Northeastern University
EECE 5550 Mobile Robotics Lab 4 submission

Yash Mewada

Nov 21,2022

1 Solution 1 - Implementation of Extended Kalman Filter

1.1 Define the discrete-time dynamical model of a robot moving with constant velocity over a 2D space

As we know that the Kalman filters work with a linear model and the main idea behind implementing the Extended Kalman filter is to linearise the non-linear motion. This will be seen further as we define the motion model.
The general formula for the Kalman filter process or prediction update is given as:

$$x_{t+1} = A_t x_t + B u_t + \epsilon; \quad \epsilon \sim \mathcal{N}(0, R_t) \tag{1}$$

$$z_t = C_t x_t + \delta_t; \quad \delta_t \sim \mathcal{N}(0, Q_t) \tag{2}$$

Noise with zero mean and Covariance matrix as R_t. Given the state vector as $x_t = \begin{pmatrix} P_x \\ P_y \\ V_x \\ V_y \end{pmatrix}$. Given constant velocity. The position of the bot changes after an interval of $\Delta t = 1$.
$(P_{x+1} - P_x)/\Delta t = V_x$ Velocity of the bot across X.
Hence the motion model can be written as:

$$\begin{pmatrix} P_{x+1} \\ V_{x+1} \\ P_{y+1} \\ V_{y+1} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ V_x \\ P_y \\ V_y \end{pmatrix} + \begin{pmatrix} 0 \\ \epsilon_{vx} \\ 0 \\ \epsilon_{vy} \end{pmatrix} \quad \sim \mathcal{N}(0, R_t)$$

$$\begin{pmatrix} P_{x+1} \\ V_{x+1} \\ P_{y+1} \\ V_{y+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ V_x \\ P_y \\ V_y \end{pmatrix} + \begin{pmatrix} 0 \\ \epsilon_{vx} \\ 0 \\ \epsilon_{vy} \end{pmatrix} \quad \sim \mathcal{N}(0, R_t) \tag{3}$$

In the above equation ϵ_{px} and ϵ_{py} are 0 due to the fact that the pose at next time step is only dependent on noise from only the velocities of that time step and not on the noise from previous position.

This is similar to $x_{t+1} = Ax_t + E$ assuming Bu = 0 because we are considering a constant velocity model, where the last matrix is the noise in the motion model. **Here A matrix is $R \in R^{4X4}$ and ϵ matrix is $R \in R^{4X1}$** is considered as Gaussian noise with R_t as its covariance matrix with is denoted as below i.e by the relation of variances of each state with all other states. It can be written as below:

As we saw earlier that the position at any time step is not dependent on the noise from previous time step, hence the R_t can be rewritten as a diagonal matrix...

$$R_t = \begin{pmatrix} \sigma_{px}^2 & \sigma_{px}\sigma_{vx} & \sigma_{px}\sigma_{py} & \sigma_{px}\sigma_{vy} \\ \sigma_{vx}\sigma_{px} & \sigma_{vx}^2 & \sigma_{vx}\sigma_{py} & \sigma_{vx}\sigma_{vy} \\ \sigma_{px}\sigma_{py} & \sigma_{py}\sigma_{vx} & \sigma_{py}^2 & \sigma_{py}\sigma_{vy} \\ \sigma_{px}\sigma_{vy} & \sigma_{vy}\sigma_{vx} & \sigma_{py}\sigma_{vy} & \sigma_{vy}^2 \end{pmatrix}$$

$$R_t = \begin{pmatrix} \sigma_{px}^2 & 0 & 0 & 0 \\ 0 & \sigma_{vx}^2 & 0 & 0 \\ 0 & 0 & \sigma_{py}^2 & 0 \\ 0 & 0 & 0 & \sigma_{vy}^2 \end{pmatrix} \tag{4}$$

1.2 Define the observation model

The general notation for the measurement model is shown in (2) from equation (2) where matrix C_t is $R \in R^{2X1}$ and δ follows Gaussian noise. Given that the robot observes two of the landmarks at each time step with the same sensor noise the C_t matrix becomes as:

$$d(l^1) = \sqrt{((l_x^1 - p_x)^2 + ((l_y^1 - p_y)^2} \tag{5}$$

$$d(l^2) = \sqrt{(l_x^2 - p_x)^2 + (l_y^2 - p_y)^2}$$

$$z_t = C_t x_t + \delta_t; \quad \delta_t \sim N(0, Q_t)$$

$$C_t = \begin{pmatrix} d(l^1) \\ d(l^2) \end{pmatrix} \sim \mathcal{N}(0, Q_t)$$

$$d(l^1) \sim \mathcal{N}(0, \sigma_p)$$

$$d(l^2) \sim \mathcal{N}(0, \sigma_p) \tag{6}$$

$$Q_t = \begin{pmatrix} \sigma_p^2 & 0 \\ 0 & \sigma_p^2 \end{pmatrix}$$

$$x_t = \begin{pmatrix} P_x \\ V_x \\ P_y \\ V_y \end{pmatrix}$$

Also the measurement model has independent co variances, the C_t matrix will also follow the same. Here it is seen that the measurement model becomes non-linear as matrix C_t and matrix x_t cannot be computed due to their size. Hence we use Jacobians to convert this non-linear matrix C_t into a linear state. It will be seen in the next answer.

1.3 Define the Jacobians of the models to obtain the linearized models

$$J = \begin{pmatrix} \frac{\partial d(l^1)}{\partial P_x} & \frac{\partial d(l^1)}{\partial V_x} & \frac{\partial d(l^1)}{\partial P_y} & \frac{\partial d(l^1)}{\partial V_y} \\ \frac{\partial d(l^2)}{\partial P_x} & \frac{\partial d(l^2)}{\partial V_x} & \frac{\partial d(l^2)}{\partial P_y} & \frac{\partial d(l^2)}{\partial V_y} \end{pmatrix} \tag{7}$$

Here $d(l^i)$ determines the two landmarks given. Based on this formula the Jacobian matrix reduces to:

$$J = \begin{pmatrix} A & 0 & B & 0 \\ C & 0 & D & 0 \end{pmatrix}$$

$$J = \begin{pmatrix} (P_x - l_x^1)/d(l^1) & 0 & (P_y - l_y^1)/d(l^1) & 0 \\ (P_x - l_x^2)/d(l^2) & 0 & (P_y - l_y^2)/d(l^2) & 0 \end{pmatrix} \tag{8}$$

Here $d(l^1)$ and $d(l^2)$ are equations from (5) and where A,B,C,D are:

$$\text{A} = -(l_x^1 - P_x).\{(l_x^1 - p_x)^2 + (l_y^1 - p_y)^2\}^{-1/2}$$

$$\text{B} = -(l_x^1 - P_y).\{(l_x^1 - p_x)^2 + (l_y^1 - p_y)^2\}^{-1/2}$$

$$\text{C} = -(l_x^2 - P_x).\{(l_x^2 - p_x)^2 + (l_y^2 - p_y)^2\}^{-1/2}$$

$$\text{D} = -(l_x^2 - P_y).\{(l_x^2 - p_x)^2 + (l_y^2 - p_y)^2\}^{-1/2}$$

(8) is the most simplified form of the Jacobian Matrix.

1.4 Apply the propagation and update steps of EKF and write the updates of the covariance

The propagation step for EKF can be described as below:
$X_t \sim \mathcal{N}(\mu_t, \Sigma_t)$.

$$\mu_t = \begin{pmatrix} \mu_{px} \\ \mu_{vx} \\ \mu_{py} \\ \mu_{vy} \end{pmatrix}$$

$$\Sigma_t = \begin{pmatrix} \sigma_{px}^2 & 0 & 0 & 0 \\ 0 & \sigma_{vx}^2 & 0 & 0 \\ 0 & 0 & \sigma_{py}^2 & 0 \\ 0 & 0 & 0 & \sigma_{vy}^2 \end{pmatrix} \tag{9}$$

$$X_{t+1} = A_t + \Sigma_t, \Sigma \sim \mathcal{N}(0, R_t)$$

$$X_{t+1} \sim \mathcal{N}(A_t\mu_t, A_t\Sigma_t A_t^t) + \mathcal{N}(0, R_t)$$

$$X_{t+1} \sim \mathcal{N}(A_t\mu_t, A_t\Sigma_t A_t^T + 0, R_t) \tag{10}$$

The above derivation refers to the propagate update model of the EKF. Next we derive the measurement model for the same.

$$k_{t+1} = \hat{\Sigma_{t+1}} C_{t+1}^T \{C_{t+1}\hat{\Sigma_{t+1}} C_t^T + 1 + Q_{t+1}\}$$

$$\mu_{t+1} = \hat{\mu_{t+1}} + k_{t+1}\{z_{t+1} - C_{t+1}\hat{\mu_{t+1}}\} \tag{11}$$

$$\Sigma_{t+1} = \{I - k_{t+1}C_{t+1}\}^{-1}$$

Using the standard form for the extended kalman filter.

2 Solution 2 - Scan matching using iterative closest point

In practice, Iterative Closest Point provides us with the highest precision of aligning the two point clouds of the same features registered by the laser scanner from different positions of the robot. This process is done iteratively.
** Note the final and here listed code blocks might be different due to the implementation of classes.**

2.1 Implement a function Estimate Correspondences

The below code presents the implementation of the required function. For the first iteration, it returns 3299 corresponding points i.e for every 3299 X there is a Y point in the given point clouds.
Here the given point clouds and other functions are done using NumPy matrix operations for easy and faster output.

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

Listing 1: Importing required libraries

```
  def EstimateCorrespondences(X,Y,t,R,dmax):
      """
      Args:
          X: X point cloud
          Y: Y point cloud
          t: Optimal translation you found using SVD decomposition. For the initial guess, we consider it as 0.
          R: Optimal rotation you found using SVD decomposition. For initial guess we consider it as I3 i.e no rotation is there between the points.
      Returns:
          c: A list of tuples with indices of the corresponding X and Y points from the point clouds.
      """
      c = []
      x = np.transpose(X) # Transpose the point clouds for easy matrix calculations
      y = np.transpose(Y)
      for i in range(X.shape[0]):
          xp = x[:,i]
          yj = np.linalg.norm(y - (np.dot(R,xp) + t),axis=0) #calculate the norm/ euclidean distance between y and R.x+t
          yind = np.argmin(yj) # get the index of the minimum norm from those calculated
          if (yj[yind] ¡ dmax):
              c.append((i,yind)) # append those indices to the corresponded list.
              self.xlist.append(X[i])
              self.ylist.append(Y[yind])
      return c
```

Listing 2: Function Estimate Correspondences

2.2 Implement the function Compute Optimal Rigid Registration

Based on the corresponding point indices from the function Estimate Correspondences it will extract those indices X and Y points from the given point clouds and will perform mean and deviation of those points. The cross-covariance matrix is calculated which will further be decomposed with help of Single Value Decomposition.

```
   def ComputeOptimalRigidRegistration(X,Y,corresponds):
       """
       Args:
           X: X point cloud
           Y: Y point cloud
           corresponds : The list of corresponded points you found from EstimateCorrespondences function
       Returns:
           trans: Optimal translation from the corresponded points
           rot: Optimal rotation from the corresponded points
       """
       Xcentroid = np.matrix([0.,0.,0.]) #declare empty float matrices for centroids
       Ycentroid = np.matrix([0.,0.,0.])

       Xcentroid = np.mean(self.xlist,axis=0) #calculate mean of the found corresponded points from the cloud
       Ycentroid = np.mean(self.ylist,axis=0)

       sdx = np.subtract(self.xlist,Xcentroid) # calculate their deviation from centroid
       sdy = np.subtract(self.ylist,Ycentroid)

       for i in range(len(corresponds)):
           self.W += np.dot(sdy[i,:].T,sdx[i,:]) # calculate the cross covariance matrix

       self.W = self.W/len(corresponds)

       U,˙,vt = np.linalg.svd(self.W) #decompse the found cross covariance matrix
       rot = np.dot(U,vt) #calculate rotation and optimal translation from SVD
       trans = Ycentroid.T - (np.dot(rot,Xcentroid.T))

```

```
        return (trans,rot) #return those translations and rotational matrices
```

Listing 3: Compute Optimal Rigid Registration

2.3 Write a function that implements the ICP algorithm

The below-mentioned snippet shows the implementation of the ICP algorithm as Compute function.

```
        def compute(X,Y,t,R,dmax,num_ICP_iters):
        """
        Args:
            X: X point cloud
            Y: Y point cloud
            t: Optimal translation you found using SVD decomposition. For initial guess we consider it as 0.
            R: Optimal rotation you found using SVD decomposition. For initial guess we consider it as I3 i.e no roation is there between the points.
            dmax: The maximum overlapping distance.
            num_ICP_iters: Number of iterations the ICP should run.
        Returns:
            tr: Optimal translation from the corresponded points
            ro: Optimal rotation from the corresponded points
            cor: A list of tuples of indices of final corresponded points.
        """
        tr = t
        ro = R
        for i in range(num_ICP_iters):
            cor = self.EstimateCorrespondences(X,Y,tr,ro,dmax)
            tr,ro = self.ComputeOptimalRigidRegistration(X,Y,cor)
            self.xlist = [] #setting the global lists and matrices to empty.
            self.ylist = []
            self.W = np.matrix([[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])

            # USED FOR GENERATING GIF FROM EACH TRANSFORMATIONS BETWEEN THE CLOUDS
            # icx = np.dot(ro,self.X.T) + tr
            # icx = icx.T
            # fig = plt.figure()
            # ax = fig.add_subplot(111, projection='3d')
            # ax.scatter(self.Y[0:,0], self.Y[0:,1], self.Y[0:,2], c='r', marker='o',s=0.5)
            # ax.scatter(icx[0:,0], icx[0:,1], icx[0:,2], c='b', marker='^',s=0.5)
            # ax.set_xlabel('X Label')
            # ax.set_ylabel('Y Label')
            # ax.set_zlabel('Z Label')
            # filename = str(i)+'plot.png'
            # fig.savefig(filename)
            # plt.close(fig)

        return tr,ro,cor
```

Listing 4: Compute Optimal Rigid Registration

2.4 Implement the ICP algorithm based on given parameters

Below is the implementation of ICP under the given conditions.

```
if __name__ == "__main__":
    x = np.loadtxt("C:/Users/yashm/Downloads/pclX.txt", dtype=float)
    xpcd = np.matrix(x)

    y = np.loadtxt("C:/Users/yashm/Downloads/pclY.txt", dtype=float)
    ypcd = np.matrix(y)

    icp = IterativeClosedPoint(xpcd,ypcd)

    R = np.matrix([[1,0,0],[0,1,0],[0,0,1]])
    t = np.matrix([[0],[0],[0]])

    start = time.time()
    dmax = 0.25
    num_iters = 30
    trans,rot,cor = icp.compute(xpcd,ypcd,t,R,dmax,num_iters)
```

```
    print("The ICP took %s seconds to compute" % (time.time()-start))
    icx = np.dot(rot,xpcd.T) + trans

    RMSE = 0
    for i,j in cor:
        x = xpcd[i].T
        y = ypcd[j].T
        y_norm = np.linalg.norm(y - (np.dot(rot,x) + trans))
        RMSE+=y_norm
    RMSE= np.sqrt(RMSE/len(cor))

    print("RMSE: ",RMSE)
    print("Translation Matrix: ",trans.T)
    print("Rotation Matrix: ",rot)
    print("Lenght of Correspondences: ",len(cor))

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    ax.scatter(icx[0], icx[1], icx[2], c='r', marker='^',label = 'Moving',s = 0.5)
    ax.scatter(ypcd[:,0], ypcd[:,1], ypcd[:,2], c='b',marker='^',label = 'Fixed',s = 0.5)
    ax.legend()
    # ax.scatter(xpcd[:,0], xpcd[:,1], xpcd[:,2], c='g',s=0.5)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    plt.show()
```

Listing 5: Compute Optimal Rigid Registration

Based on the given formula to calculate the Root Mean Square Error for the final transformation, the value converges to 0.008265770571572737 for 30 iterations.

```
The ICP took 18.58958888053894 seconds to compute

RMSE:  0.008265770571572737

Translation Matrix:  [[ 0.49788282 -0.29587521  0.29742624]]

Rotation Matrix:  [ 0.95141847 -0.15327827 -0.26703682]
                  [ 0.22538643  0.93757762  0.26485687]
                  [ 0.20977095 -0.31217619  0.92657551]
Length of Correspondences:  5750
```

Listing 6: Final output of the code

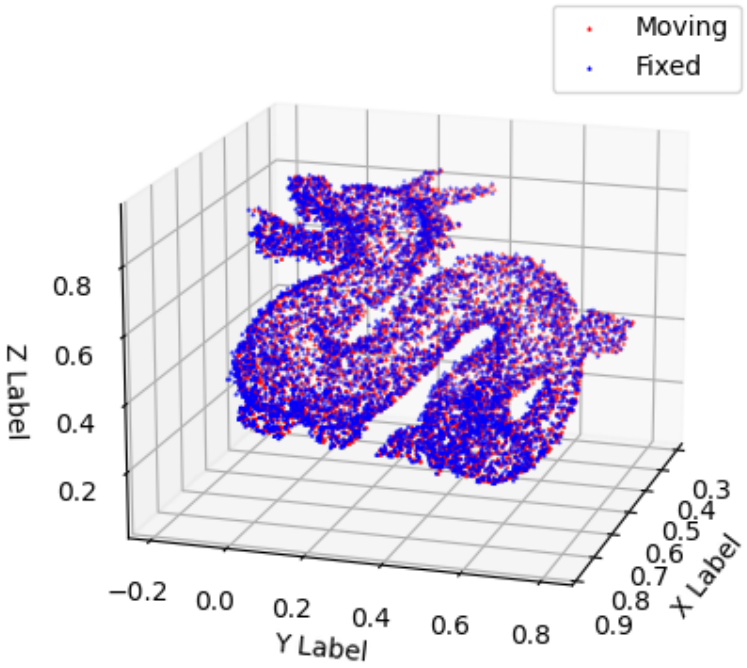
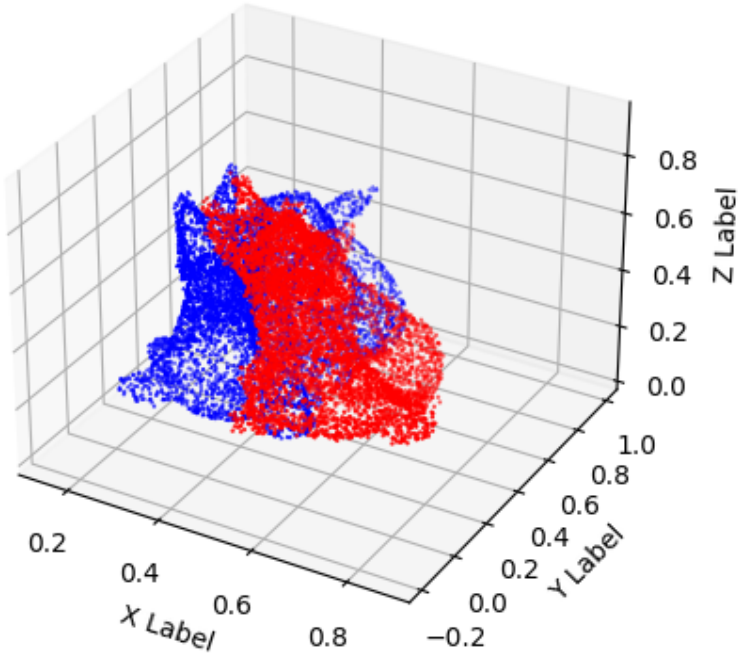


Fig 1: Output of ICP under 30 iterations

** The below GIF will only work on Adobe Acrobat.**



vid: 1 Output of ICP in real-time. Click on the GIF to play

3 Solution 3 State estimation by particle filtering on Lie Groups

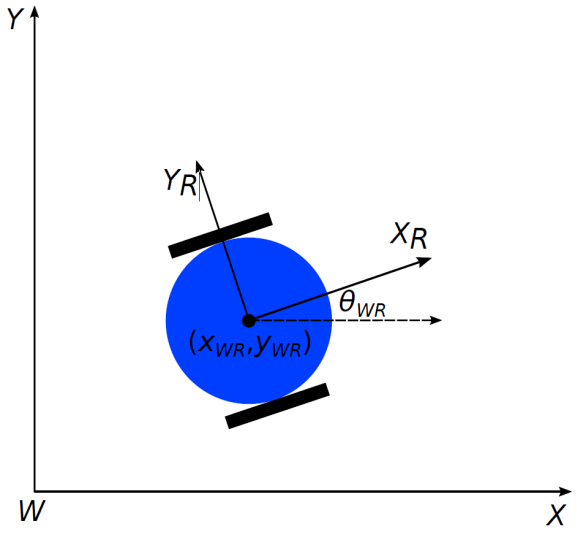


Fig 2: Schematic for differential drive robot.

As we saw in HW2 that for a constrained differential drive robot as shown above with given r as its radius of wheels, ϕ_r and ϕ_l as the wheel speeds of the left and right wheels.

3.1 Derive a generative motion model

Based on the figure above the equations for the motion model can be written as below:

$$\begin{pmatrix} x_w^\bullet r \\ y_w^\bullet r \\ \theta_w^\bullet r \end{pmatrix} = \begin{pmatrix} R(\theta) & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} r/2(\phi_r^\bullet + \phi_l^\bullet) \\ 0 \\ r/\omega(\phi_r^\bullet - \phi_l^\bullet) \end{pmatrix} \tag{12}$$

The (12) suggests that the pose of the robot at any given time is a function of its wheel velocities as input. As the kinematic equation (12) describes how the left and right wheel velocities define the motion of the robot over the Lie group of Lie(SE(2)).

$$\Omega^\bullet : R^2 \to Lie(SE(2)). \tag{13}$$

The derivative map can be written in a simplified way as below:

$$\Omega^\bullet(\phi_l^\bullet, \phi_r^\bullet) = \begin{pmatrix} 0 & -r/\omega(\phi_r^\bullet - \phi_l^\bullet) & r/2(\phi_r^\bullet + \phi_l^\bullet) \\ r/\omega(\phi_r^\bullet - \phi_l^\bullet) & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{14}$$

The velocity field on Lie(SE(2)) is determined by the robot's commanded left and right wheel speeds is the $\boldsymbol{left-invariant}$ vector field determined by the Lie group element from (14). The motion model of the robot given its imprecise wheel velocities will be equal to the integral curve of the vector field we derived in HW 2. This can be rewritten as:

$$\gamma(t) = X_0 exp(t\Omega^\bullet(\phi_l^\bullet, \phi_r^\bullet)) \tag{15}$$

Putting (14) in (15) we get the motion model i.e the pose of the robot at t_2 as a function of its pose at t_1 (initial pose) given the commanded wheel speeds with wheel radius r, track width w and variances σ_l and σ_r.

$$X_{t1}(t) = X_0 exp(t_1\Omega^\bullet(\phi_l^\bullet, \phi_r^\bullet)) \tag{16}$$

$$X_{t2}(t) = X_0 exp(t_2\Omega^\bullet(\phi_l^\bullet, \phi_r^\bullet)) \tag{17}$$

In (15) it describes the trajectory of the robot at any given time $\boldsymbol{(t)}$ is dependent on the initial position of the robot i.e in our case X_0 is the origin and the exponential of the derivative map from (14).
Given the noise in the wheel speeds as ϵ_l and ϵ_r with $\mathcal{N}(0, \sigma_l^2)$ and $\mathcal{N}(0, \sigma_r^2)$.
Hence the overall noise of the system after incorporating the motion model will be equal to:

$$\gamma(t) = X_0 exp(t\Omega^\bullet(\phi_l^\bullet, \phi_r^\bullet)), \epsilon \sim \mathcal{N}(0, \sigma_l^2 + \sigma_r^2) \tag{18}$$

where $\epsilon \sim \mathcal{N}(0, \sigma_l^2 + \sigma_r^2)$ is the noise in the model with mean as 0 and standard deviation as $\sigma_l^2 + \sigma_r^2$.

3.2 Derive a closed loop formula for the measurement likelihood function, $\mathbf{p}(z_t|x_t)$

In the general case the probability function is considered to be a scalar vector, and a common density function is given by a Gaussian Function as:

$$p(x_t) = (2\pi\sigma^2)exp(-1/2((x-\mu)^2/\sigma^2)) \tag{19}$$

But in our case the noisy GPS positions received are multidimensional and the noise $\epsilon_p \sim \mathcal{N}(0, \sigma_p^2 I_2)$ and recalling it back from lecture 4 on review of Probability the measurement model can be defined as a Probability Density Function over a Multivariate Gaussian Distribution and the reason for that is the GPS data we receive is fused with X and Y coordinates of the robot. Also, the basic motive behind using the Particle Filter is that the models are assumed to be non-linear and non-Gaussian.

$$z_t = l_t + \epsilon_p \tag{20}$$

$$\epsilon_p = z_t - l_t \ , \ \epsilon_p \sim \mathcal{N}(0, \sigma_p^2 I_2) \tag{21}$$

$$p(z_t|x_t) = \frac{1}{(det(2\pi\sigma_p^2 I_2))^{1/2}} exp\{(\frac{1}{2\sigma_p^2}(z_t - l_t)^T I_2^{-1}(z_t - l_t))\} \tag{22}$$

$$\Sigma = \sigma_p^2 I_2$$

(22) describes the measurement likelihood function and where $\Sigma = \sigma_p^2 I_2$ is the covariance of the function. As in the below answers we will see that for the initial case the Particle filter will make a belief over its position making use of the wheel speeds and approximating its position based on the motion model we derived in (15).
Then it again makes its belief over the position via the measurement model using the information from the sensors, here the GPS sensor with noise. Here the filter works in a way that the less probable i.e the particles whose weights are less are been decayed and that same amount of particles are replaced in position with the ones with higher weight.

3.3 Write a function to implement Particle Filter Propagate Step

For this, we first have to define a motion model which we derived in (15).

```
    def motion_model(self,t2,u,r,w):
        """
        Args:
            t2: The time at which we want to find the position of bot.
            u:  The commanded wheel speeds at time t2.
            r:  radius of wheels.
            w:  Width of track
        Returns:
            motion_model: The final transformation matrix based on the input command at
    desired time.
        """
        manifold_map = np.matrix([[0.,-r*(u[0] - u[1])/w,r*(u[0] + u[1])/2],
                                  [r*(u[0] - u[1])/w ,0.,0.],
                                  [0.,0.,0.]])
        motion_model = self.x_init@expm(t2*manifold_map) #x_init is the initial position of
    the bot. Here as identity matrix of size 3.
        return motion_model
```

Listing 7: Compute motion model for the bot.

```
    def ParticleFilterPropagate(self,t1,particles,phi_l,phi_r,t2,r,w,sigma_r,sigma_l):
        """
        Args:
            t1: Current time.
            particle set: Initial particle set which describes robot belief at time t = 0.
                  This is similar to p(x1).
            phi_r: commanded right wheel speed
            phir_l: commanded left wheel speed
            t2: Time at which you want to propagate the robot's belief provided the wheel
    speeds of time t2.
            r: radius of wheels.
            w: Width of track
            sigma_r: variance of right wheel speed
            sigma_l: variance of left wheel speed
        returns:
            pose_true: True position of the robot based on the commanded wheel speeds without
    noise.
            particles: belief the robot thinks it is at time t2 based on the noise
               distribution in the wheel speeds
        """
        pose_true = self.motion_model(t2,(phi_r,phi_l),r,w)
        pose_true = np.array((pose_true.item((0,2)),pose_true.item((1,2)))) # extract the X
    and Y coordinates from the transformation

        my_list =np.zeros((self.num_particles,2))
        ud1 = u[0] + np.random.normal(loc=0,scale = sigma_r,size = self.num_particles)
        ud2 = u[1] + np.random.normal(loc=0,scale = sigma_l,size = self.num_particles)
        my_list[:,0] += ud1
        my_list[:,1] += ud2

        i = 0
```

```
        for ir,jl in my_list:
            pose = self.motion_model(t2,(ir,jl),r,w)
            particles[i,0] = pose.item((0,2))
            particles[i,1] = pose.item((1,2))
            i += 1

        return pose_true,particles
```

Listing 8: Compute Particle Filter Propagate step

Given $\sigma_r = 0.05$ and $\sigma_l = 0.05$.

3.4 Write a function to implement Particle Filter Update Step

Here as we are considering the importance of weighted re-sampling. Our weight of the particles is proportional to the multivariate Gaussian distribution we saw in (22).
Here the mean is the particle sets for X and Y individually whose PDF (Probability Density Function) is also over the measurements from GPS. The variance for the GPS noise is also given as $\sigma_p = 0.1$
Also here it is given that the position X and Y are independent and their covariance matrix is a diagonal matrix hence we can directly multiply their Gaussian samples which is similar to line 10 in the below code. But if in any case X and Y are dependent then we cannot apply this step and we have to make use of dependent X and Y multivariate Gaussian sampling which we saw in (22) provided we are given the co variances of σ_{xy} and σ_{yx} which in our case are zero as the covariance is multiplied with I_2 as seen in (21)

```
    def ParticleFilterUpdate(self,particles,z,sigma_p):
        """
        Args:
            particles: Initial particle set which describes robot belief. This is similar to p
    (x1).
            z: noisy measurement from GPS at time t2.
            sigma_p: given variance of the noisy GPS measurement. In practice this is replaced
     by empty set of weighs.
        returns:
            particles: The particle set based on the posterior belief given the measurement
    from sensor.
        """
        weights.fill(1.) # initialise weights with 1
        for i in range(len(z)):
            weights *= scipy.stats.norm(particles[:,i],self.sigma_p).pdf(z[i]) # find the
    weights of each particle based on their distribution and posterior belief.

        weights /= sum(weights) #normalise the weights i.e p(z).

        positions = (np.arange(N) + np.random.random()) / N # sample the random position with
    size equal to that of weights
        indexes = np.zeros(N, 'i') #initialise indexes as int datatype.
        cumulative_sum = np.cumsum(weights) # find the cumulative sum of all weights.
        i, j = 0, 0
        while i < self.num_particles and j < self.num_particles:
            if positions[i] < cumulative_sum[j]: #if the sampled position index is less than
    cumulative sum of that index, then add that index.
                indexes[i] = j
                i += 1
            else:
                j += 1

        particles[:] = particles[indexes] # replace that particles with sampled indexes
        return particles
```

Listing 9: Compute the Particle Filter propagate step

3.5 Generate 1000 realisations of particles at time t = 10 and calculate their empirical mean and covariance

Based on the previous sections in which we developed the particle filter propagate step and motion models. We use the same functions to generate 1000 particles based on the commanded wheel speeds.

```
if ``name`` == ```main```:
N = 1000
px = np.zeros((N, 2))  # Generate empty particles

particlefilter = Particle˙Filter(N) #initialise particle filter class

phi˙l = 1.5 #robot parameters and variances in wheel speeds
phi˙r = 2.0
w = 0.5
r = 0.25
u = [phi˙r,phi˙l]

x,pose = particlefilter.ParticleFilterPropagate(0,px,u,10,r,w) #run particle filter
propagate step

mean = np.average(pose,axis=0)
cov = np.average((pose-mean)**2,axis=0)
print(”Empirical Mean:”,mean)
print(”Covariance:”,cov)

fig = plt.figure()
ax = fig.add˙subplot()
ax.scatter(pose[:,0], pose[:,1],label = ’Particles’)
ax.scatter(x[0], x[1],label = ’Bot position based on motion model’)
ax.legend()
plt.xlim([0,10])
plt.ylim([0,10])
plt.show()
```

Listing 10: Generate 1000 particles based on the commanded velocities for time t

Below is the output for that where we can see that the particles are been approximated near to the true value but with too much of variance between them as it has no input from the measurement model whether their belief is true or not.

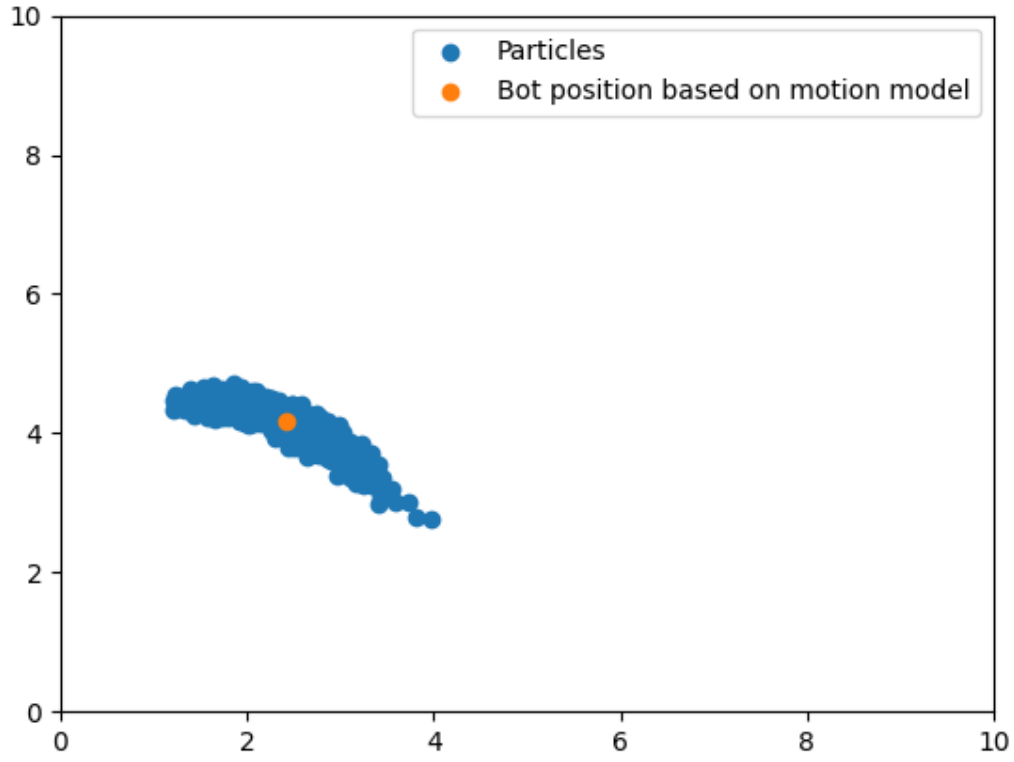


Fig 3: Output for 1000 particles at t=10

```
Empirical Mean: [1.07210353 3.07641702] for t:10

Covariance: [0.38878701 0.03858074]
            [0.03858074 0.01976835] for t:10
```

Listing 11: Empirical mean and co variances at t

As from the below image it is evident that the propagate step approximates the position of particles in a banana manner due to the fact that there is impreciseness in the wheel speeds and the robot can end up in any given position where the particles are distributed. Also when we calculate the mean it suggests that the robot is drifted from its original position which is termed as dead reckoning.

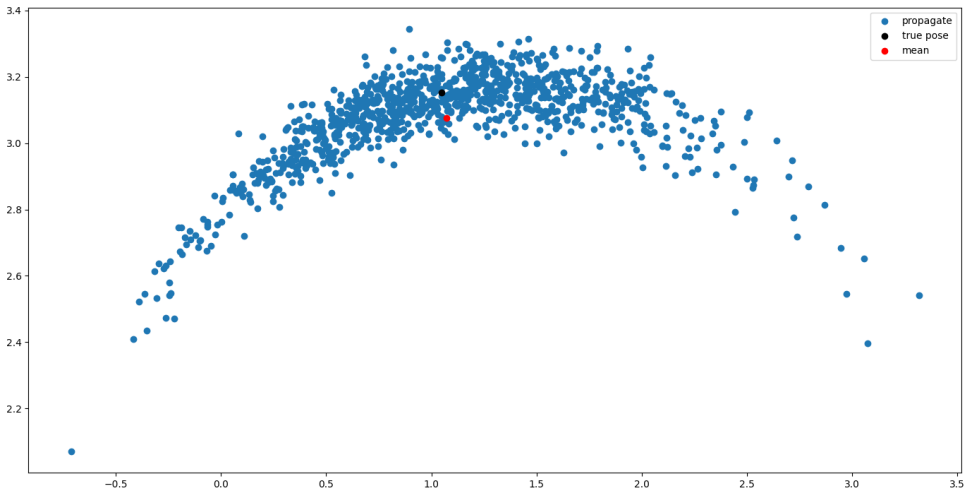


Fig 4: Output for 1000 particles at t=10

3.6 Simulate the evolution of differential drive robot's belief over its position with dead reckoning

```
if ``name`` == ```main```:
N = 1000 #number of particles to compute.
px = np.zeros((N, 2))  # Particle store

particlefilter = Particle˙Filter(N)
phi˙l = 1.5
phi˙r = 2.0
w = 0.5
r = 0.25
u = [phi˙r,phi˙l]

samp˙t = np.arange(5,25,5) # create a list of time from 0-25 with 5 spacing

fig = plt.figure()
ax = fig.add˙subplot()
for i in samp˙t:
    x,pose,init˙particles = particlefilter.ParticleFilterPropagate(0,px,u,i,r,w)
    ax.scatter(pose[:,0], pose[:,1],s=0.5)
    ax.scatter(x[0], x[1],c='black')

    filename = 'pf-' + str(i) +'.png'
    fig.savefig(filename)
    plt.close(fig)
    mean = np.mean(pose,axis=0)
    diff = np.stack((pose[:,0],pose[:,1]),axis = 0)
    cov = np.cov(diff)
    print("Empirical Mean:",mean,"for t:",i)
    print("Covariance:",cov,"for t:",i)
```

```
    plt.show()
```

Listing 12: simulating the evolution of robot's belief

Below is the output of the simulation for time t = 0 to time t = 20 for every 5 seconds.Black dots suggest the position of robot based on motion model

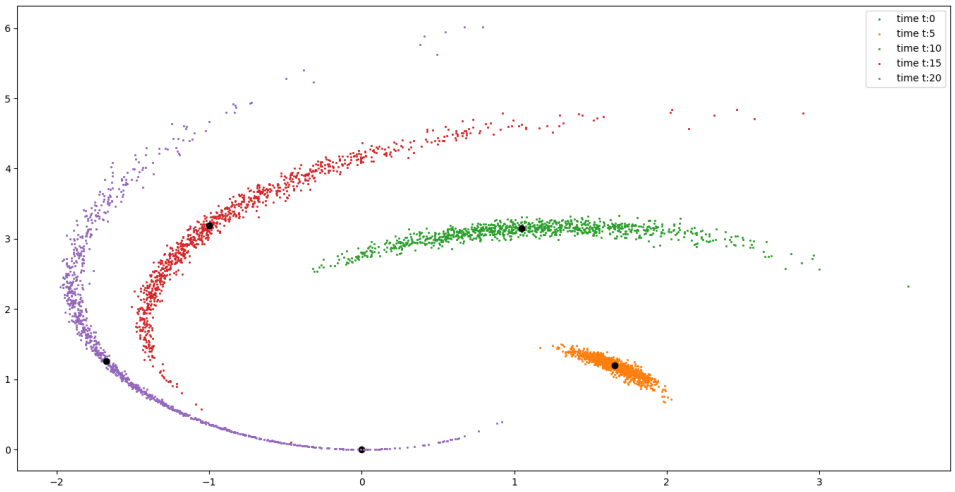
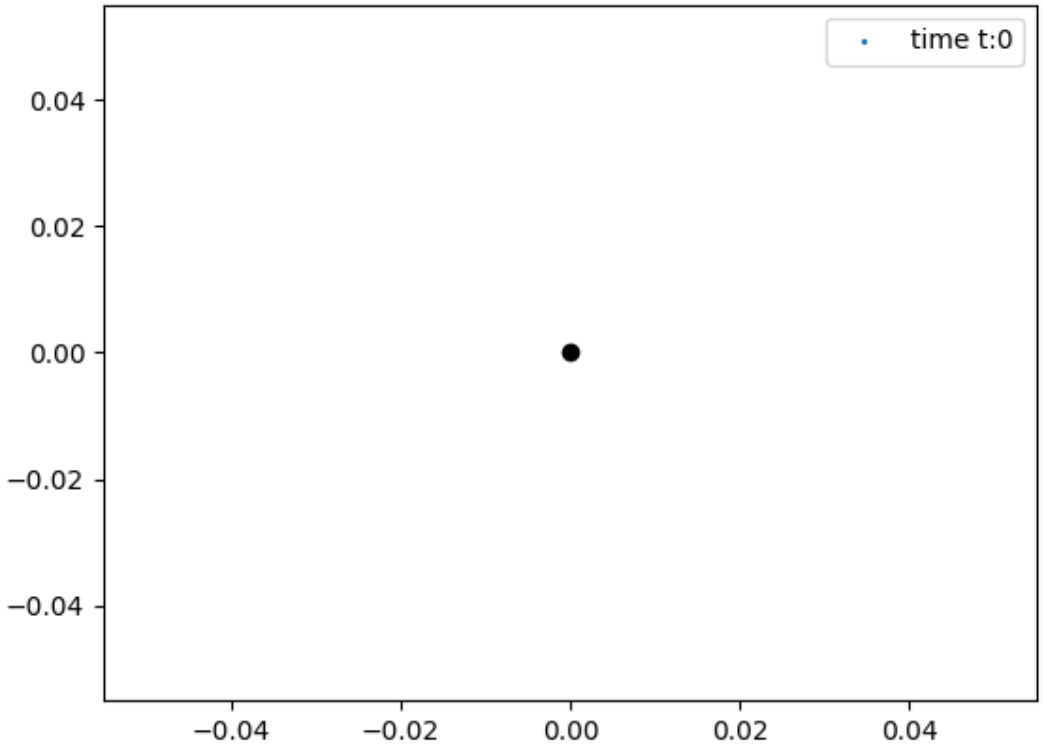


Fig 5: Output of simulation of particles over the belief in robot's position.

```
    Empirical Mean: [1.65447726 1.18838215] for t: 5
    Covariance: [[ 0.0196078  -0.0159895 ]
                 [-0.0159895   0.01607992]] for t: 5
    Empirical Mean: [1.08441615 3.0797753 ] for t: 10
    Covariance: [[0.36107712 0.03349055]
                 [0.03349055 0.01787833]] for t: 10
    Empirical Mean: [-0.79379173  3.09628258] for t: 15
    Covariance: [[0.48842691 0.52834526]
                 [0.52834526 0.80433318]] for t: 15
    Empirical Mean: [-1.33938906  1.48486256] for t: 20
    Covariance: [[ 0.35752184 -0.35012872]
                 [-0.35012872  1.60825227]] for t: 20
```

Listing 13: Empirical mean and co variances for different time



Vid: 2 Output of particle filter propagate step from t=0-25. Click on the GIF to play

From the above output, we can see that over time the propagate step evolves the belief of the robot's position, and based on the given data the particles scatter over their belief. This represents the dead reckoning effect in particles. We will see in the next output that the position of the robot truly drifted from its true position over time.

3.7 Apply particle filter Propagate and Update step to generate approximations

Based on the particle filter propagate and update functions we wrote above gives us an best approximation about robot's position after measurement.

```
    if ``name`` == ``main``:
    N = 1000
    px = np.zeros((N, 2))  # Particle store
    pw = np.array([1.0]*N) #initialise empty particle weights

    particlefilter = Particle˙Filter(N)
    phi˙l = 1.5
    phi˙r = 2.0
    w = 0.5
    r = 0.25
    u = [phi˙r,phi˙l]

    samp˙t = np.arange(5,25,5)
    zs = [(1.6561,1.2847),(1.0505,3.1059),(-0.9875,3.2118),(-1.6450,1.1978)]

    fig = plt.figure()
    ax = fig.add˙subplot()
    ax.legend()
    for i,f in zip(samp˙t,zs):
        x,pose,init˙particles = particlefilter.ParticleFilterPropagate(0,px,u,i,r,w) #
    propagate step
        new˙particles = particlefilter.ParticleFilterUpdate(pose,f,pw) #measurement update
    step
```

```
        mean = np.mean(pose,axis=0)
        std = np.stack((pose[:,0],pose[:,1]),axis = 0)
        cov = np.cov(std)
        print("Emperical Mean:",mean,"for t:",i)
        print("Covariance:",cov,"for t:",i)

        ax.scatter(new_particles[:,0], new_particles[:,1],label = 'update at t:' + str(i),s =
    0.5)
        ax.scatter(x[0], x[1],c='black',label='Robot Pos')
        ax.scatter(0, 0,c='black')
        ax.scatter(f[0], f[1],c='r',label='GPS')
        ax.legend()

    plt.show()
```

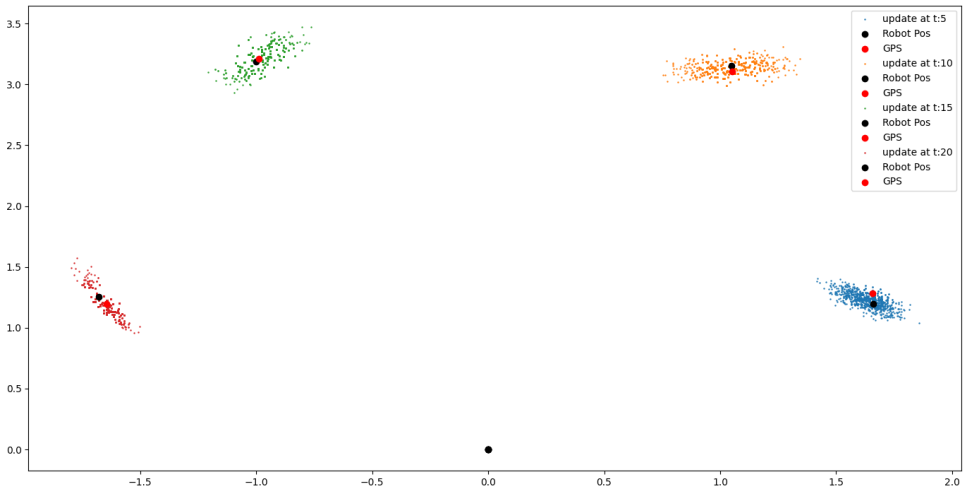


Fig 6: Output measurement update with prior information of the update step.

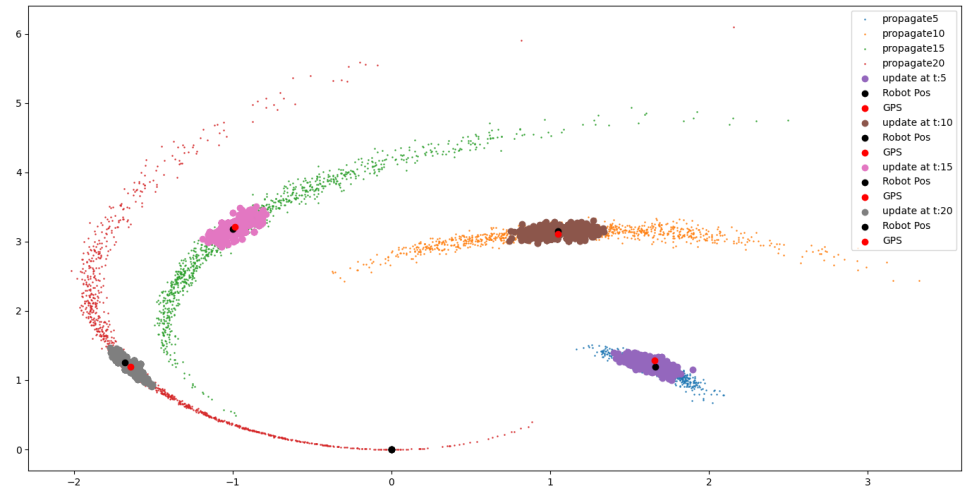


Fig 7: Output of difference between measurement step and propagate step.

```
Empirical Mean: [1.63019377 1.23683014] for t: 5
Covariance: [[ 0.00520279 -0.00316897]
             [-0.00316897  0.00393681]] for t: 5
```

```
    Empirical Mean: [1.04230044 3.13710552] for t: 10
    Covariance: [[0.01025536 0.00102201]
                 [0.00102201 0.00298339]] for t: 10

    Empirical Mean: [-0.98582194  3.20338249] for t: 15
    Covariance: [[0.00455407 0.0039209 ]
                 [0.0039209  0.00825629]] for t: 15

    Empirical Mean: [-1.64572697  1.19210524] for t: 20
    Covariance: [[ 0.00249266 -0.00412511]
                 [-0.00412511  0.0089683 ]] for t: 20
```

Listing 14: Empirical Mean and Co variances

If we compare it with the mean and co variances from 3.6 we can see that as the particle filter iterates over time it's empirical mean and co variances among the variables decreases significantly.