PROBLEM 3

For the following problem we use given parameters as shown below:

```
N = 1000 #Number of samples
pdL = 1.5 #phi_dot_L = LEFT commanded wheel speed
pdR = 2 #phi_dot_R = RIGHT commanded wheel speed
r = 0.25 #Radius of each wheel
w = 0.5 #Distance between left and right wheel
stdL = 0.05 #standard deviation LEFT wheel speed
stdR = 0.05 #standard deviation RIGHT wheel speed
stdP = 0.1  #standard deviation measurement error
```

Helper Methods:

- For mean and covariance:

```
def stats(x_data,y_data,N):

    ### MEAN/Centroid of the set
    x_cen=sum(x_data)/N
    y_cen=sum(y_data)/N

    ###Covariance
    #List created after subtracting mean
    x_centered  = [x - x_cen for x in x_data]
    y_centered  = [y - y_cen for y in y_data]
    # multiply the list created after subtracting mean ELEMENTWISE
    xy_multiplied_list = [x_centered[i] * y_centered[i] for i in range(len(x_centered))]

    #square each centered element of the list
    x_cs = [x_centered[i]**2 for i in range(len(x_centered))]
    y_cs = [y_centered[i]**2 for i in range(len(y_centered))]

    COV=sum(xy_multiplied_list)/(N)
    COVx = sum(x_cs)/N
    COVy = sum(y_cs)/N

    mean = (x_cen,y_cen)
    cov = np.array([[COVx],[COVy]])
    return mean,cov
```

- For generating x and y data from the pose:

```
def dataFROMpose(X):
    #create x data and y data
    x_data=[]
    y_data=[]

    for i in range (N):
        x_data.append(X[i][0][0])
        y_data.append(X[i][0][1])

    return x_data,y_data
```

Part A:

(a) Starting with:

$$\left.\begin{aligned} \bar{\varphi}_L &= \dot{\varphi}_L + \epsilon_L, & \epsilon_L &\sim N(0, \sigma_L^2) \\ \bar{\varphi}_R &= \dot{\varphi}_R + \epsilon_R, & \epsilon_R &\sim N(0, \sigma_r^2) \end{aligned}\right\} (13)$$

We can sample ϵ_L & ϵ_R from the Normal distribution described above, which gives us True wheel speeds.

⇒ $\dot{\varphi}_L, \dot{\varphi}_R, r, w, \sigma_L, \sigma_R$ are constants provided hence:

$$\boxed{p(x_{t_2} \mid x_{t_1}, \dot{\varphi}_L, \dot{\varphi}_r, r, w, \sigma_L, \sigma_R) \sim p(x_{t_2} \mid x_{t_1}, \epsilon_L, \epsilon_R)}$$

where ϵ_L & ϵ_R can be sampled using distribution mentioned in (13).

⇒ Next we can get $\bar{\varphi}_L$ & $\bar{\varphi}_R$ using eq(13)

⇒ Then we get Robots velocity in its body centric frame

$$V_R = \begin{bmatrix} V_{xR} \\ V_{yR} \\ \dot{\theta}_R \end{bmatrix} = \begin{bmatrix} r/2\,[\bar{\varphi}_R + \bar{\varphi}_L) \\ 0 \\ \frac{r}{w}\,[\bar{\varphi}_R - \bar{\varphi}_L) \end{bmatrix}$$

⇒ Using the velocity vector we can define a function that maps Robots wheel speeds to Robot's velocity in the Lie (SE(2)] at I

$$\dot{\Omega}(\bar{\varphi}_L, \bar{\varphi}_R) = \begin{bmatrix} 0 & -r/w\,(\bar{\varphi}_R - \bar{\varphi}_L & r/2\,(\bar{\varphi}_R + \bar{\varphi}_L) \\ r/w\,(\bar{\varphi}_R - \bar{\varphi}_L) & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

⇒ Then we can define an exponential map that starts from $X = X_{t_1}$ at $t = t_1$ and reports the robots pose $\gamma(t_2)$ at $t = t_2$:

$$\gamma(t_2) = X_{t_1} \cdot \exp\left[(t_2 - t_1)\,\dot{\Omega}\right]$$

The above discussed velocity-based motion model can be implemented as follows:

First, we use a function that adds noise to the wheel velocities using the parameters for a normal distribution:

```
#Function that adds noise to the LEFT and RIGHT wheel speeds
def add_sample():
    #Get True wheel speeds by adding noise
    phi_L = pdL + np.random.normal(0,stdL)
    phi_R = pdR + np.random.normal(0,stdR)
    return phi_L,phi_R
```

Then we define a function that uses the noisy wheel speeds to get the robots velocity vector in its body centric frame. We can use the velocity vector for the Lie Algebra motion model to derive an integral curve that describes the robot's trajectory. We can use it to derive the pose of robot on this trajectory at a given time.

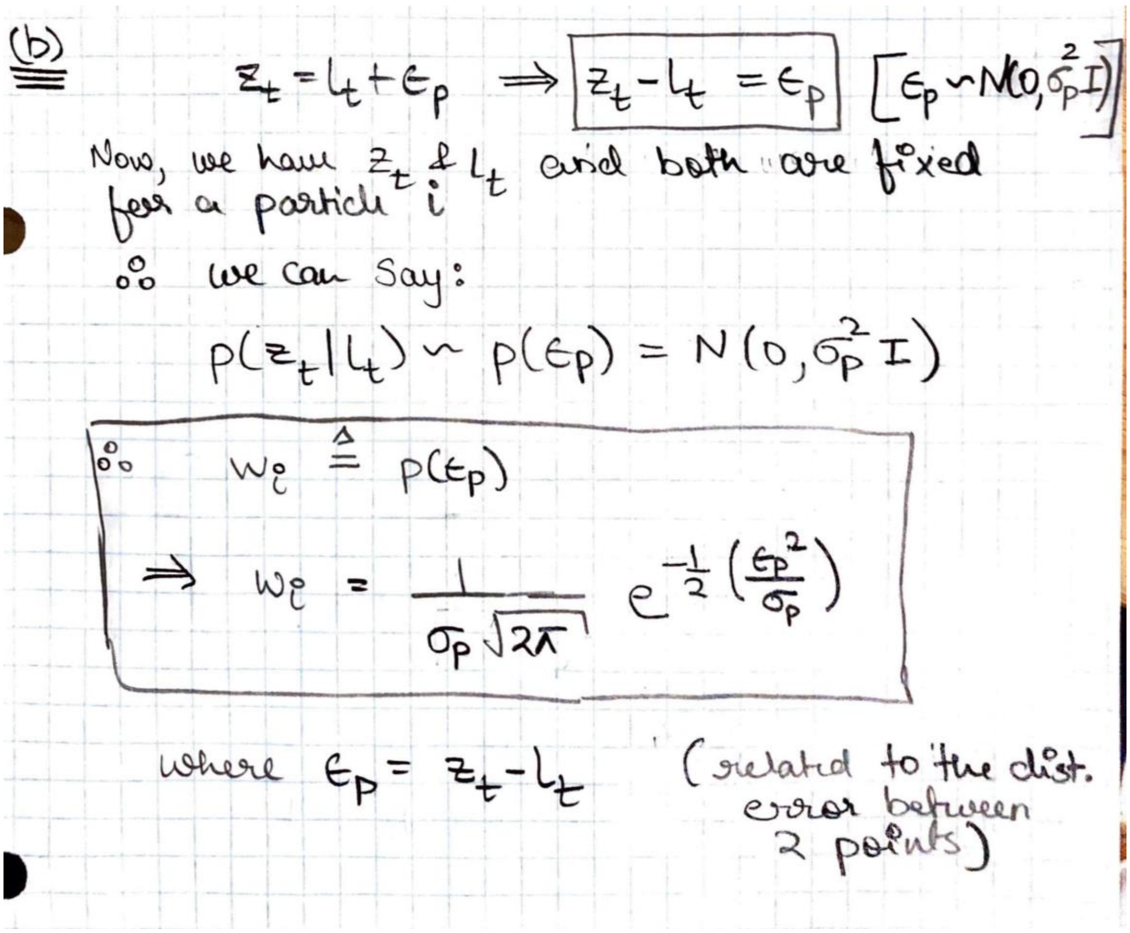
```
def add_pose(t1,X_t1,t2):
    #sample noisy wheel speeds
    vl,vr = add_sample()
    #vl,vr = 1.5,2

    #Velocity vector of the robot in its body centric frame
    V=np.array([[(r/2)*(vr+vl)],[0],[(r/w)*(vr-vl)]])

    #Mapping that sends the pair of wheel speeds (  φ̇l,  φ̇r)
    #to the robot's velocity  Ω̂(  φ̇l,  φ̇r)
    #in the Liegroup SE(2) at I.
    omega = np.array([[0,-V[2][0],V[0][0]],[V[2][0],0,0],[0,0,0]])

    #Exponential map that describes the integral curve of velocity vector
    #It makes use of the mapping defined above where omega is the element of Lie(SE(2))
    #Using matrix exponential we bring the velocity BACK
    #from the Liegroup[SE(2)] to the group[SE(2)]
    #γ: R→SE(2) that starts at X=X1 and t=t1 and reports the robot's pose γ(t2) at time t2
    X_t2 = X_t1 @ expm((t2-t1)*omega)

    return X_t2
```

We can see the implementation of the above method that uses the robot's path from t =0 where the pose is I_3 and derives it subsequent pose at time t=1.

```
In [34]: add_pose(0,np.eye(3),1)

Out[34]: array([[ 0.95999675, -0.28001115,  0.43496632],
                [ 0.28001115,  0.95999675,  0.06214062],
                [ 0.        ,  0.        ,  1.        ]])
```

Part B

The closed form of measurement likely hood formula is p(zt|xt) is p(ϵp) as the only variation here is happening is due to measurement noise which is the difference between the zt and xt **[the distance between coordinated of the 2 points].**



(b)

$$z_t = l_t + \epsilon_p \Rightarrow \boxed{z_t - l_t = \epsilon_p} \quad [\epsilon_p \sim N(0, \sigma_p^2 I)]$$

Now, we have z_t & l_t and both are fixed for a particle i

∴ we can say:

$$p(z_t|l_t) \sim p(\epsilon_p) = N(0, \sigma_p^2 I)$$

$$\boxed{\begin{aligned} \therefore \quad & w_i \triangleq p(\epsilon_p) \\ \Rightarrow \quad & w_i = \frac{1}{\sigma_p \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\epsilon_p^2}{\sigma_p}\right)} \end{aligned}}$$

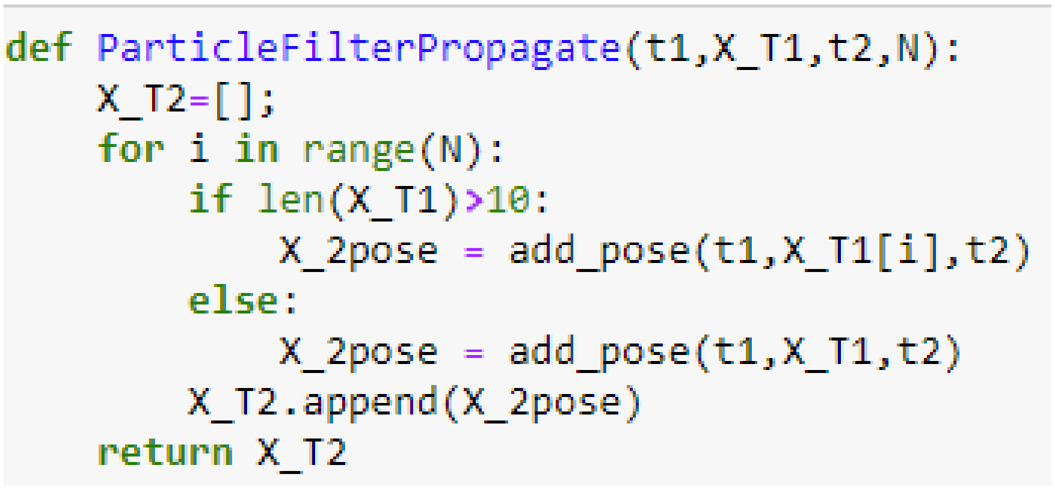
where $\epsilon_p = z_t - l_t$ (related to the dist. error between 2 points)

The above discussed procedure for evaluating the measurement likelihood function can be implemented as follows:

```python
def CalculateImportanceWeights(X,z):
    weights=[]
    distance = []

    XX = np.array(X)
    translation = XX.T[2]

    #Samples weights using the distance error of each particle from the provided measurement zt
    for i in range (N):
        x,y = translation[0:2,i]
        dis = np.linalg.norm(z-np.array([[x],[y]]))
        w = norm.pdf(dis, loc=0, scale=stdP)
        distance.append(dis)
        weights.append(w)

    #convert List to numpy array
    wnp=np.array(weights)
    weight_nparr=np.reshape(wnp,(1,N))

    return weight_nparr
```

Part C

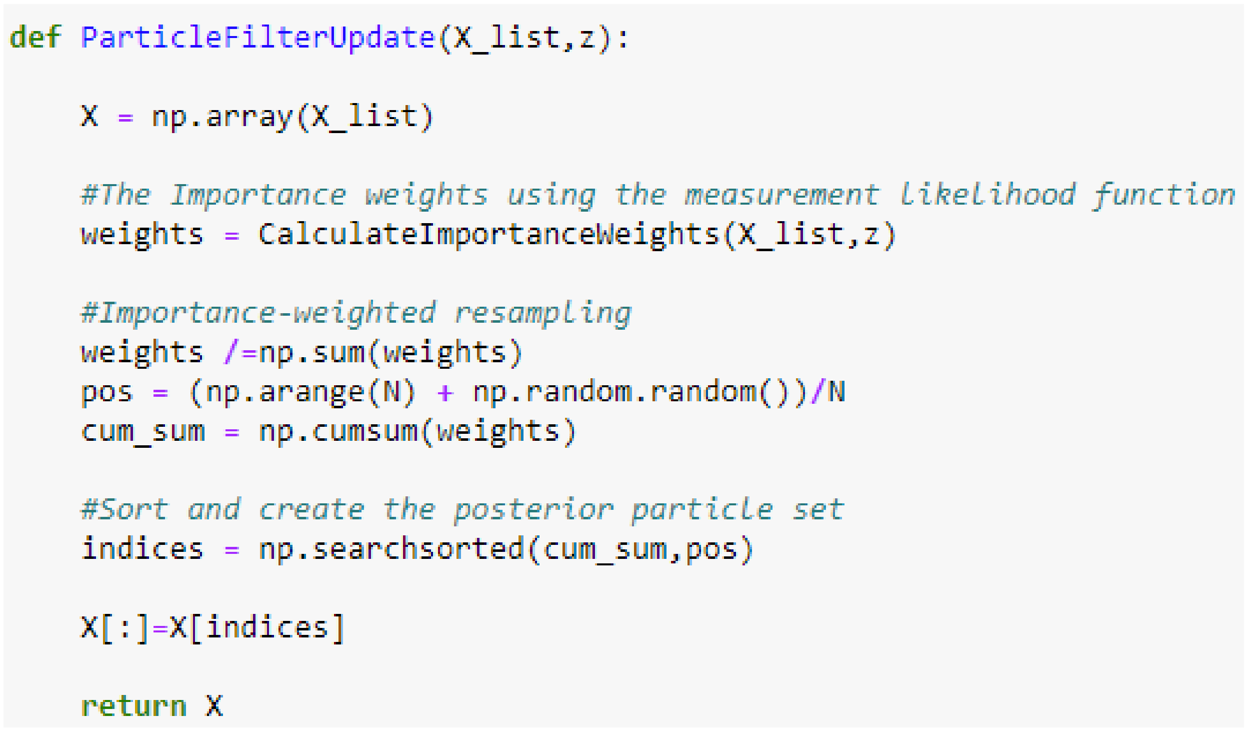
Then finally making use of these two methods defined in the Part A to model particle propagation using the following code that returns the set X_{t2} containing (N=1000) particles representing probable pose of the robot at that given time. The implementation can be seen in Part E.



```python
def ParticleFilterPropagate(t1,X_T1,t2,N):
    X_T2=[];
    for i in range(N):
        if len(X_T1)>10:
            X_2pose = add_pose(t1,X_T1[i],t2)
        else:
            X_2pose = add_pose(t1,X_T1,t2)
        X_T2.append(X_2pose)
    return X_T2
```

Part D

Using the procedure for calculating the importance weights in part B we can implement the Measurement Update step for the filter as follows:



```python
def ParticleFilterUpdate(X_list,z):

    X = np.array(X_list)

    #The Importance weights using the measurement likelihood function
    weights = CalculateImportanceWeights(X_list,z)

    #Importance-weighted resampling
    weights /=np.sum(weights)
    pos = (np.arange(N) + np.random.random())/N
    cum_sum = np.cumsum(weights)

    #Sort and create the posterior particle set
    indices = np.searchsorted(cum_sum,pos)

    X[:]=X[indices]

    return X
```

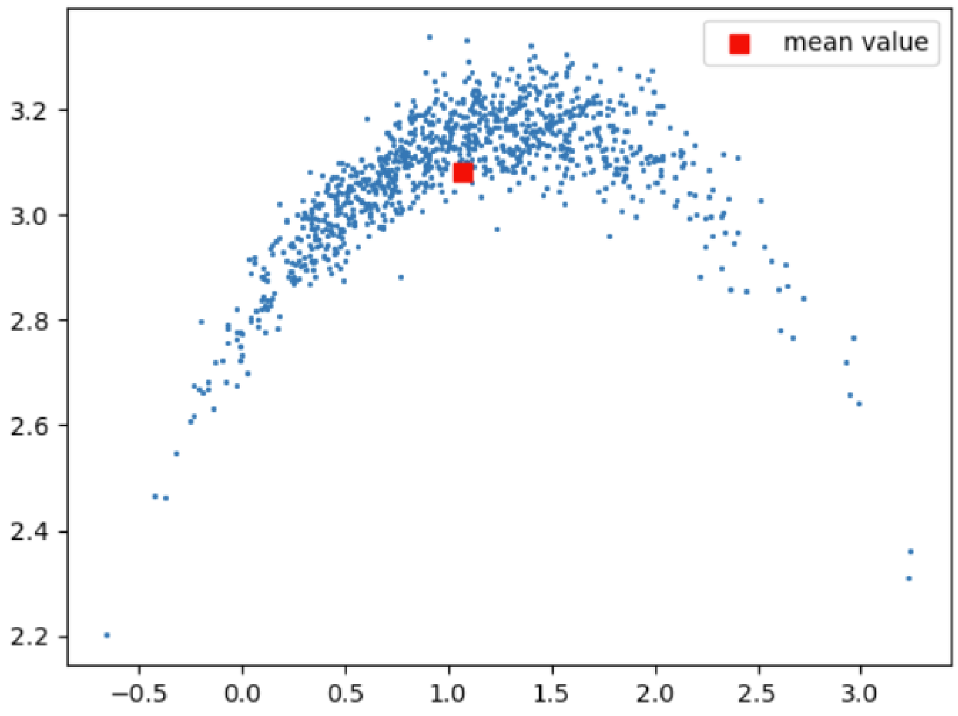
PART E

Particle propagation from t1 = 0 to t2 = 10s, was implemented using the following code:

```
################# PART E #####################################

##############################################################

X0=np.eye(3)

#Particle PROPAGATION
X10=ParticleFilterPropagate(0,X0,10,N)

#Create x data and y data
x_data,y_data=dataFROMpose(X10)

#Mean and Covariance
mean, COV = stats(x_data,y_data,N)
x_cen, y_cen = mean

#plot the position of particle
fig = plt.figure()
f1=plt.scatter(x_data,y_data,s=1.5)
f2=plt.scatter(x_cen,y_cen,s=50,marker='s',color = 'r')
plt.legend([f2], ['mean value'],markerscale=1)

print("For t = 10secs:\nMEAN:",mean,"\nCOVARIANCE: ",COV.T)
```

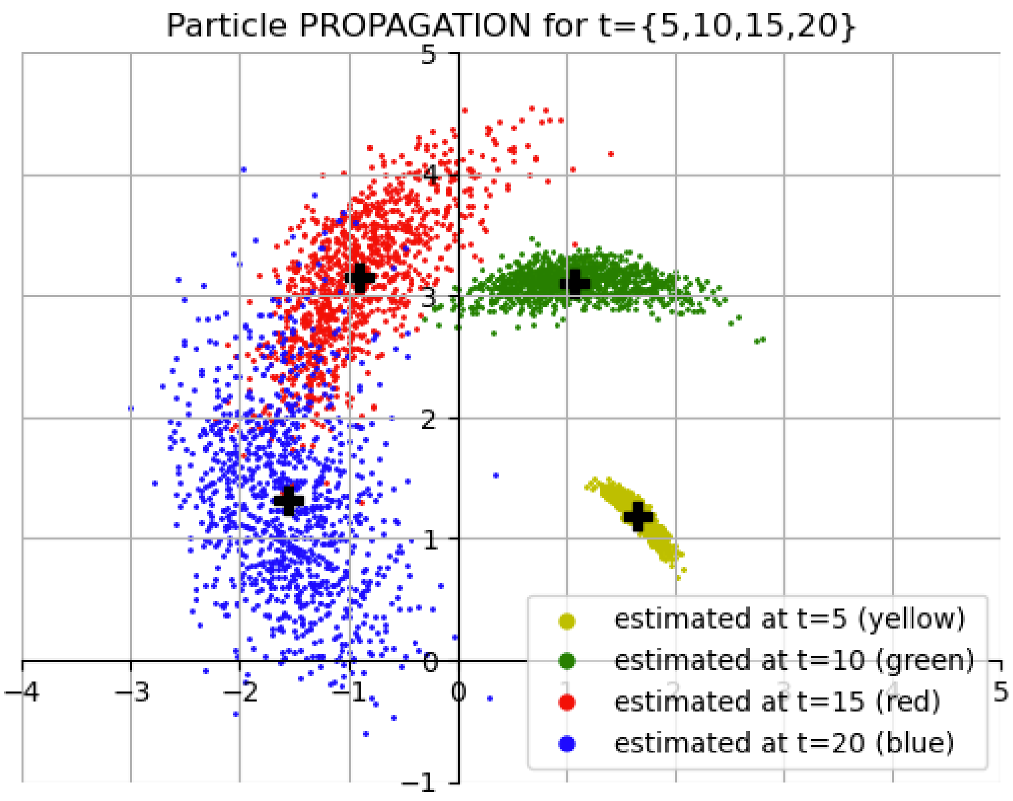
The results can be seen as follows:

```
For t = 10secs:
MEAN: (1.0662823535788726, 3.081239273877932)
COVARIANCE:  [[0.36862464 0.01752858]]
```



PART F

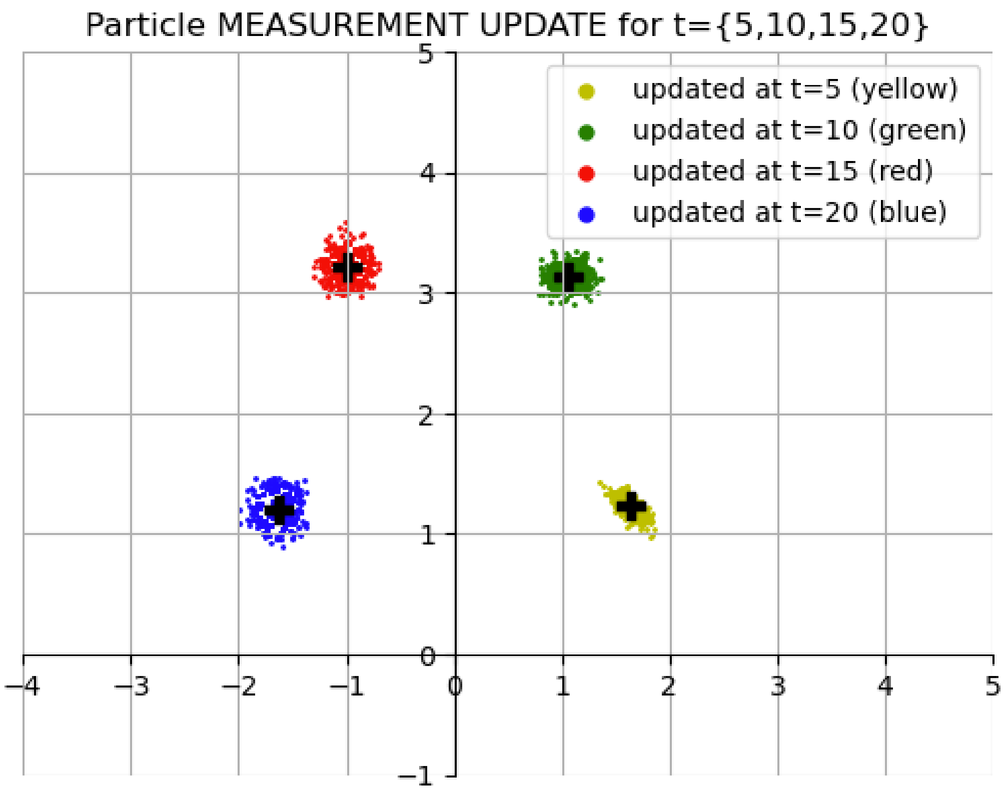
Similarly, as PART E we can implement PART F and the code can be obtained from the python notebook file for which the results are as follows:



```
For t = 5secs:
MEAN: (1.6634416888420471, 1.1843495847212036)
COVARIANCE:  [[0.01916906 0.01637278]]


For t = 10secs:
MEAN: (1.0818598316820975, 3.1013097875717173)
COVARIANCE:  [[0.25360774 0.01396287]]


For t = 15secs:
MEAN: (-0.8954210663625015, 3.153207251687587)
COVARIANCE:  [[0.29002741 0.33440281]]


For t = 20secs:
MEAN: (-1.5575322086762624, 1.318232371663429)
COVARIANCE:  [[0.243417   0.60589905]]
```

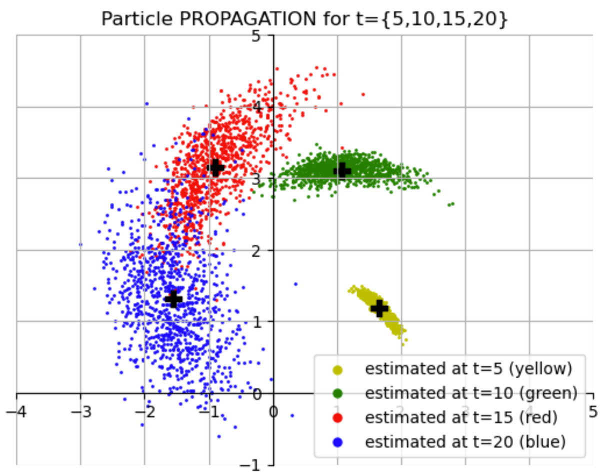
PART G

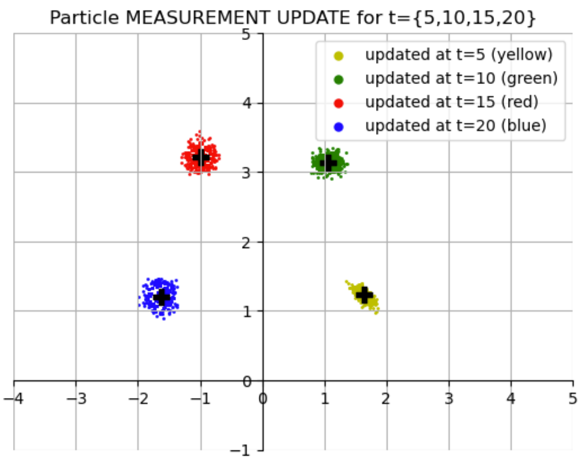
We can implement PART G and the code can be obtained from the python notebook file for which the results are as follows:



```
For t = 5secs:
MEAN: (1.6326612066153197, 1.2310074526581702)
COVARIANCE:  [[0.0049981  0.00394964]]


For t = 10secs:
MEAN: (1.0576027742469782, 3.1304123182913037)
COVARIANCE:  [[0.00856607 0.00503481]]


For t = 15secs:
MEAN: (-0.9875416283386953, 3.2103420086804855)
COVARIANCE:  [[0.00843555 0.00921587]]


For t = 20secs:
MEAN: (-1.6299531509686782, 1.192754861750899)
COVARIANCE:  [[0.01006618 0.0094587 ]]
```

We can see from the graph of Part G and F that for the given time interval t = {5,10,15,20}. The particles after propagation step are sampled quite far apart from each other. However, using the measurement update step and resampling the particles based on the weight importance we had particles close to each other. This can also be seen by comparing the Covariances for before and after resampling (Part F and Part G respectively).





NOTE:

The code can be found in the attached ParticleFilter_StateEstimation.ipynb file