Comp 150 Probabilistic Robotics Homework 1:

Kalman Filter

1 Kalman Filter: Prediction

A balloon drone has encountered a glitch in its program and needs to reboot its on-board computer. While rebooting, the drone is helpless and cannot issue motor commands. To help the drone, you'll need some understanding of the Kalman filter algorithm.

The drone operates in a 1-D world where xt is the position at time t, while $\dot{x}$ t and $\ddot{x}$t are the velocity and acceleration. For simplicity, assume that delta t = 1.

Due to random wind fluctuations, at each new time step, your acceleration is set randomly accordingly to the distribution N (μwind, σ^2 wind), where μwind = 0.0 and σ^2wind = 1.0.

Question 1.1: what is the minimal state vector for the Kalman filter so that the resulting system is Markovian?
Solution: For the balloon drone, the relevant information about the state is the position and velocity of the drone at a given time. Therefore, the minimal state vector for the Kalman filter should contain both the position and velocity at each time step. This is represented by the state vector:

$$\vec{x}_t = \begin{bmatrix} x_t \\ \dot{x}_t \end{bmatrix}$$

Question 1.2: Design the state transition probability function $p(x_t | u_t, x_{t-1})$. The transition function should contain linear matrices A and B and a noise covariance R.

Solution: The system must follow linear dynamics, with randomness in state transition modelled as Gaussian noise.

$$\vec{x}_t = A_t \vec{x}_{t-1} + B_t \vec{u}_t + \vec{\epsilon}_t$$

Hence, $\epsilon_t \sim \mathcal{N}(0, R)$

The state transition probability is thus given by,

$$\mathcal{N}(A_t x_{t-1}, B_t u_t, R_t)$$

Question 1.3: Implement the state prediction step of the Kalman filter, assuming that at time t = 0, we start at rest, i.e., xt = ˙xt = ¨xt = 0.0. Use your code to calculate the state distribution for times t = 1, 2, …5

Solution: Below is the Python implementation code for the above question:

```
import math
import numpy as np


def KF_prediction(mu_prev, sigma_prev):
    mu_bar = A_t.dot(mu_prev)
    sigma_bar = A_t.dot(sigma_prev).dot(A_t.transpose()) + R_t
    mu = mu_bar
    sigma = sigma_bar
    return mu, sigma


mu_w = 0
var_w = 1
A_t = np.array([[1, 1], [0, 1]])
G = np.array([[0.5], [1]])
R_t = var_w * G.dot(G.transpose())
mu_0 = np.array([[0], [0]])
sigma_0 = np.array([[0, 0], [0, 0]])

for t in range(6):
    if t == 0:
        mu_t = mu_0
        sigma_t = sigma_0
    else:
        mu_t, sigma_t = KF_prediction(mu_t, sigma_t)
        print('\u03BC' + chr(8320 + t) + ' = ')
        print(mu_t)
        print('\u03A3' + chr(8320 + t) + ' = ')
        print(sigma_t)
        print()
```

The above code gives the values of ‘μ’ and the Covariance matrices Sigma ‘∑’.

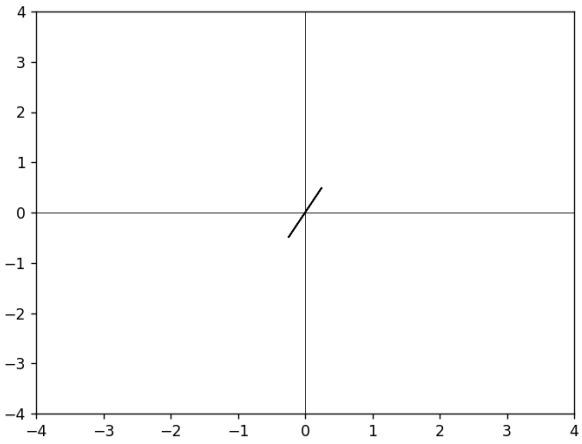
Question 1.4: For each value of t in the previous question, plot the joint posterior over x and ˙x in a diagram where x is the horizontal and ˙x is the vertical axis. For each posterior, you are asked to plot the uncertainty ellipse which is the ellipse of points that are one standard deviation away from the mean.

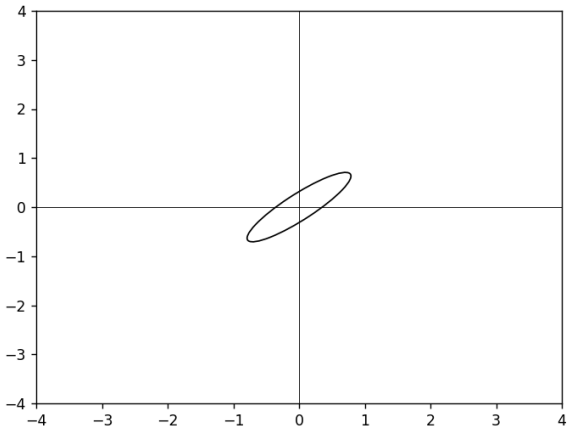
Solution: Below is the Python implementation code for the above question.

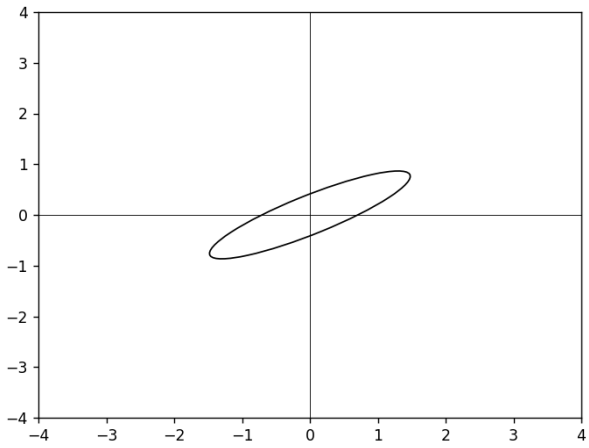
```
import math
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse


def KF_prediction(mu_prev, sigma_prev):
    mu_bar = A_t.dot(mu_prev)
    sigma_bar = A_t.dot(sigma_prev).dot(A_t.transpose()) + R_t
    mu = mu_bar
    sigma = sigma_bar
    return mu, sigma


mu_w = 0
var_w = 1
A_t = np.array([[1, 1], [0, 1]])
G = np.array([[0.5], [1]])
R_t = var_w * G.dot(G.transpose())
mu_0 = np.array([[0], [0]])
sigma_0 = np.array([[0, 0], [0, 0]])

for t in range(6):
    if t == 0:
        mu_t = mu_0
        sigma_t = sigma_0
    else:
        mu_t, sigma_t = KF_prediction(mu_t, sigma_t)
        print('\u03BC' + chr(8320 + t) + ' = ')
        print(mu_t)
        print('\u03A3' + chr(8320 + t) + ' = ')
        print(sigma_t)
        print()
        a = sigma_t[0][0]
        b = sigma_t[0][1]
        c = sigma_t[1][1]
        w = (a+c)/2 + math.sqrt( ((a-c)/2)**2 + b**2 )
        h = (a+c)/2 - math.sqrt( ((a-c)/2)**2 + b**2 )
        t = math.atan2(w-a, b)
        ellipse = Ellipse(xy=(0, 0), width=math.sqrt(w), height=math.sqrt(h),
angle=math.degrees(t), fc='none', ec='black')
        ax = plt.subplot(111)
        ax.add_patch(ellipse)
        ax.set_xlim(-4, 4)
```

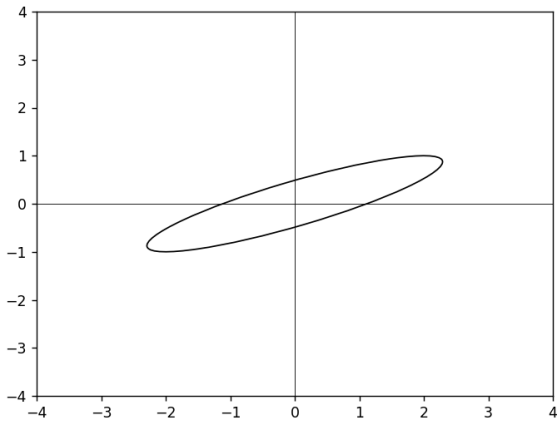
```
        ax.set_ylim(-4, 4)
        ax.axhline(y=0, color='k', lw=0.5)
        ax.axvline(x=0, color='k', lw=0.5)
        plt.show()
```

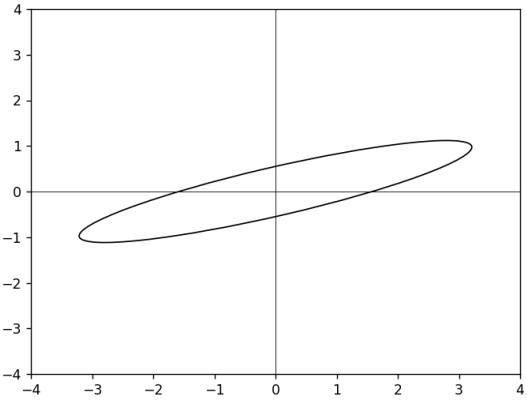
The above code plots the uncertainty ellipses for t=1 to 5 respectively as shown below:











2) Kalman Filter: Measurement Prediction alone will result in greater and greater uncertainty as time goes on. Fortunately, your drone has a GPS sensor, which in expectation, measures the true position. However, the measurement is corrupted by Gaussian noise with covariance σ 2 gps = 8.0.

Question 2.1: Define the measurement model. You will need to define matrices C and Q

Solution: The general expression for the measurement model is **zt = Ctxt + δt.**

Since the GPS is only measuring the position, the measurement vector, zt, is a one dimensional, 1 × 1 matrix. In addition, since the GPS only has noise in the position measurement, the noise vector, δt, is also a one dimensional, 1 × 1 matrix, with zero mean and covariance of σ2 gps.

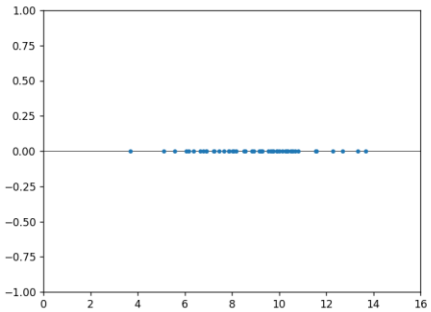
Because the measurement vector, zt, only includes the position, the general expression can be written as:
zt = [1 0] [xt ẋt] + δt
where,
Ct = [1 0] δt ~ N (0, 8)

Question 2.2: Implement the measurement update. Suppose at time t = 5, the drone's computer has rebooted, and we query our sensor for the first time to obtain the measurement z = 10. State the parameters of the Gaussian estimate before and after incorporating the measurement. Afterwards, implement the sensor modal to randomly sample the true position, corrupted with noise σ 2 gps.

Solution: The below code implements the code for the above problem:

```
import math
import numpy as np
import matplotlib.pyplot as plt

xmean = 8.37563452
xvar = 6.70050761
xstn = math.sqrt(xvar)
nx = 50
samp = np.random.normal(loc=xmean, scale=xstn, size=nx)
y = np.zeros(nx)

ax = plt.subplot(111)
ax.set_xlim(0, 16)
ax.set_ylim(-1, 1)
ax.axhline(y=0, color='k', lw=0.5)
plt.plot(samp, y, '.')
plt.show()
```

The below image shows the plot of true positions for a sample size of 50



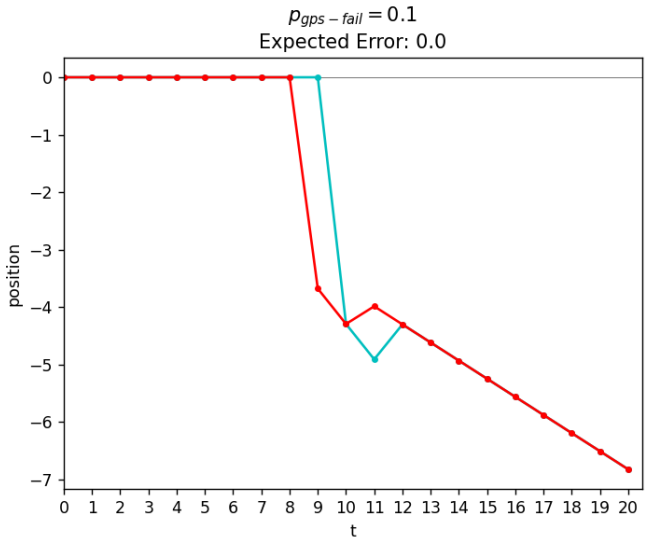
Question 2.3: All of a sudden, the sky gets cloudy which may cause the sensor to fail and not produce a measurement with probability pgps−fail. For three different values of this probability (e.g., 0.1, 0.5, and 0.9), compute and plot the expected error from the true position at time t = 20. You may do so by running up to N simulations and use the observed errors to obtain the expected error empirically.

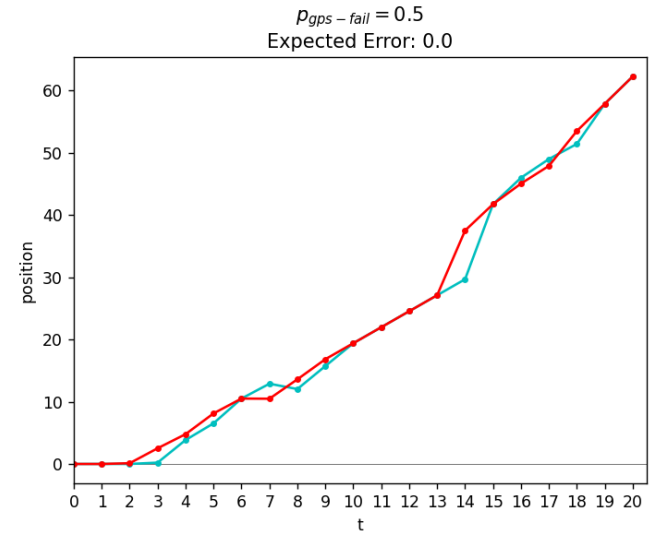
Solution: The below Python code can be used to simulate the prediction, measurement, and estimation of true position with different pgps−fail.

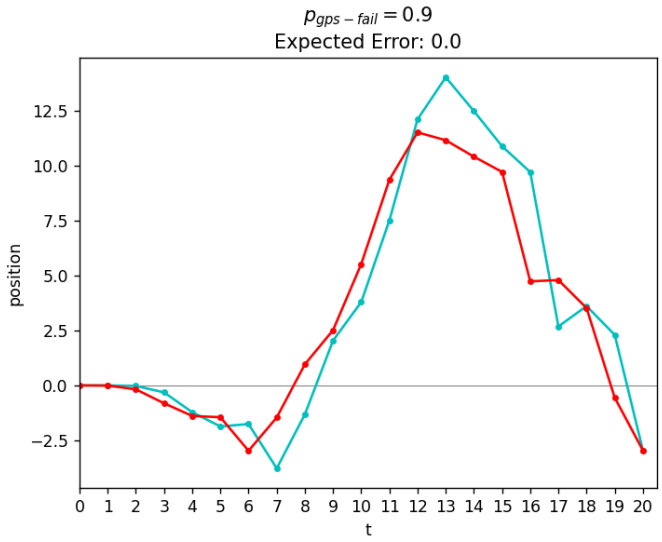
```
import math
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt


def Kalman_filter(mu_prev, sigma_prev, p_fail):
    mu_bar = A_t.dot(mu_prev)
    sigma_bar = A_t.dot(sigma_prev).dot(A_t.transpose()) + R_t
    if np.random.choice(a=[0, 1], size=None, p=[1-p_fail, p_fail]):
        # randomly generate z_t
        z_t = np.array(np.random.normal(loc=mu_bar[0][0],
scale=math.sqrt(sigma_bar[0][0]), size=None))
        K_t =
sigma_bar.dot(C_t.transpose()).dot(inv(C_t.dot(sigma_bar).dot(C_t.transpose()) +
Q_t))
    else:
        # set K_t to zero (no measurement)
        z_t = np.array([[0]])
        K_t = np.array([[0], [0]])
    mu = mu_bar + K_t.dot(z_t - C_t.dot(mu_bar))
    sigma = (np.identity(2) - K_t.dot(C_t)).dot(sigma_bar)
    return mu_bar, mu, sigma


p_gps_fail = float(input("Enter p_gps-fail: "))  # input any float from 0 to 1
mu_w = 0
var_w = 1
A_t = np.array([[1, 1], [0, 1]])
G = np.array([[0.5], [1]])
R_t = var_w * G.dot(G.transpose())
C_t = np.array([[1, 0]])
Q_t = np.array([[8]])
mu_0 = np.array([[0], [0]])
sigma_0 = np.array([[0, 0], [0, 0]])
bel = []
pos = []
```

```
for t in range(21):
    if t == 0:
        mu_bel = mu_0
        mu_t = mu_0
        sigma_t = sigma_0
    elif t != 20:
        mu_bel, mu_t, sigma_t = Kalman_filter(mu_t, sigma_t, p_gps_fail)  #
measurement acquired according to p_gps_fail
    else:
        mu_bel, mu_t, sigma_t = Kalman_filter(mu_t, sigma_t, 0)  # need
measurement at t = 20 to observe error
    bel.append(mu_bel[0][0])
    pos.append(mu_t[0][0])

ax = plt.subplot(111)
ax.set_xlim(-0, 20.5)
plt.xticks(np.arange(0, 21, 1.0))
ax.set_xlabel('t')
ax.set_ylabel('position')
ax.axhline(y=0, color='k', lw=0.3)
plt.plot(list(range(21)), bel, 'c.-')
plt.plot(list(range(21)), pos, 'r.-')
plt.title(r'$p_{gps-fail} = $' + str(p_gps_fail) + '\n' + 'Expected Error: ' +
str(round(abs(pos[-1] - bel[-1]), 3)))
plt.show()
```

The below plots show the resulting values. Blue line indicates the prediction and the red line indicates the true position.







4) Extra Credit Now, formulate both the prediction and measurement steps in the 2-D case. Construct a plot showing the true position and the position tracked by the Kalman filter over the first 30 time steps.

Solution: Below is the python implementation of the above question.

```
import math
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt


def Kalman_filter(mu_prev, sigma_prev):
    mu_bar = A_t.dot(mu_prev)
    sigma_bar = A_t.dot(sigma_prev).dot(A_t.transpose()) + R_t
```

```
    z_t = np.array(np.random.normal(loc=mu_bar[0],
scale=np.array([[math.sqrt(sigma_bar[0][0]), math.sqrt(sigma_bar[1][1])]])))
    K_t =
sigma_bar.dot(C_t.transpose()).dot(inv(C_t.dot(sigma_bar).dot(C_t.transpose()) +
Q_t))
    mu = mu_bar + K_t.dot(z_t - C_t.dot(mu_bar))
    sigma = (np.identity(2) - K_t.dot(C_t)).dot(sigma_bar)
    return mu_bar, mu, sigma


mu_w = 0
var_w = 1
A_t = np.array([[1, 1], [0, 1]])
G = np.array([[0.5], [1]])
R_t = var_w * G.dot(G.transpose())
C_t = np.array([[1, 0]])
Q_t = np.array([[8]])
mu_0 = np.array([[0, 0], [0, 0]])
sigma_0 = np.array([[0, 0], [0, 0]])
belx = []
bely = []
posx = []
posy = []

for t in range(31):
    if t == 0:
        mu_bel = mu_0
        mu_t = mu_0
        sigma_t = sigma_0
        z = np.array([[0, 0]])
    else:
        mu_bel, mu_t, sigma_t = Kalman_filter(mu_t, sigma_t)
    belx.append(mu_bel[0][0])
    posx.append(mu_t[0][0])
    bely.append(mu_bel[0][1])
    posy.append(mu_t[0][1])

ax = plt.subplot(111)
ax.axvline(x=0, color='k', lw=0.3)
ax.axhline(y=0, color='k', lw=0.3)
ax.set_xlabel('x position')
ax.set_ylabel('y position')
plt.plot(belx, bely, 'c.-')
plt.plot(posx, posy, 'r.-')
plt.title("Predicted and True X-Y Position of the Sail Boat")
```

```
plt.show()
```

The below plot shows the next 30 time steps.