following:

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.

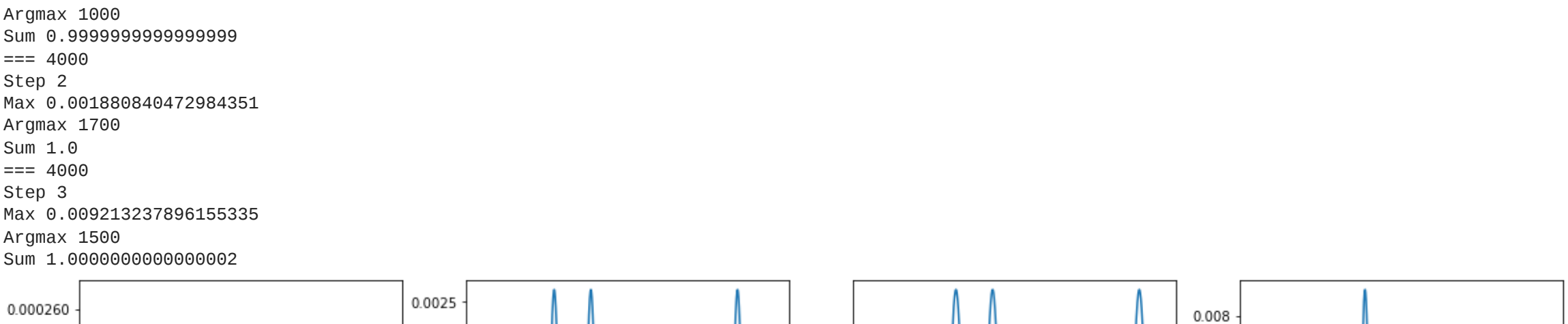
Hint: The result should look like this.

```
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 [5]:
```
# Solution
from functools import partial

initial_belief = np.ones(4000) / 4000

initial_belief = np.ones(4000) / 4000

def update_belief(bel_x, u, sees_station):
    arr = np.zeros(4000)
    # TODO propagate belief
    def update_time(bel_x, u_t):
        motion_m= partial(p_motion_model, u=-u_t)
        prediction = np.array([np.dot(motion_m(float(x)), bel_x) for x in range(4000)])
        print("===", len(prediction))
        return prediction

    def update_m(prediction, sees_station):
        measurement_m= p_measured_station()  if sees_station else p_not_measured_station()
        new_bel= measurement_m * prediction
        return new_bel/new_bel.sum()

    return update_m(update_time(bel_x, u), sees_station)

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
=== 4000
Step 1
Max 0.002659615202676221
Argmax 1000
Sum 0.9999999999999999
=== 4000
Step 2
Max 0.001880840472984351
Argmax 1700
Sum 1.0
=== 4000
Step 3
Max 0.009213237896155335
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.

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.)

---

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 $(\Delta_t) \rightarrow B = \Delta_t \cdot I$.
- 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)

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

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:

$x_{pt} = A_t x_{t-1} + B_t u_t$

$\hat{C} = A_t C^u_{t-1} A_t^T + R_t$

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:

$bel(x_t) = \mu_t = \bar{\mu}_t + K_t(z_t - C_t\bar{\mu}_t)$

$and \hat{C} = (I - K_t C_t) C^m$

$K_t = C^m C_t^T (C_t C^m C_t^T + Q_t)^- 1$

$z_t = H_t x_{pt}$

$K_t = \hat{C} C^{mT} ({C^m}_t \hat{C} C^{mT} + Q_t)^{-1}$

$x_t = X_{pt} + K_t(z_t - C^m x_{pt})$

$\hat{C} = (I - K_t C^m)\hat{C}$

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 [6]:
```
# 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]
])

#B matrix is A times dt (here dt = 1 unit)
B = np.array([
    [1, 0],
    [0, 1]
])


# 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_mat = np.array([
    [1, 0],
    [0, 1]
])

B_mat = np.array([
    [1, 0],
    [0, 1]
])


# Implement the filter
def predict(x, C_hat, u):
    x = np.dot(A_mat,x) + np.dot(B_mat,u)
    C_hat = np.dot(np.dot(A_mat,C_hat),A.transpose()) + C_m
    return x, C_hat

def measure(x, C_hat, z):

    dx = z - np.dot(H,x)
    dC_hat = np.dot(np.dot(H,C_hat),H.transpose()) + C_u
    #Step 2 - Compute Kalman Gain
    K = np.dot(np.dot(C_hat,H.transpose()),np.linalg.inv(dC_hat))
    #Step 3 - Compute final updated mean and variance beliefs
    x = x + np.dot(K,dx)
    C_hat = C_hat - np.dot(np.dot(K,H),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.] [1.02e+01 1.00e-02 1.00e-02 1.02e+01]
Measure [7.0192123 5.0192123] [ 0.36149517 -0.16533763 -0.16533763  0.36149517]
X_True [4. 8.]
Pred [4.0192123 8.0192123] [ 0.56149517 -0.15533763 -0.15533763  0.56149517]
Measure [4.00633904 8.00633904] [ 0.22349855 -0.089488   -0.089488    0.22349855]
X_True [5. 6.]
Pred [5.00633904 6.00633904] [ 0.42349855 -0.079488   -0.079488    0.42349855]
Measure [5.00233049 6.00233049] [ 0.19522744 -0.06875542 -0.06875542  0.19522744]
X_True [6. 3.]
Pred [6.00233049 3.00233049] [ 0.39522744 -0.05875542 -0.05875542  0.39522744]
Measure [6.00086882 3.00086882] [ 0.18763426 -0.06219546 -0.06219546  0.18763426]
```

Explanation (1 Pt): X(4,4) is the initial position of Robot. We have z vector that we got by taking dot product of H and x_true, measurement information and covariation data at every step. Now the information from the C_Hat, state models and motions vector is used to predict the variance and position of robot at every step and thus, the robot reaches to [6.00086882 3.00086882], that is the final destination.

In [ ]: